INTEL + 1AND1 IOT HACKATHON SERVICE TUTORIAL

DEVELOPERS CAN CREATE AN END-TO-END SOLUTION THAT STREAMS TELEMETRY DATA TO THE CLOUD AND HAVE THIS STREAMED INTO A SERVICE VIA MQTT TO PERFORM REALTIME ANALYTICS

BACKGROUND

Streammyiot.com is hosting a flavour of the Intel EnableIOT system that allows IOT devices (such as Intel Edison, Intel Joule, Raspberry Pi, etc) to stream data to the cloud and easily plot charts, perform basic operations using the rule engine (e.g. if the temperature exceeds 30 degress, send an email). This system has been extended to allow external services (created by developers) to develop completely custom applications to perform realtime analytics, call third party APIs, etc. Streammyiot.com has been deployed on 1and1 infrastructure and is providing developers with Virtual Machines (VMs) to deploy your services.
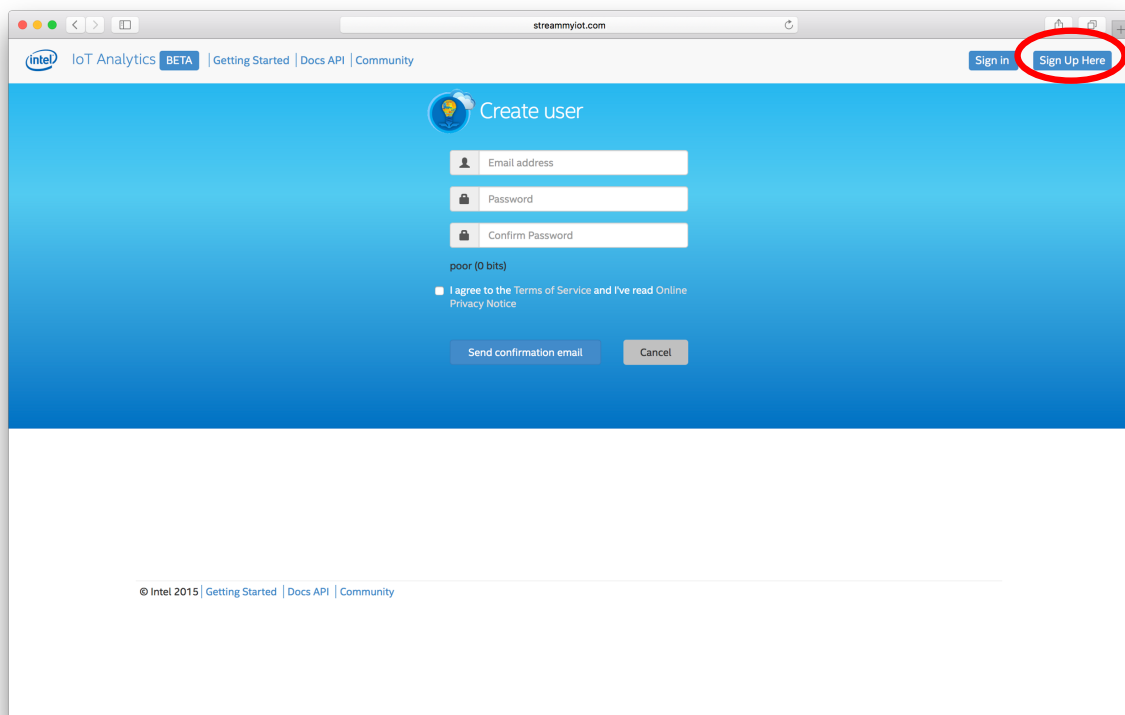
The data flow is as follows (using temperature as an example):

Temperature Sensor -> IOT device -> Agent (running in same IOT device) -> Cloud (Streammyiot.com) via REST -> Rules engine -> MQTT Broker (Mosquitto) -> **Your service (complete processing)** -> Trigger actuation -> Cloud (Streammyiot.com) -> IOT device

