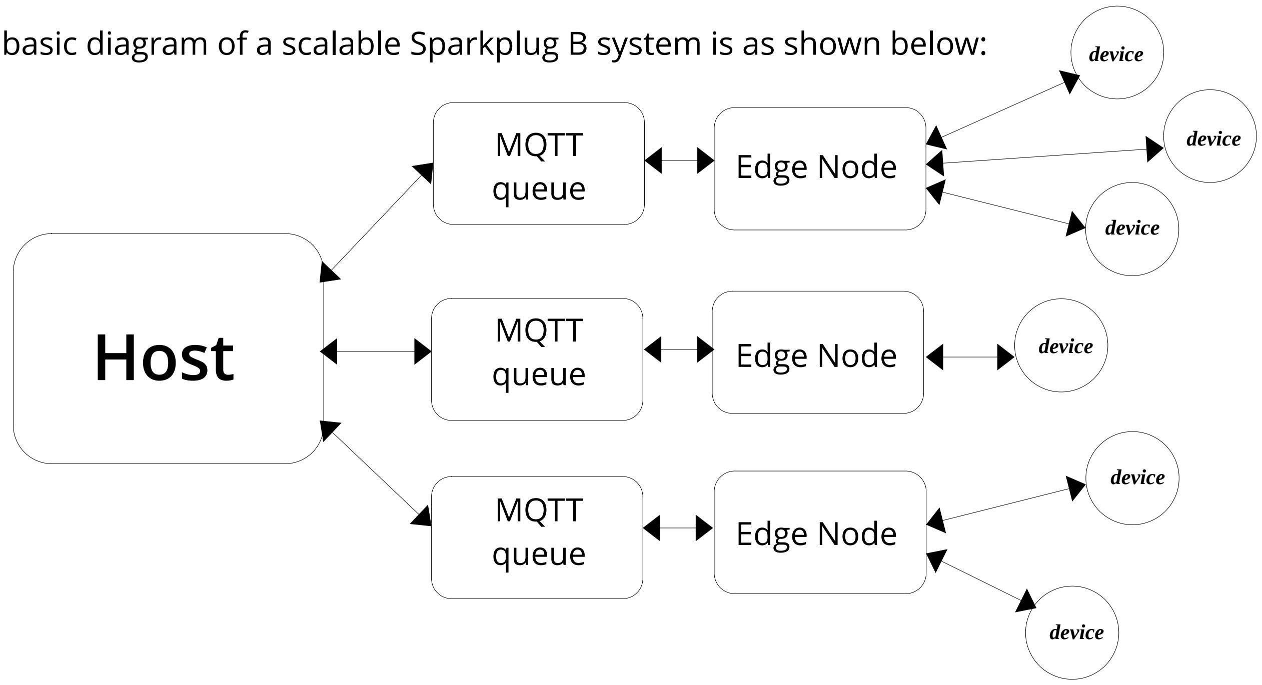
Introduction

Sparkplug B is a communication protocol developed for IOT devices. The protocol itself builds on top of MQTT and Google's Protobuf protocols. Sparkplug uses MQTT to transport the messages between multiple devices in a Sparkplug network. All of the data communicated between the devices, except for a few status messages, are encoded in a few specific Protobuf formats.

The basic diagram of a scalable Sparkplug B system is as shown below:



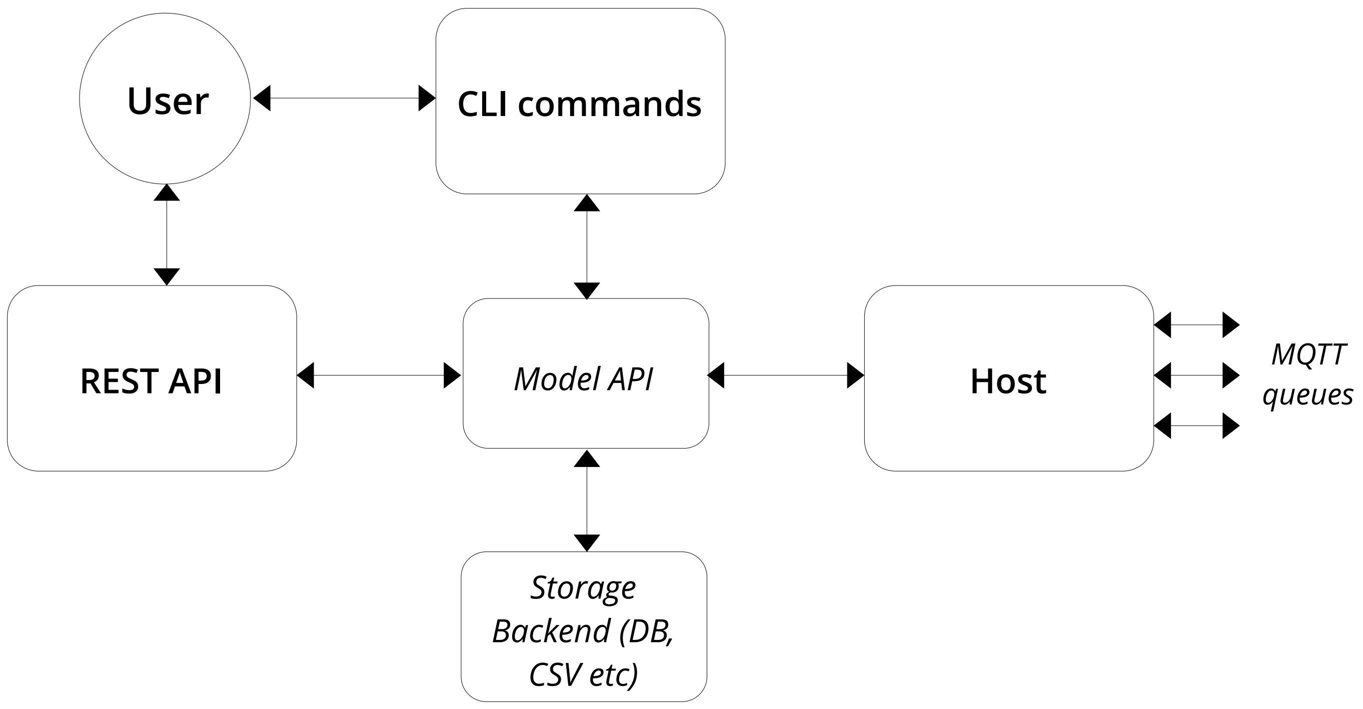
The system consists of a few edge nodes, one or more MQTT queues, and a Sparkplug host. Each of the IOT devices interfaces with one of the edge nodes in the system. It is the job of the edge node to gather raw data from the devices, encode it in the specified Protobuf format, and then put it in the MQTT queue for the host to consume as per the Sparkplug B specification. To ensure a scalable system, there can be multiple MQTT queues where the edge nodes can put their messages on. However, each edge node must connect to a single MQTT queue.