This data flow shows a complete end-to-end approach whereby we stream data to the cloud, perform some processing in our service and then can trigger an actuation event in order for the same (although could be a different) IOT device to do something such as turn on/off LED.
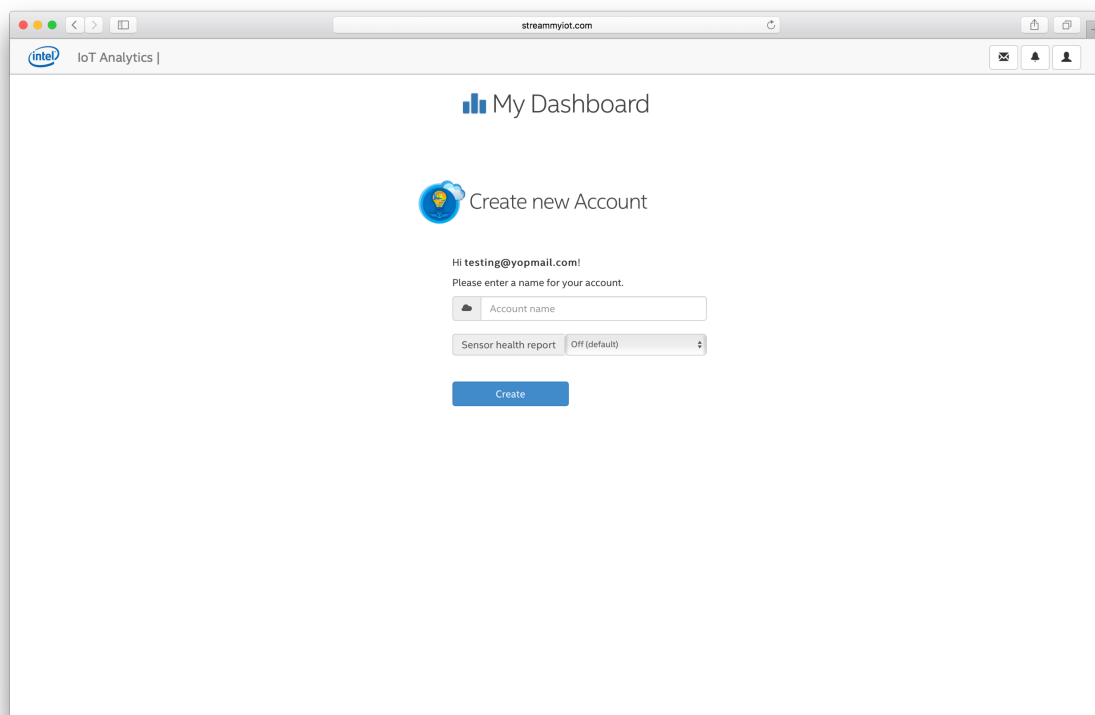
STEP 1 – CREATE A USER ACCOUNT
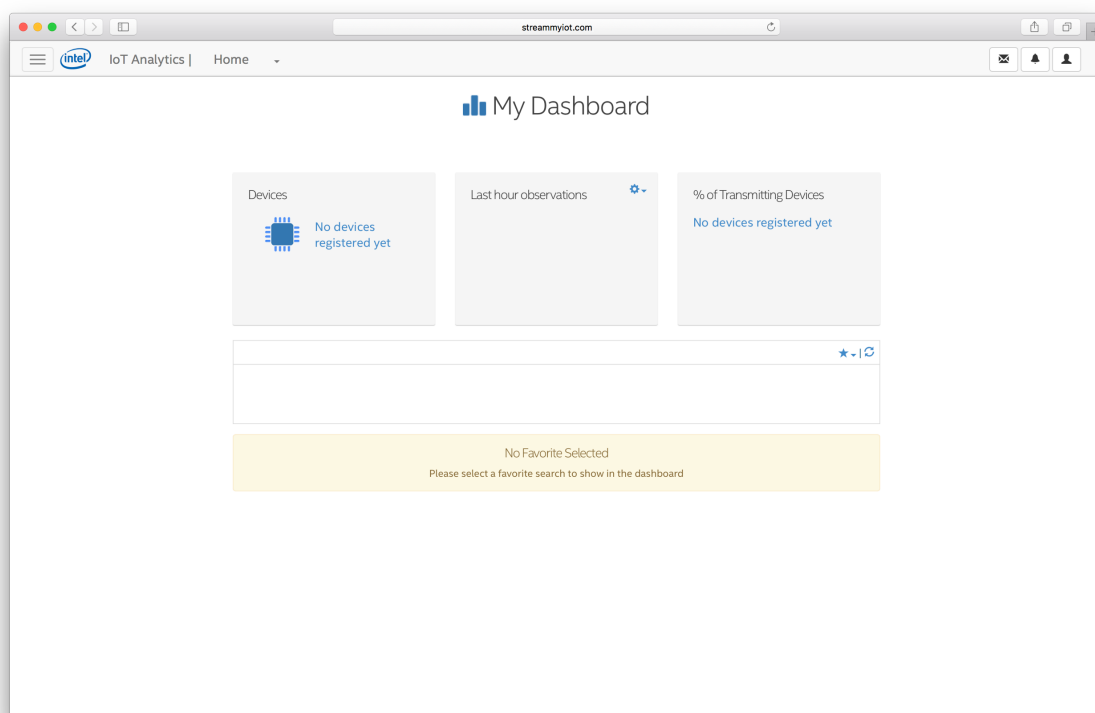
Create an account with streammyiot.com

Once you're received your email confirmation, you can login using 'Sign in' button next to the 'Sign Up Here' button. Once you have logged in, you need to create an account. The platform allows a user to have multiple accounts and each account can hold its collection of IOT devices, rules, charts, etc. You should now see this:

You'll now see a dashboard for your created account and if you create multiple accounts, you can change the dashboard by clicking the Arrow shown in the red circle below:

STEP 2 – CONNECTING YOUR IOT DEVICE TO THE CLOUD

1. Connect your IOT device (which now on in we'll assume is an Intel Edison) via SSH or Serial using Putty (on Windows) or Minicom (on Linux).

2. You should test your connection to the cloud (streammyiot.com) using:
```
iotkit-admin test
```

The response you should receive will be similar to:
```
2017-03-13T14:45:26.352Z - info: Trying to connect to host ...
2017-03-13T14:45:26.409Z - info: Connected to streammyiot.com
2017-03-13T14:45:26.410Z - info: Environment: local
2017-03-13T14:45:26.410Z - info: Build: 0.15.0
2017-03-13T14:45:26.411Z - info: Trying to connect to WS server ...
2017-03-13T14:45:26.471Z - info: Connection to Web Socket Server successful
2017-03-13T14:45:26.491Z - info: Websocket connection closed. Reason: 1000 Normal
connection closure
```

3. You need to activate your Edison by first getting your activation code by clicking the name of your account circled in red below:
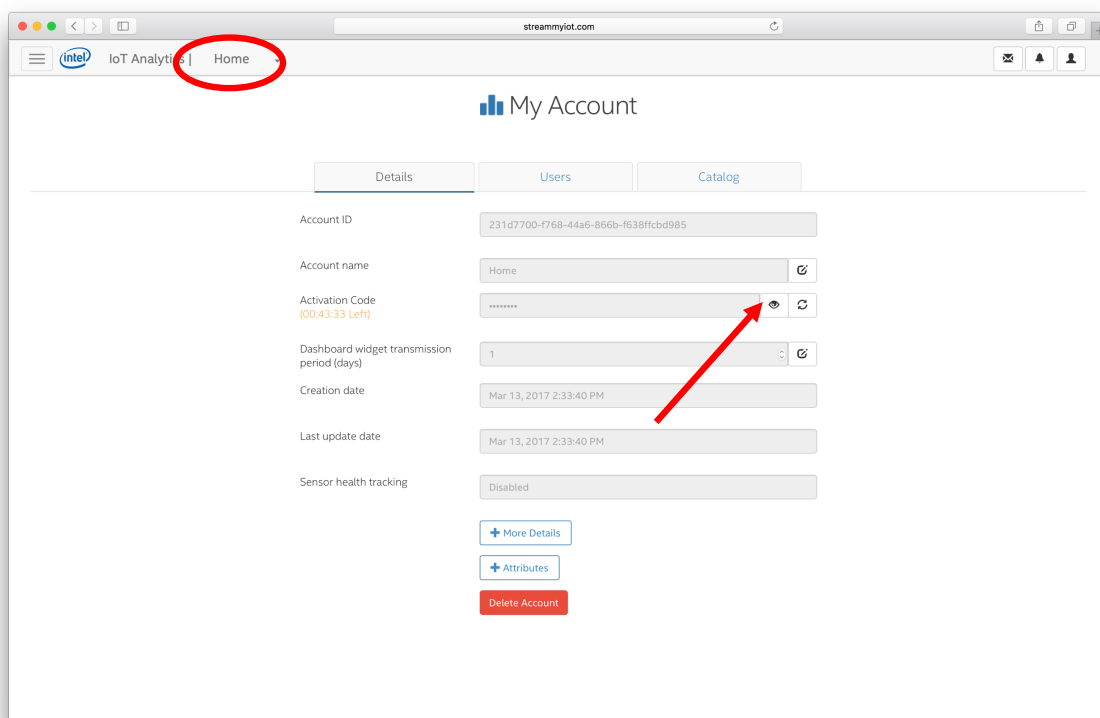
Click the Eye icon to reveal your activation code shown with a red arrow above.

Return to your Edison and set a device ID (e.g. my-name-edison-01):
```
iotkit-admin set-device-id {DEVICE_ID}
```

Activate your Edison and activate your device using:
```
iotkit-admin activate {ACTIVATION_CODE}
```

The response should be similar to:
```
2017-03-13T14:55:14.118Z - info: Activating ...
2017-03-13T14:55:14.203Z - info: Saving device token...
2017-03-13T14:55:14.207Z - info: Updating metadata...
2017-03-13T14:55:14.209Z - info: Metadata updated.
```

4. The above steps configure the agent but we now need to start the agent using:

```
systemctl start iotkit-agent
```

5. You can confirm the device is connected your account by exploring 'My Devices' shown here:
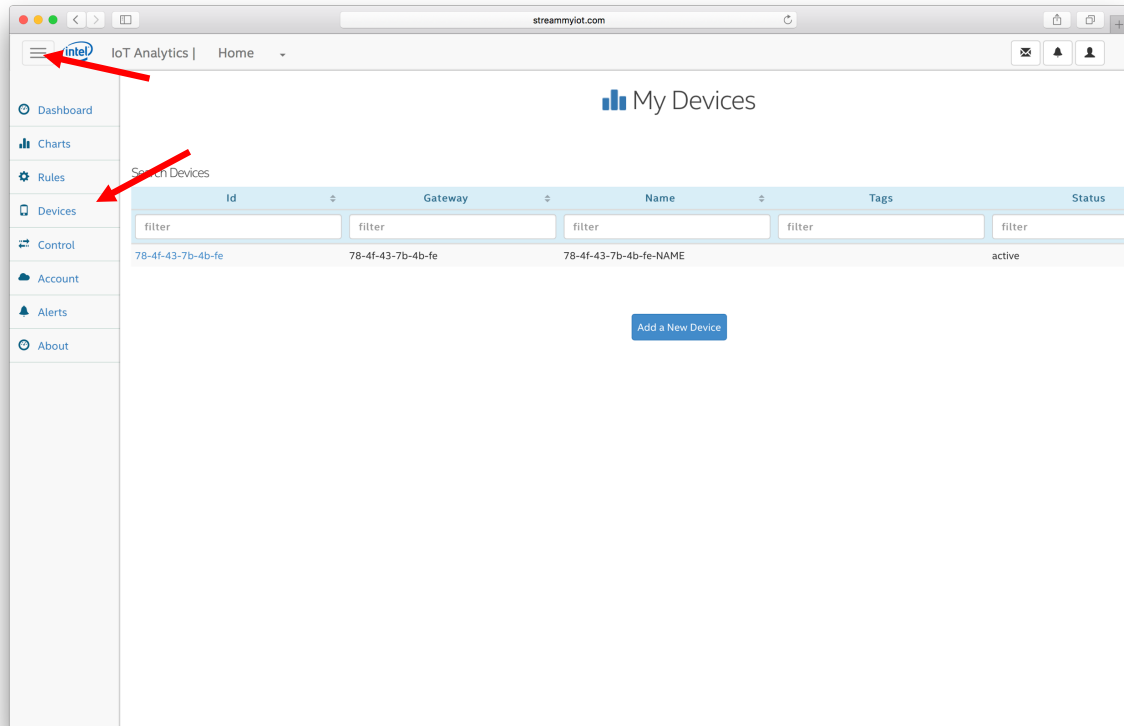
By default the system has given the device a name which is used throughout the platform so it is useful to change it to something more human.