The Sparkplug host consumes the data sent by the edge nodes, and gathers it for further processing by the user. In a Sparkplug environment, the host is the only interface that an user can directly interact with.

The purpose of this challenge was to implement a Sparkplug B compliant host which can collect the data sent by the devices, and provide a few interfaces for an user to query the data. The details of the implementation of the host is specified on the next page.

Implementation

The Sparkplug B host side system implemented in this particular project has the following architecture:



The host system acts a bridge between the MQTT queues and the storage. It listens for various (N/D)BIRTH, (N/D)DEATH, (N/D)DATA publications, and stores the information to the storage backend via the use of the internal model APIs. The host spawns a handler for each of the specified message queues in the configuration file. These handlers talk to the model API asynchronously whenever they receive data in their respective queue.

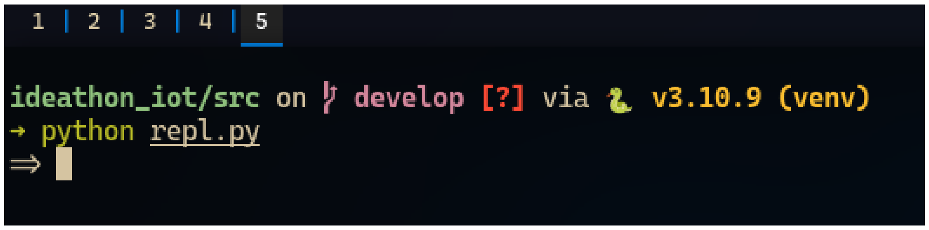
The model API provides an interface for all other systems to talk to the storage backend, without the knowledge of how the storage is actually implemented. Right now, the storage backend is implemented as a ***sqlite*** DB. The storage APIs implement the actual SQL queries that are needed to be run for any particular action to happen. The rest of the system only calls the functions provided by the model API to get their respective data. By implementing the same functions for another backend, the storaging solution can be switched without any change to the host, CLI or the REST modules.

This project provides two ways of interaction to the storage by the user - via the use of a command line REPL (Read – Eval – Print Loop), or via the use of a REST API. All three of the host system, CLI and REST interfaces are independent modules and can be run without any dependency on the other. The next section will show how the interaction using the various interfaces take place in this particular implementation.

Interactions

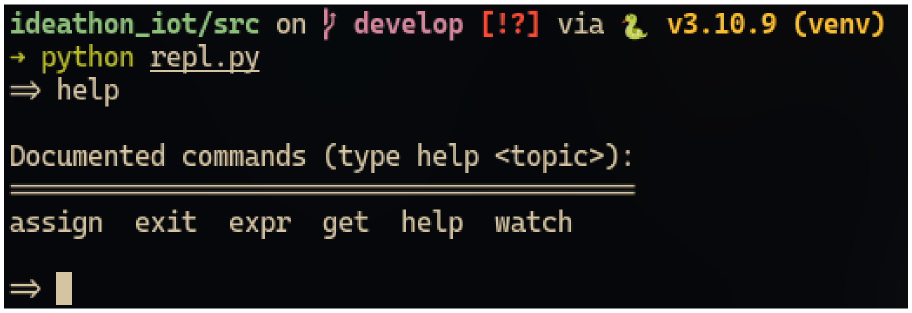
1. Command Line

The system provides a command line interface for querying the data, and in future sending commands to the nodes and/or devices.
Launching the CLI gives an user the following prompt:



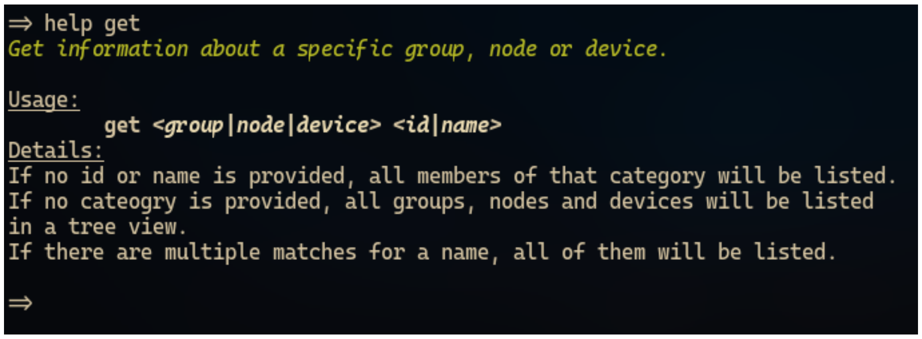
```
 1 | 2 | 3 | 4 | 5

ideathon_iot/src on  develop [?] via 🐍 v3.10.9 (venv)
➜ python repl.py
⇒ 
```

To get a list of commands to run, we start with `help`:



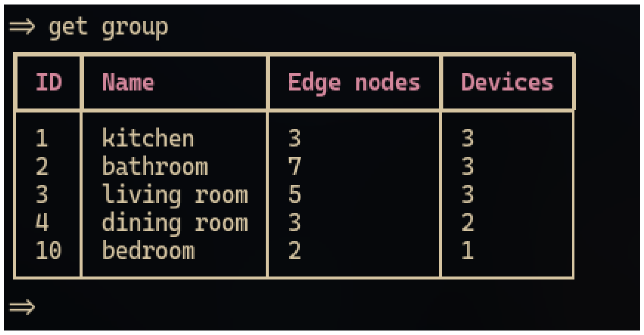
```
ideathon_iot/src on  develop [!?] via 🐍 v3.10.9 (venv)
➜ python repl.py
⇒ help

Documented commands (type help <topic>):
════════════════════════════════════════
assign  exit  expr  get  help  watch

⇒ 
```

To view help for a particular command, we use `help <command>`. Let's start with `help get`.