1. When your device sends telemetry data to the cloud, it needs to be sent with what type of data it is. E.g. Temperature, humidity, vibration etc. Each account has a catalog of components and these components correspond to either a type of data being streamed to the cloud (typically a sensor) or an actuator attached to the Edison that the cloud wants to trigger (e.g. a motor, an LED, etc). To view the catalog:

You are able to create your own components and add these to your catalog for other types of data and/or actuators. This tutorial will use the 'temperature.v1.0' catalog item. The Edison must register this catalog item as a component:

```
iotkit-admin register temp temperature.v1.0
```

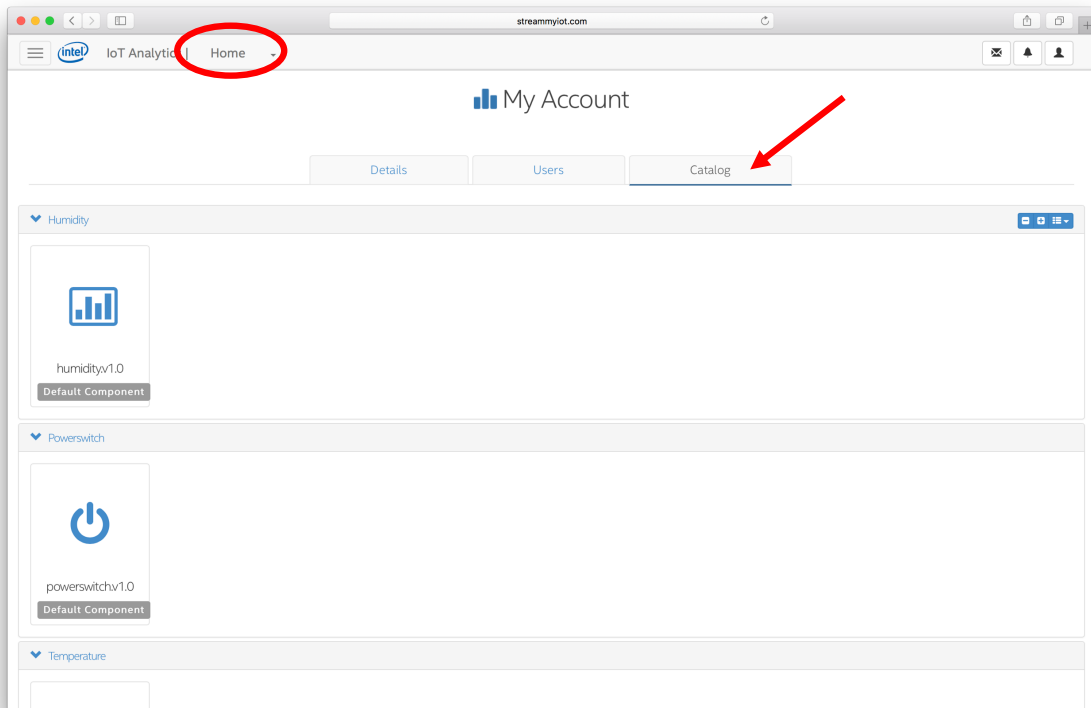Note: we have created our component with the name 'temp', this can be anything of your choice and will be used later.

The response will be similar to:
```
2017-03-13T16:00:58.399Z - info: Starting registration ...
2017-03-13T16:00:58.402Z - info: Device has already been activated. Updating ...
2017-03-13T16:00:58.403Z - info: Updating metadata...
2017-03-13T16:00:58.407Z - info: Metadata updated.
Attributes sent
2017-03-13T16:00:58.617Z - info: Component registered name=temp, type=temperature.v1.0,
cid=e3679d3e-ccd4-465c-9c65-cb9fe1d417e3, deviceId=78-4f-43-7b-4b-fe
```

2. From the command line you can already stream data that is useful for testing purposes:
```
iotkit-admin observation temp 33
```

This response will be similar to:
```
2017-03-13T16:18:09.386Z - info: Submitting:  n=temp, v=33
2017-03-13T16:18:09.479Z - info: Response received:  response=none detail, status=0
2017-03-13T16:18:09.480Z - info: Observation Sent response=none detail, status=0
```