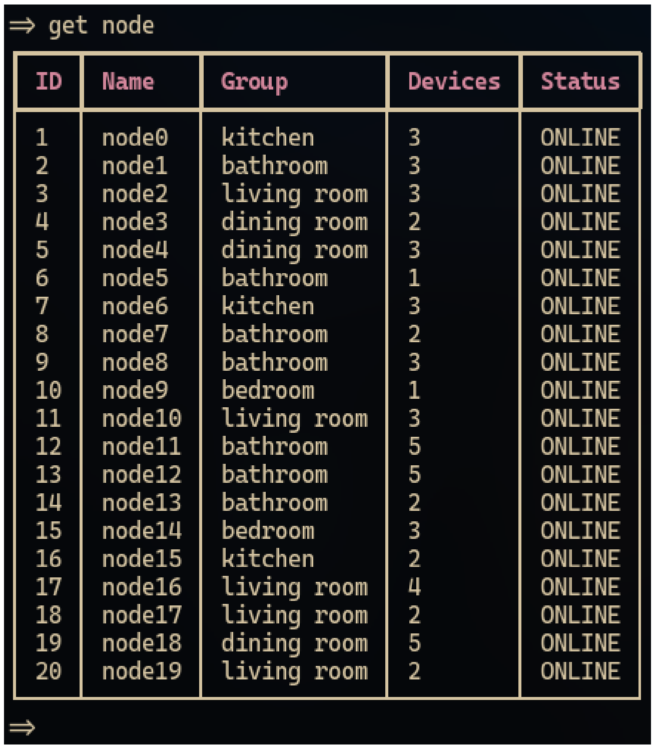
```
⇒ help get
Get information about a specific group, node or device.

Usage:
        get <group|node|device> <id|name>
Details:
If no id or name is provided, all members of that category will be listed.
If no cateogry is provided, all groups, nodes and devices will be listed
in a tree view.
If there are multiple matches for a name, all of them will be listed.

⇒
```

As per the instructions, `get` retrieves details for a particular group/node/device. Let's try `get group`.



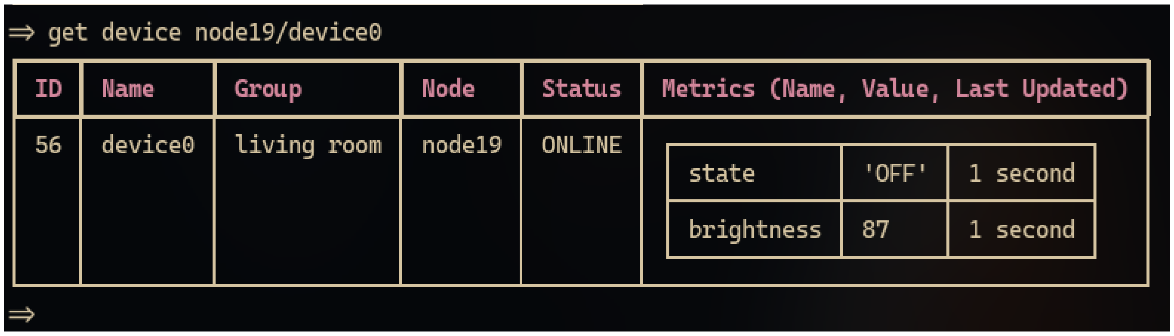
```
⇒ get group
┌────┬─────────────┬────────────┬─────────┐
│ ID │ Name        │ Edge nodes │ Devices │
├────┼─────────────┼────────────┼─────────┤
│ 1  │ kitchen     │ 3          │ 3       │
│ 2  │ bathroom    │ 7          │ 3       │
│ 3  │ living room │ 5          │ 3       │
│ 4  │ dining room │ 3          │ 2       │
│ 10 │ bedroom     │ 2          │ 1       │
└────┴─────────────┴────────────┴─────────┘
⇒
```

To get the list of all nodes, we can use **`get node`**.



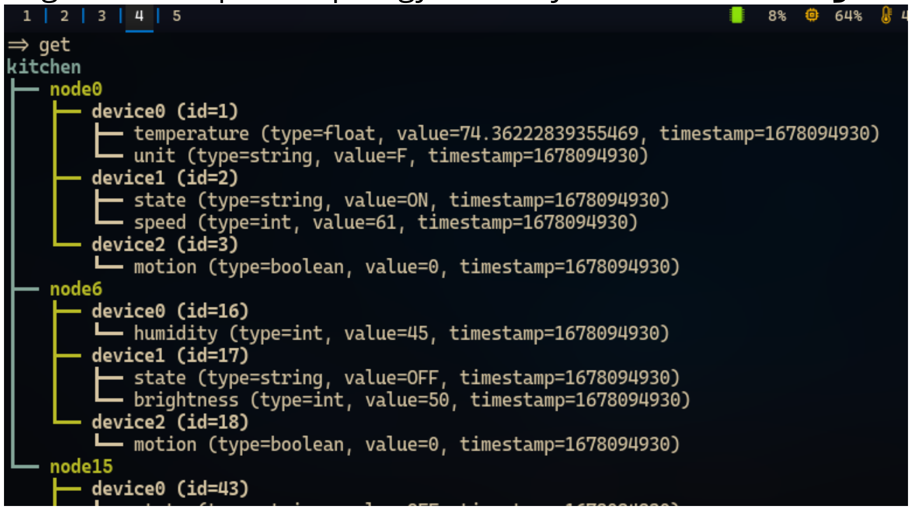
```
⇒ get node
┌────┬────────┬─────────────┬─────────┬────────┐
│ ID │ Name   │ Group       │ Devices │ Status │
├────┼────────┼─────────────┼─────────┼────────┤
│ 1  │ node0  │ kitchen     │ 3       │ ONLINE │
│ 2  │ node1  │ bathroom    │ 3       │ ONLINE │
│ 3  │ node2  │ living room │ 3       │ ONLINE │
│ 4  │ node3  │ dining room │ 2       │ ONLINE │
│ 5  │ node4  │ dining room │ 3       │ ONLINE │
│ 6  │ node5  │ bathroom    │ 1       │ ONLINE │
│ 7  │ node6  │ kitchen     │ 3       │ ONLINE │
│ 8  │ node7  │ bathroom    │ 2       │ ONLINE │
│ 9  │ node8  │ bathroom    │ 3       │ ONLINE │
│ 10 │ node9  │ bedroom     │ 1       │ ONLINE │
│ 11 │ node10 │ living room │ 3       │ ONLINE │
│ 12 │ node11 │ bathroom    │ 5       │ ONLINE │
│ 13 │ node12 │ bathroom    │ 5       │ ONLINE │
│ 14 │ node13 │ bathroom    │ 2       │ ONLINE │
│ 15 │ node14 │ bedroom     │ 3       │ ONLINE │
│ 16 │ node15 │ kitchen     │ 2       │ ONLINE │
│ 17 │ node16 │ living room │ 4       │ ONLINE │
│ 18 │ node17 │ living room │ 2       │ ONLINE │
│ 19 │ node18 │ dining room │ 5       │ ONLINE │
│ 20 │ node19 │ living room │ 2       │ ONLINE │
└────┴────────┴─────────────┴─────────┴────────┘
⇒
```

To get the details of a specific device, **`get [group_name/]node_name/device_name`** can be used like the following.



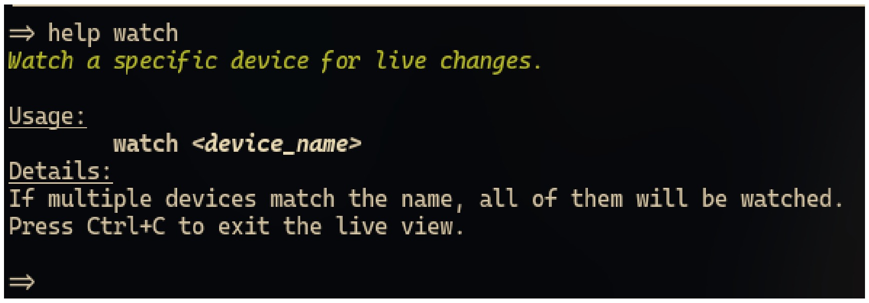
```
⇒ get device node19/device0
┌────┬─────────┬─────────────┬────────┬────────┬───────────────────────────────────────┐
│ ID │ Name    │ Group       │ Node   │ Status │ Metrics (Name, Value, Last Updated)   │
├────┼─────────┼─────────────┼────────┼────────┼───────────────────────────────────────┤
│ 56 │ device0 │ living room │ node19 │ ONLINE │  ┌────────────┬───────┬──────────┐    │
│    │         │             │        │        │  │ state      │ 'OFF' │ 1 second │    │
│    │         │             │        │        │  ├────────────┼───────┼──────────┤    │
│    │         │             │        │        │  │ brightness │ 87    │ 1 second │    │
│    │         │             │        │        │  └────────────┴───────┴──────────┘    │
└────┴─────────┴─────────────┴────────┴────────┴───────────────────────────────────────┘
⇒
```

To get the complete topology of the system, we can run **`get`** without any arguments:



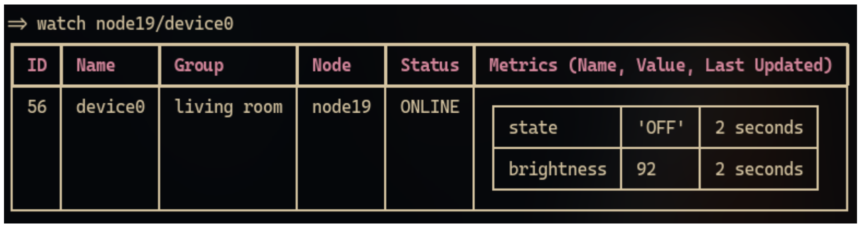
```
 1 | 2 | 3 | 4 | 5                                                  8%    64%   4
⇒ get
kitchen
├── node0
│   ├── device0 (id=1)
│   │   ├── temperature (type=float, value=74.36222839355469, timestamp=1678094930)
│   │   └── unit (type=string, value=F, timestamp=1678094930)
│   ├── device1 (id=2)
│   │   ├── state (type=string, value=ON, timestamp=1678094930)
│   │   └── speed (type=int, value=61, timestamp=1678094930)
│   └── device2 (id=3)
│       └── motion (type=boolean, value=0, timestamp=1678094930)
├── node6
│   ├── device0 (id=16)
│   │   └── humidity (type=int, value=45, timestamp=1678094930)
│   ├── device1 (id=17)
│   │   ├── state (type=string, value=OFF, timestamp=1678094930)
│   │   └── brightness (type=int, value=50, timestamp=1678094930)
│   └── device2 (id=18)
│       └── motion (type=boolean, value=0, timestamp=1678094930)
└── node15
    ├── device0 (id=43)
```

The list will contain all the groups, nodes and devices in the system in a tree view.
The next command that we can look at, is named `watch`.



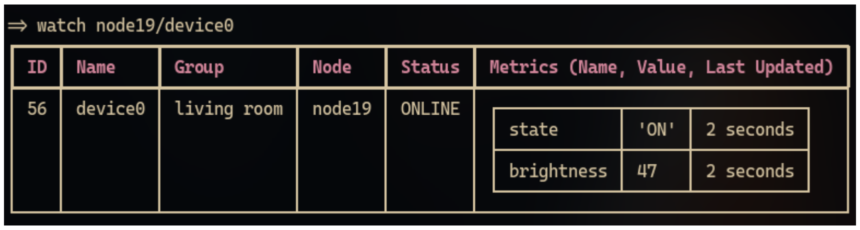
```
⇒ help watch
Watch a specific device for live changes.

Usage:
        watch <device_name>
Details:
If multiple devices match the name, all of them will be watched.
Press Ctrl+C to exit the live view.

⇒
```

Running `watch node19/device0` gives the following display:



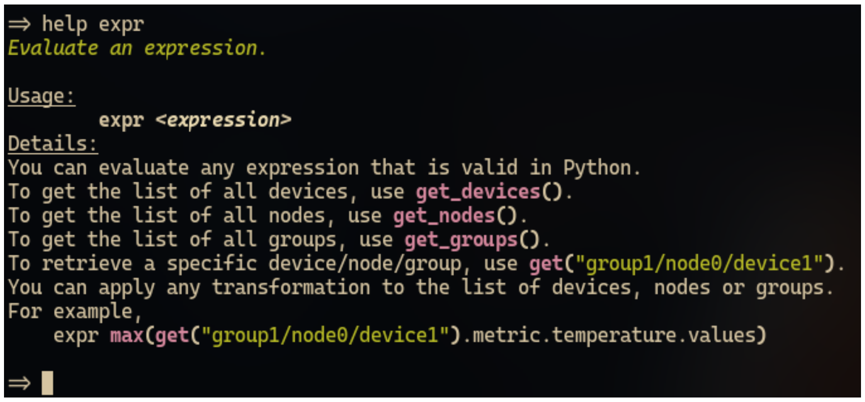
```
⇒ watch node19/device0
┌────┬─────────┬─────────────┬────────┬────────┬─────────────────────────────────────┐
│ ID │ Name    │ Group       │ Node   │ Status │ Metrics (Name, Value, Last Updated) │
├────┼─────────┼─────────────┼────────┼────────┼─────────────────────────────────────┤
│ 56 │ device0 │ living room │ node19 │ ONLINE │                                     │
│    │         │             │        │        │  ┌────────────┬───────┬───────────┐ │
│    │         │             │        │        │  │ state      │ 'OFF' │ 2 seconds │ │
│    │         │             │        │        │  ├────────────┼───────┼───────────┤ │
│    │         │             │        │        │  │ brightness │ 92    │ 2 seconds │ │
│    │         │             │        │        │  └────────────┴───────┴───────────┘ │
└────┴─────────┴─────────────┴────────┴────────┴─────────────────────────────────────┘
```