3. Now it's time to develop our application to run on the Edison. This application will data from a sensor (this tutorial focuses on a temperature). Reading data from a sensor can be made easy using the MRAA or UPM libraries. Now we can have the telemetry data we can stream this to the cloud via the agent that is running on the Edison. This is as simple as sending JSON via **UDP** to **localhost** on port **41234**. Here is example JavaScript code:
```
var dgram = require('dgram');
```

```javascript
var PORT = 41234;
var HOST = '127.0.0.1';

var sendObservation = function(data, cb){
      var message = new Buffer(JSON.stringify(data));

      var client = dgram.createSocket('udp4');
      client.send(message, 0, message.length, PORT, HOST, function(err, response){
            cb && cb(err, response);
            client.close();
      });
}

var data = {"n": "temp", "v": 25};

sendObservation(data, function(err, response){
      if (err) throw err;
      console.log("Data sent");
});
```
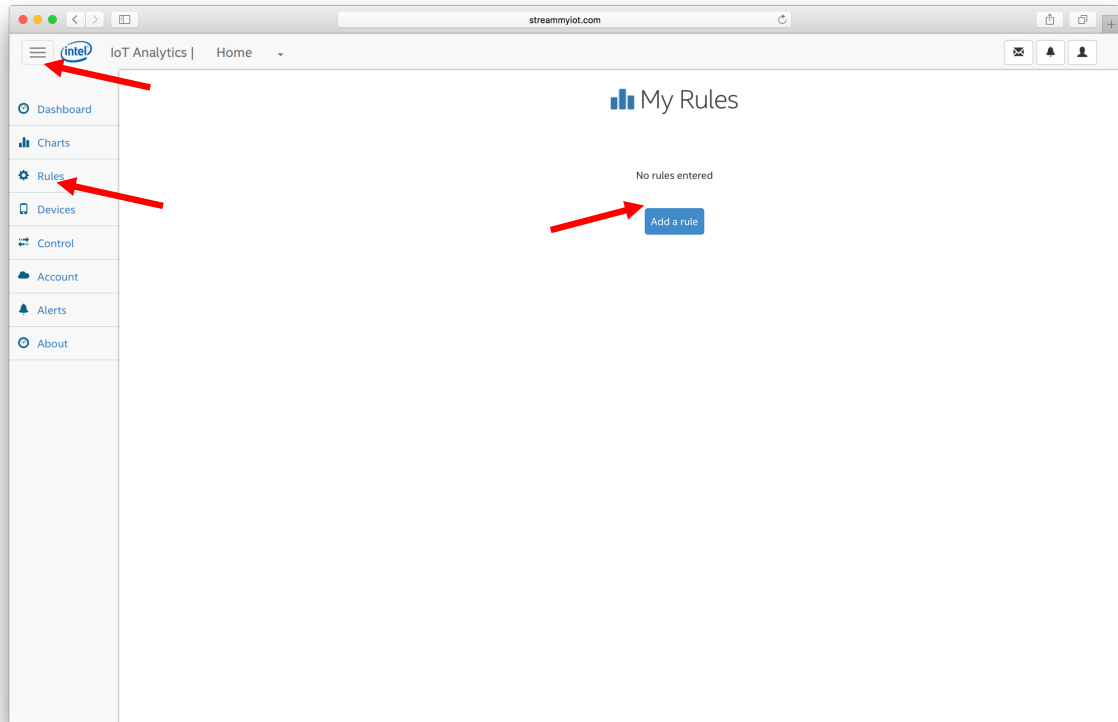
Note: the 'data' variable that contains an object with an 'n' property which corresponds to the name you gave to the component and 'v' which is the value you would like to stream to the agent, and the agent will forward this to the cloud.

1. We can leverage the rules engine to setup a rule that will forward the data is receives to your analytics service:

2. We can create a rule using the Notifications type 'HTTP Endpoint':
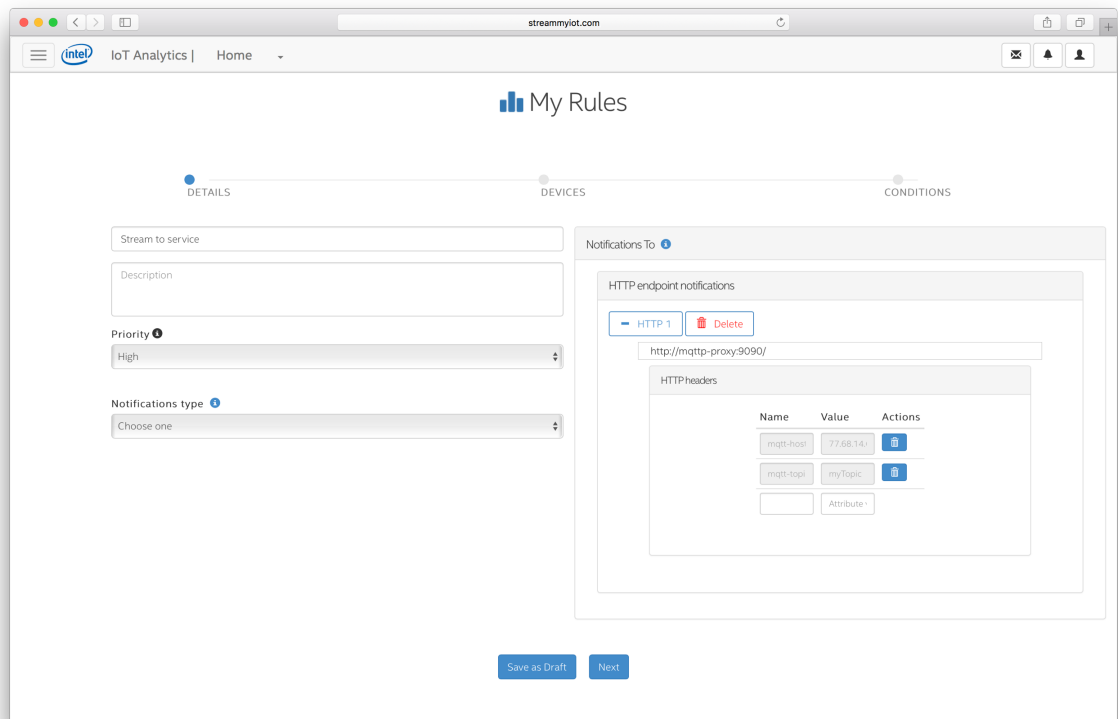
We have created a proxy to handle HTTP requests and redistribute via MQTT.

The HTTP Endpoint url is: http://mqttp-proxy:9090/

Now you need add HTTP headers so the proxy know which MQTT broker to use and which topic to publish to. Add the following headers:

Name: mqtt-host   Value: <hostname or IP address>

Name: mqtt-topic   Value: <your topic name of choice>

Note: You can either use a test MQTT broker (such as test.mosquitto.org) or installing a MQTT broker such as Mosquitto.

Then click 'next':

You must now select the devices that this rule applies to, shown above by clicking the checkbox.

Then click 'Next':

Here you can enter the conditions which when met will forward the telemetry data to your analytics service. Ensure click the checkbox for 'Enable Automatic Reset' otherwise the rule will only get executed once. Select the 'Monitored Measure', for this example it will be 'temp'. Then select a 'Basic condition', use the '>' operator and then enter '0' for the value. Click 'Done' and see your rule:
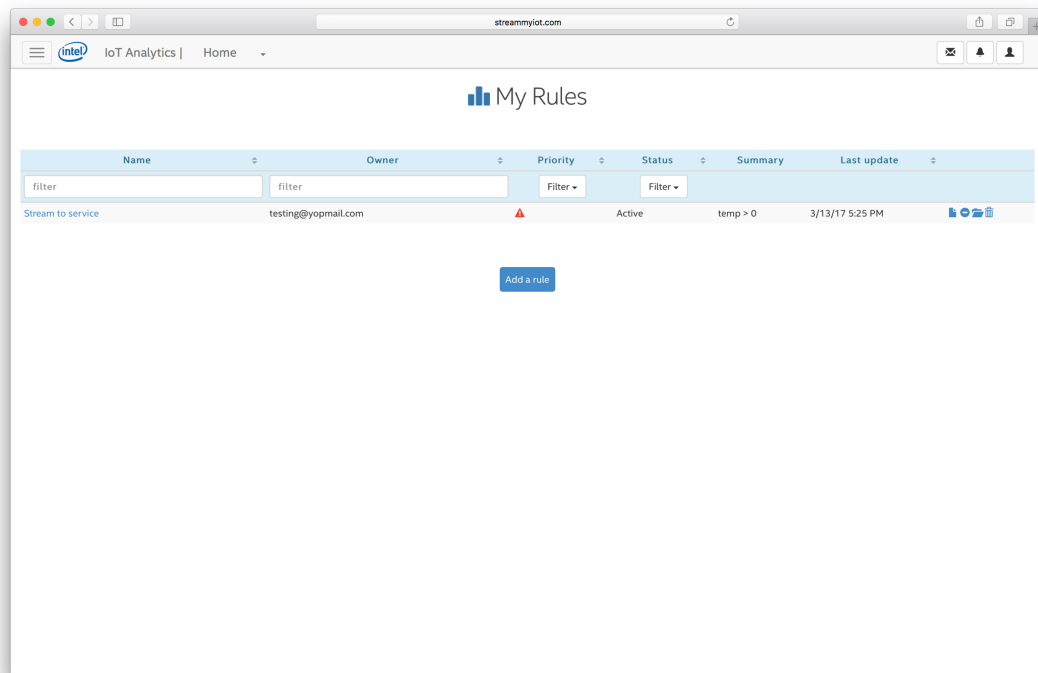
3. Now we have our telemetry reaching the cloud and on its way to our MQTT broker, we can consume messages (from the same topic) from the MQTT broker, process them, perform some compute (analytics, etc) and also perform an actuation. Actuation is where we instruct the Edison to do something an attached LED, motor, robotic arm, etc. Here is an example code of our analytics service written in JavaScript (design to run using NodeJS):

```javascript
var mqtt = require('mqtt')

var mqttHost = "mqtt://<hostname or IP address of MQTT broker>";
var mqttTopic = "myTopic";

var client  = mqtt.connect(mqttHost);

client.on('connect', function () {
        console.log("Connected")
        client.subscribe(mqttTopic)
})

client.on('message', function (topic, message) {
        var data = JSON.parse(message.toString())

        var value = parseInt(data.conditions[0].components[0].valuePoints[0].value, 10);

        console.log()
        console.log("incoming value: " + value)
})
```

This is a simple example of reading messages from our MQTT broker and we can extract the original value from our sensor (as well as a lot of other data) and this is stored in the 'value' variable. With this value, we can now perform any computation we wish. For the purpose of this tutorial, our analytics service will simply check if the value is even and turn off an LED attached to the Edison, and if it's odd, turn it off.

1. Now that we're receiving the telemetry data into our analytics service, we can perform any compute we wish; query databases, call third party APIs, calculate statistics, trigger an actuator attached to your Edison etc.
    This tutorial will calculate if the incoming value is even or odd and trigger an actuator by making an API call to streammyiot.com that will then stream this to your Edison. We've developed a simple wrapper in JavaScript for the API to make it easier to deal with the underlying HTTP RESTful APIs.