The view gets automatically updated after 1 second, notice the `Metrics` column.



```
⇒ watch node19/device0
┌────┬─────────┬─────────────┬────────┬────────┬─────────────────────────────────────┐
│ ID │ Name    │ Group       │ Node   │ Status │ Metrics (Name, Value, Last Updated) │
├────┼─────────┼─────────────┼────────┼────────┼─────────────────────────────────────┤
│ 56 │ device0 │ living room │ node19 │ ONLINE │                                     │
│    │         │             │        │        │  ┌────────────┬──────┬───────────┐  │
│    │         │             │        │        │  │ state      │ 'ON' │ 2 seconds │  │
│    │         │             │        │        │  ├────────────┼──────┼───────────┤  │
│    │         │             │        │        │  │ brightness │ 47   │ 2 seconds │  │
│    │         │             │        │        │  └────────────┴──────┴───────────┘  │
└────┴─────────┴─────────────┴────────┴────────┴─────────────────────────────────────┘
```

To query the device details and perform computation on their results, the REPL provides a command named `expr`:

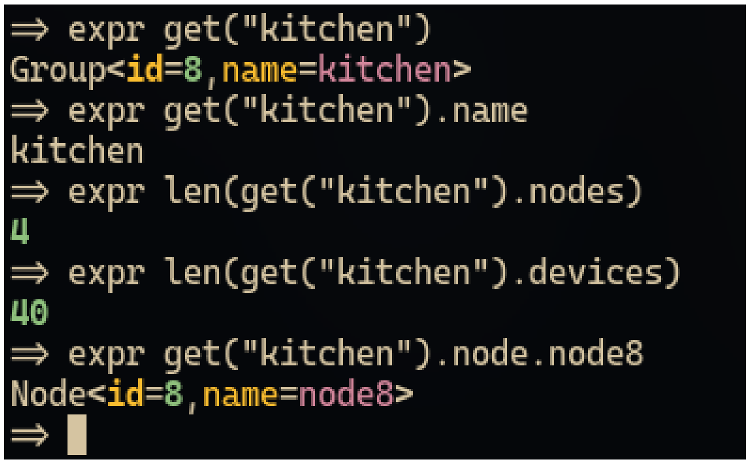


```
⇒ help expr
Evaluate an expression.

Usage:
        expr <expression>
Details:
You can evaluate any expression that is valid in Python.
To get the list of all devices, use get_devices().
To get the list of all nodes, use get_nodes().
To get the list of all groups, use get_groups().
To retrieve a specific device/node/group, use get("group1/node0/device1").
You can apply any transformation to the list of devices, nodes or groups.
For example,
    expr max(get("group1/node0/device1").metric.temperature.values)

⇒ █
```

The expr command can run any valid Python expression, and provides a small set of functions on its own to get the user started. `expr` provides direct access to the model API and in turn, to the storage backend of the system.
Beyond standard python types, any `expr` expression can return an object of any one of these types: `Group`, `Node`, `Device` or `Metric`.

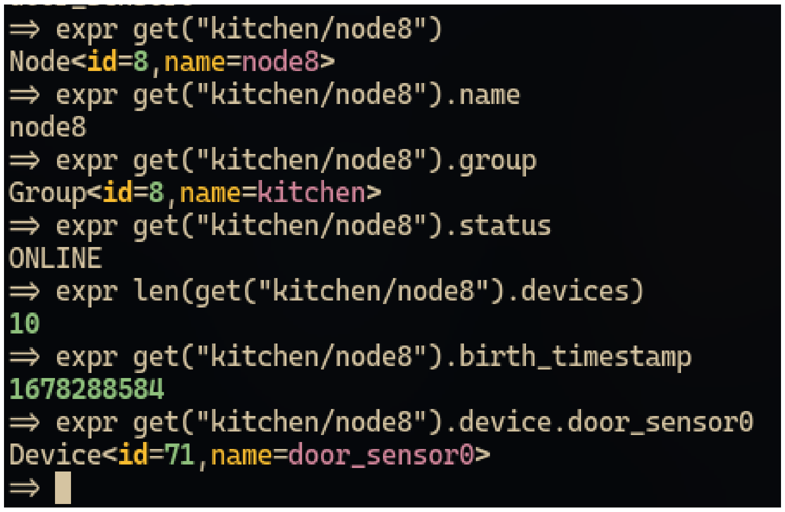
An object of any of these classes has some distinct properties of their own, which are listed below:

1. `Group`:
   a. `name`: Name of the group
   b. `nodes`: List of `Node`s in the group
   c. `devices`: List of `Device`s in the group
   d. `node`: Selector for a specific `Node` in the group



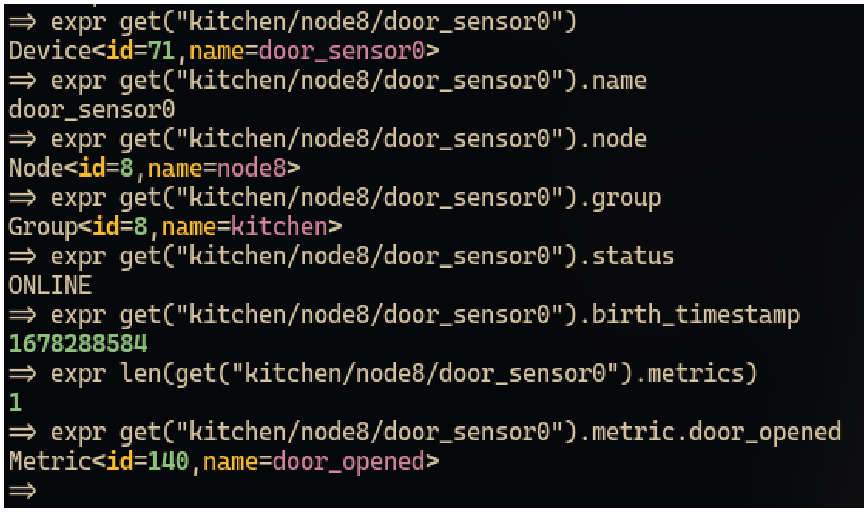
```
⇒ expr get("kitchen")
Group<id=8,name=kitchen>
⇒ expr get("kitchen").name
kitchen
⇒ expr len(get("kitchen").nodes)
4
⇒ expr len(get("kitchen").devices)
40
⇒ expr get("kitchen").node.node8
Node<id=8,name=node8>
⇒
```