2. Actuators needs to be created on streammyiot.com and also registered on your Edison:
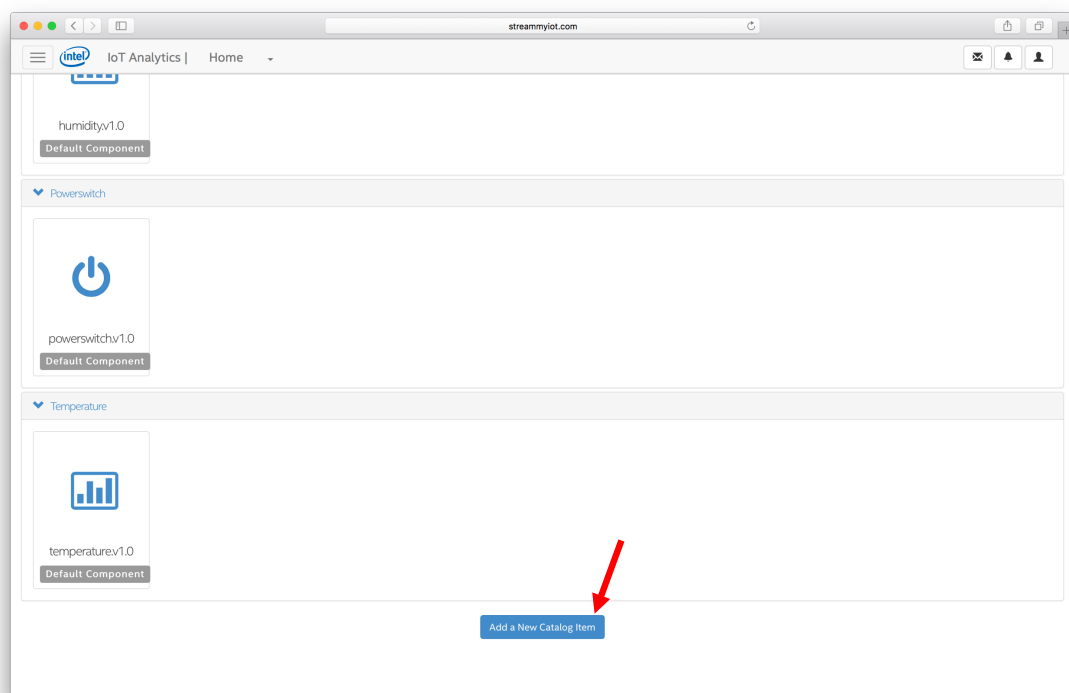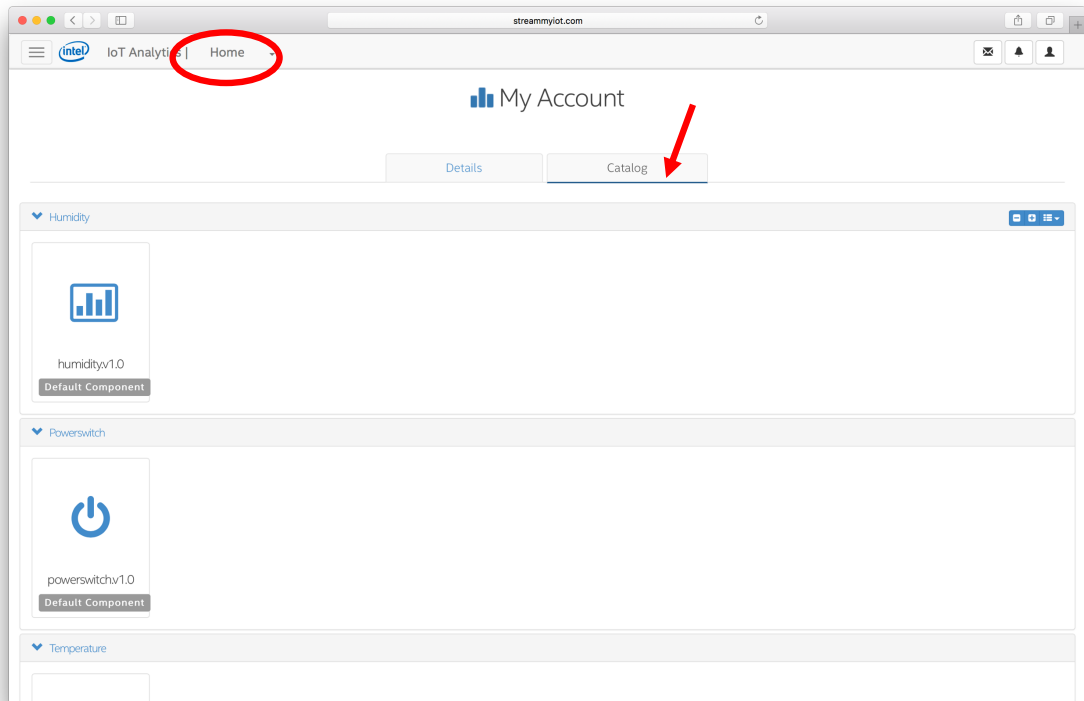
Below is an example of creating an 'alertled' component as an actuator. We specify the data type for an LED as Boolean (either on or off) and add a parameter called 'on' which can have the value '0' or '1'. In our analytics service we can then refer to this component and send a value for the parameter 'on' to your Edison.

3. Now we need to register this component with our Edison:

```
iotkit-admin register alertled alertled.v1.0
```

The response will be similar to:

```
2017-03-14T20:19:44.249Z - info: Starting registration ...
2017-03-14T20:19:44.252Z - info: Device has already been activated. Updating ...
2017-03-14T20:19:44.253Z - info: Updating metadata...
2017-03-14T20:19:44.261Z - info: Metadata updated.
Attributes sent
2017-03-14T20:19:44.497Z - info: Component registered name=alertled, type=alertled.v1.0,
cid=6af5508f-2666-4364-8981-7a4c2a4ef6bd, deviceId=78-4f-43-7b-4b-fe
```

You need to make a note of the component ID 'cid' from the response. Alternatively you can run the command line below to view all components connected to your Edison:

```
iotkit-admin components
```

The response will be similar to:

id : t	Name	cID
temperature.v1.0	temp	e3679d3e-ccd4-465c-9c65-cb9fe1d417e3
alertled.v1.0	alertled	6af5508f-2666-4364-8981-7a4c2a4ef6bd

4. Now we have our component ID for our 'alertled', we finally need our account ID as this will be used in our service:
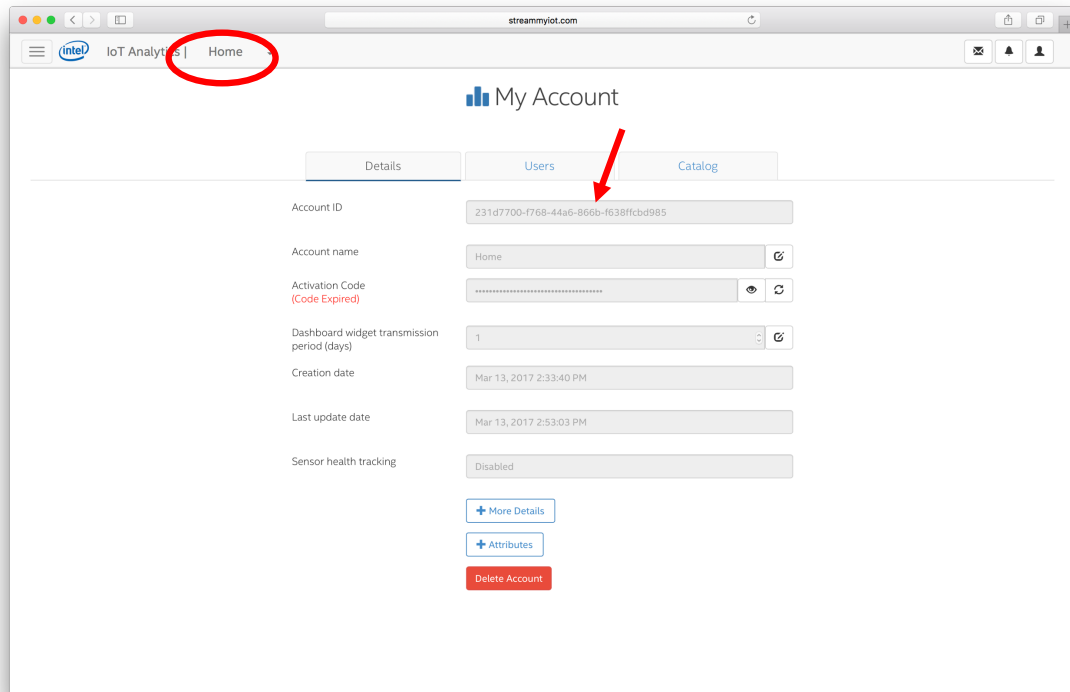
5. Putting everything together we get:

```
var mqtt = require('mqtt')
var api = require('../api')

var mqttHost = "mqtt://<hostname or IP address of MQTT broker>";
var mqttTopic = "myTopic";

var username = "<email address>";
var password = "<password>";

var accountId = "4d6ab4c1-57ae-469a-913b-b43c67981dc2";
var alertledComponentId = "e208baa6-7b7c-4ea6-90df-6c8ab5dc1fe7";

var ready = false;
var apiToken = null;
api.auth.getToken(username, password, function(err, response){
        if(err) throw err

        ready = true;
        apiToken = response.token
})

var client  = mqtt.connect(mqttHost);

client.on('connect', function () {
        console.log("Connected")
        client.subscribe(mqttTopic)
})

client.on('message', function (topic, message) {
        if(!ready) return;

        var data = JSON.parse(message.toString())

        var value = parseInt(data.conditions[0].components[0].valuePoints[0].value, 10);
```

```javascript
        console.log()
        console.log("incoming value: " + value)
        var alertledValue = (value % 2)+""

        var data = {
            "commands": [{
              "componentId": alertledComponentId,
              "parameters": [{
                    "name": "on",
                    "value": alertledValue
              }],
              "transport": "ws"
            }
         ],
         "complexCommands": []
        };
        api.control.sendActuation(accountId, data, apiToken, function(err, res){
            if(!err)
            {
                    console.log("Actuation sent")
                    console.log("component: " + "alertled")
                    console.log("parameter: " + "on")
                    console.log("value: " + alertledValue)
            }
            else
            {
                    console.log(err)
            }
        })
})
```

In order to interact with the API you must use tokens. You have a few options to get a token, but this example keeps it simple by generating a token every time the application starts using your credentials. This token is then fed into subsequent API calls (e.g. api.control.sendActuation)

1 – SIMPLE OUTLIER DETECTION

```
var mqtt = require('mqtt')
var outliers = require('outliers')
var api = require('../api')


var mqttHost = "mqtt://77.68.8.247";
var mqttTopic = "myTopic";


var username = "testing@yopmail.com";
var password = "aPassword1";


var accountId = "231d7700-f768-44a6-866b-f638ffcbd985";
var deviceId = "78-4f-43-7b-4b-fe";
var temperatureSensorComponentId = "e3679d3e-ccd4-465c-9c65-cb9fe1d417e3";


var historicalDataPoints = null;
const HISTORICAL_LIMIT = 10;


var ready = false;
var apiToken = null;
api.auth.getToken(username, password, function(err, response){
      if(err) throw err

      apiToken = response.token

      // Fetch the last hour of data points
      var oneHour = 60*60*1000;

      api.data.retrieveData(accountId, {
        "from": (new Date()).getTime() - oneHour,
        "to": (new Date()).getTime(),
        "targetFilter": {
          "deviceList": [
            deviceId
          ]
        },
        "metrics": [
          {
            "id": temperatureSensorComponentId
          }
        ]
      }, apiToken, function(err, response){
            historicalDataPoints = response.series[0].points.map(function(p){
                  return parseFloat(p.value);
            })

            if(historicalDataPoints.length > HISTORICAL_LIMIT)
            {
                  historicalDataPoints =
historicalDataPoints.slice(historicalDataPoints.length - HISTORICAL_LIMIT)
            }

            ready = true;
```

```javascript
        })
})

var client  = mqtt.connect(mqttHost);

client.on('connect', function () {
        console.log("Connected")
        client.subscribe(mqttTopic)
})

client.on('message', function (topic, message) {
        if(!ready) return;
        if(topic != mqttTopic) return;

        var data = JSON.parse(message.toString())

        var value = parseFloat(data.conditions[0].components[0].valuePoints[0].value);

        console.log()
        console.log("incoming value: " + value)

        historicalDataPoints.push(value)
        if(historicalDataPoints.length > HISTORICAL_LIMIT)
        {
                historicalDataPoints.shift();
        }
        console.log(historicalDataPoints)

        // Look for outliers
        var o = outliers(historicalDataPoints)
        if(o.length > 0)
        {
                // Process outliers:
                // Send an email/SMS?
                // Log the outliers event
                // And only when we see N events per hour
                // Turn on warning LED, shutdown machinery, etc
        }
})

client.on('error', function () {
        console.log(arguments)
})
```