2. `Node`:
   a. `name`: Name of the node
   b. `group`: `Group` that this node belongs to
   c. `status`: Online status of this node
   d. `devices`: List of `Device`s for this node
   e. `birth_timestamp/death_timestamp`: Timestamp in UTC of this node's birth/death
   f. `device`: Selector for a specific `Device` in the node



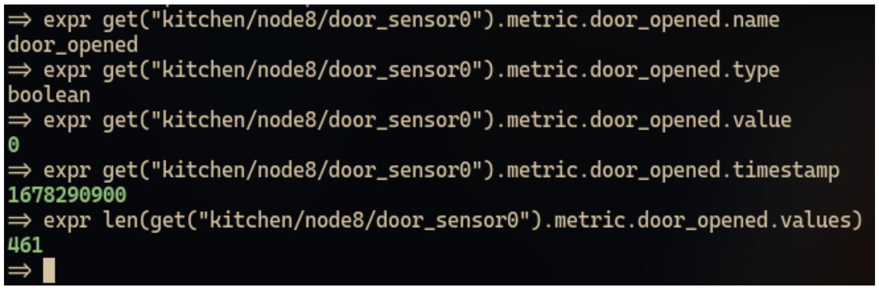
```
⇒ expr get("kitchen/node8")
Node<id=8,name=node8>
⇒ expr get("kitchen/node8").name
node8
⇒ expr get("kitchen/node8").group
Group<id=8,name=kitchen>
⇒ expr get("kitchen/node8").status
ONLINE
⇒ expr len(get("kitchen/node8").devices)
10
⇒ expr get("kitchen/node8").birth_timestamp
1678288584
⇒ expr get("kitchen/node8").device.door_sensor0
Device<id=71,name=door_sensor0>
⇒
```

3. **`Device`**:
   a. **`name`**: Name of the device
   b. **`group`**: The particular **`Group`** this device belongs to
   c. **`node`**: The particular **`Node`** this device belongs to
   d. **`status`**: Online status of this device
   e. **`birth_timestamp/death_timestamp`**: Timestamp in UTC of this device's
birth/death
   f. **`metrics`**: List of **`Metric`**s belonging to this particular device
   g. **`metric`**: Selector for a specific **`Metric`** of this device



```
⇒ expr get("kitchen/node8/door_sensor0")
Device<id=71,name=door_sensor0>
⇒ expr get("kitchen/node8/door_sensor0").name
door_sensor0
⇒ expr get("kitchen/node8/door_sensor0").node
Node<id=8,name=node8>
⇒ expr get("kitchen/node8/door_sensor0").group
Group<id=8,name=kitchen>
⇒ expr get("kitchen/node8/door_sensor0").status
ONLINE
⇒ expr get("kitchen/node8/door_sensor0").birth_timestamp
1678288584
⇒ expr len(get("kitchen/node8/door_sensor0").metrics)
1
⇒ expr get("kitchen/node8/door_sensor0").metric.door_opened
Metric<id=140,name=door_opened>
⇒
```

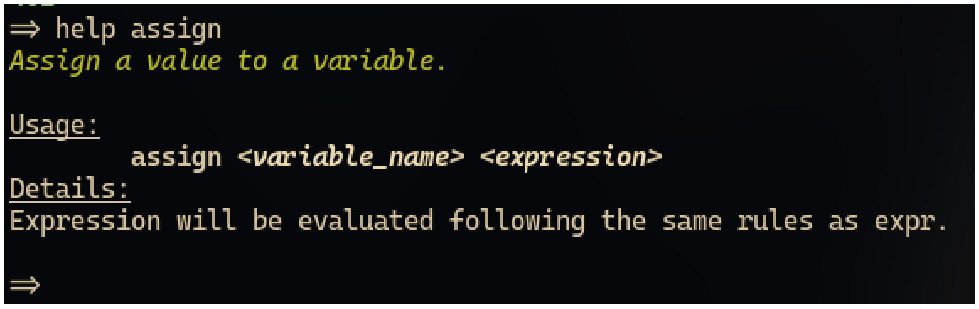
4. **`Metric`**:
   a. **`name`**: Name of the metric
   b. **`type`**: Type of the metric
   c. **`value`**: Last recorded value of the metric
   d. **`timestamp`**: UTC timestamp of the last recorded value of the metric
   e. **`values`**: A list of all (value, timestamp) tuples for the metric that has been recorded



```
⇒ expr get("kitchen/node8/door_sensor0").metric.door_opened.name
door_opened
⇒ expr get("kitchen/node8/door_sensor0").metric.door_opened.type
boolean
⇒ expr get("kitchen/node8/door_sensor0").metric.door_opened.value
0
⇒ expr get("kitchen/node8/door_sensor0").metric.door_opened.timestamp
1678290900
⇒ expr len(get("kitchen/node8/door_sensor0").metric.door_opened.values)
461
⇒
```

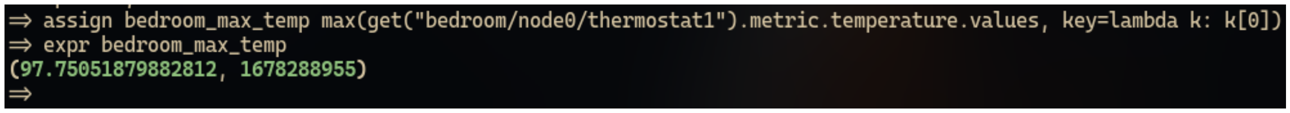
Internally, all of the modules use the same API to provide every functionality regarding all aspects of this system. So, integration with any other module in a larger system will simply consist of importing the model API, and using regular Python expressions to interact with the storage backend.

One other command that this REPL implements is named **`assign`**:



```
⇒ help assign
Assign a value to a variable.

Usage:
        assign <variable_name> <expression>
Details:
Expression will be evaluated following the same rules as expr.

⇒
```

`assign` allows the user to store the result of an expression in a variable for later use. For example, to get the highest recorded temperature in bedroom, we can use the following expression and assign it to a variable:



```
⇒ assign bedroom_max_temp max(get("bedroom/node0/thermostat1").metric.temperature.values, key=lambda k: k[0])
⇒ expr bedroom_max_temp
(97.75051879882812, 1678288955)
⇒
```

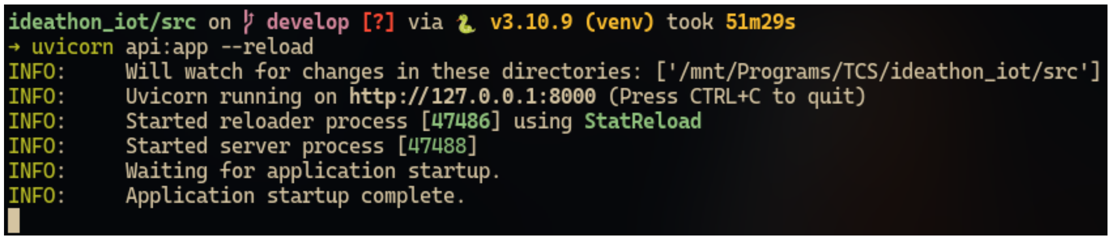
Another way of interacting to the system is via the REST APIs, which will be shown in the next section.

Interactions

2. REST API

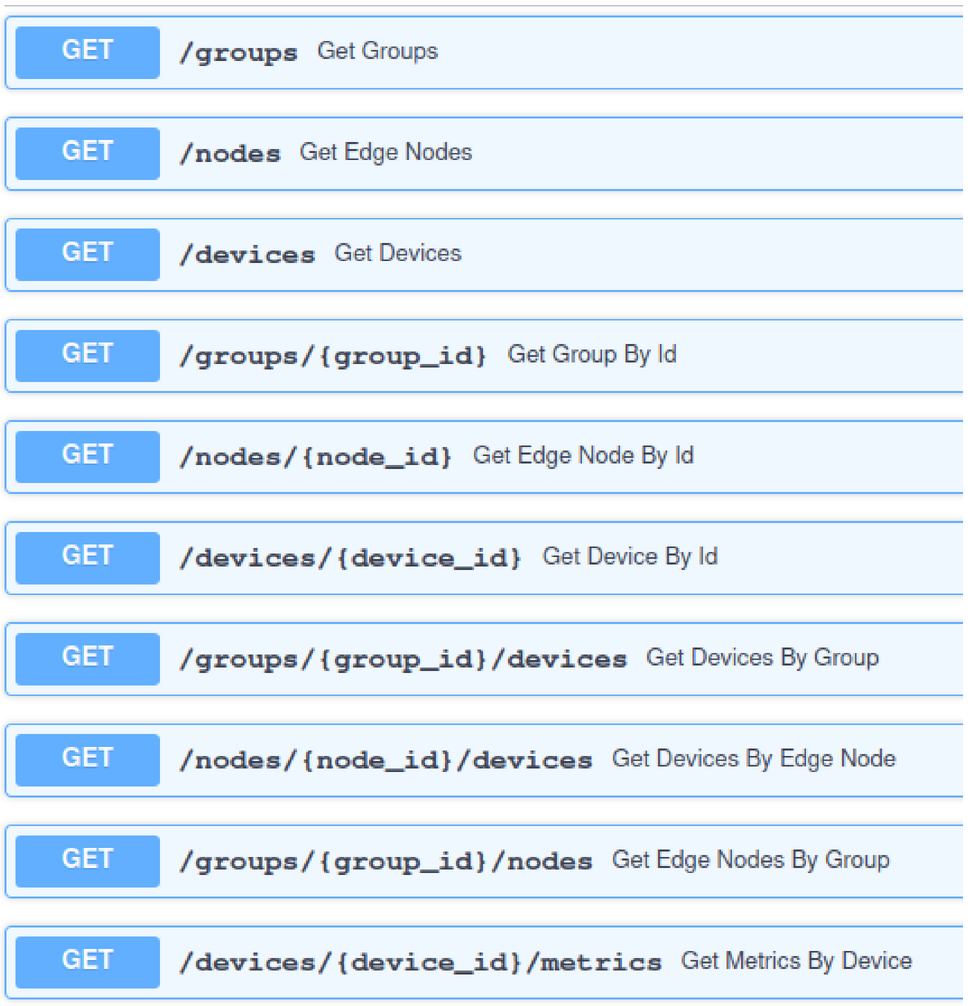
The system provides a REST API based on FastAPI which provides several endpoints for querying data.

The server can be started like the following, by using **`uvicorn`** as an example,



```
ideathon_iot/src on ᚴ develop [?] via 🐍 v3.10.9 (venv) took 51m29s
➜ uvicorn api:app --reload
INFO:     Will watch for changes in these directories: ['/mnt/Programs/TCS/ideathon_iot/src']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [47486] using StatReload
INFO:     Started server process [47488]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

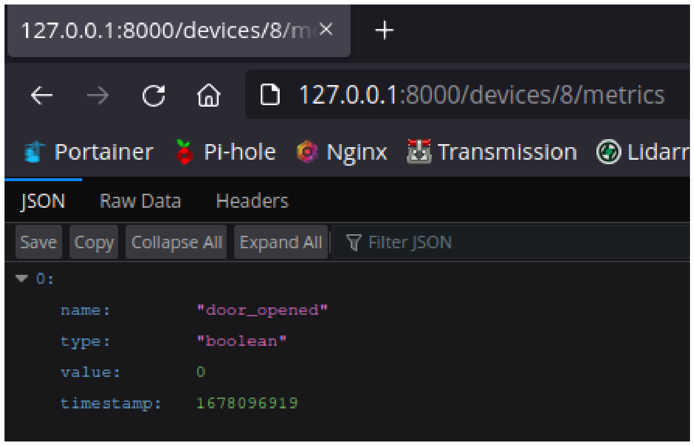
The following endpoints are provided by the API:



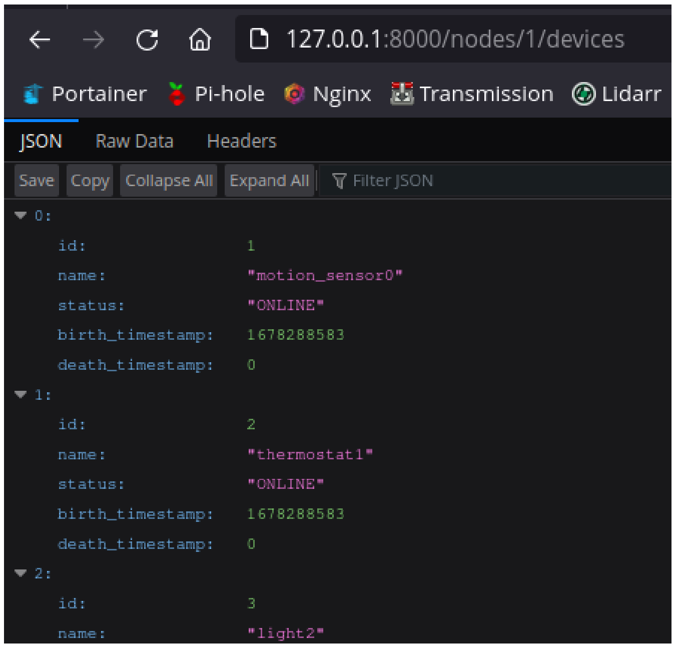
```
GET /groups Get Groups
GET /nodes Get Edge Nodes
GET /devices Get Devices
GET /groups/{group_id} Get Group By Id
GET /nodes/{node_id} Get Edge Node By Id
GET /devices/{device_id} Get Device By Id
GET /groups/{group_id}/devices Get Devices By Group
GET /nodes/{node_id}/devices Get Devices By Edge Node
GET /groups/{group_id}/nodes Get Edge Nodes By Group
GET /devices/{device_id}/metrics Get Metrics By Device
```

The REST API can be extended indepedently, and can also have methods to send messages to the devices.

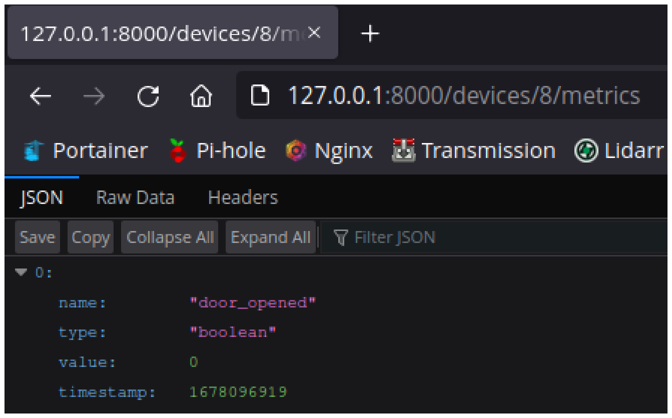
Example of the outputs given by the API endpoints are shown below:



```
▼ 0:
    name:        "door_opened"
    type:        "boolean"
    value:       0
    timestamp:   1678096919
```



```
▼ 0:
    id:                1
    name:              "motion_sensor0"
    status:            "ONLINE"
    birth_timestamp:   1678288583
    death_timestamp:   0
▼ 1:
    id:                2
    name:              "thermostat1"
    status:            "ONLINE"
    birth_timestamp:   1678288583
    death_timestamp:   0
▼ 2:
    id:                3
    name:              "light2"
```



```
▼ 0:
    name:        "door_opened"
    type:        "boolean"
    value:       0
    timestamp:   1678096919
```

All of the endpoints return JSON responses in the *key:value* format.

Configuration

The application can be configured with a configuration file named `config.json`, which should be present in the same directory as the source files. The file is read by the all the modules in the architecture. The configuration file looks like the following right now: *(comments are not officially supported in JSONs, used here just for the explanation of the config options)*

```
{
 // Ids for the spawned sparkplug hosts. There must be one host for
 // one MQTT server specified in the next section.
 "ids": ["sparkplugHost1", "sparkplugHost2"],
 // List of MQTT servers to connect to. This allows load balance
 // for a high traffic system.
 "mqtt": [{
     "host": "localhost",
     "port": 1883,
     "username": "mqtt",
     "password": "mqtt"
 }, {
     "host": "localhost",
     "port": 1884,
     "username": "mqtt",
     "password": "mqtt"
 }],
 // List of zones in the system. Zones are logical grouping for a
 // Sparkplug environment.
 "zones": [
     "bedroom",
     "kitchen",
     "living_room",
     "bathroom",
     "dining_room"
 ],
 // Connection details for the storage backend
 "db": {
     "type": "sqlite",
     "url": "sparkplug.db",
     "username": "sparkplug",
     "password": "sparkplug"
 },
 // Configuration options for the client simulator
 "client_node_count": 20,
 "client_devices_per_node": 10,
 "client_publish_interval_seconds": 5,
 // The following dictionary declares a list of metrics
 // mapped to their types
 "client_metric_types": {
     "temperature": "float",
     "humidity": "float",
     "now_playing": "string",
     "state": "boolean",
     "volume": "int",
     "brightness": "int",
     "power": "boolean",
     "mode": "string",
     "fan_speed": "int",
     "fan_direction": "string",
     "fan_mode": "string",
     "fan_state": "boolean",
     "motion": "boolean",
     "occupancy": "boolean",
     "door_opened": "boolean"
 },
 // Definition of a few types of devices present in the
 // simulator. Each type of device is mapped to a list
 // of metrics it generates. The simulator reads this list,
 // and randomly generates values of the metrics based on
 // their type as defined in the previous list.
 "client_device_types": {
     "thermostat": {
          "metrics": ["temperature", "humidity", "state"]
     },
     "fan": {
          "metrics": ["fan_speed", "fan_direction", "fan_mode",
"fan_state"]
     },
     "light": {
          "metrics": ["brightness", "power"]
     },
     "speaker": {
          "metrics": ["now_playing", "volume", "power"]
     },
     "motion_sensor": {
          "metrics": ["motion"]
     },
     "occupancy_sensor": {
          "metrics": ["occupancy"]
     },
     "door_sensor": {
          "metrics": ["door_opened"]
     }
 }
}
```

Conclusion

The source code for the application can be found in the following GitHub link:
IdeathonIOT on GitHub

It contains a readme which details how to setup and start the system. It also contains a client program which simulates a vast collection of IOT devices communicating over the Sparkplug B protocol.

This has been a great learning experience, and I want to thank the organizers and the panel for giving me such an oppurtunity to explore the world of IOT devices.