X-CUBE-AWS

rev1.0 09/06/2020

GOAL

Modify X-CUBE-AWS example to develop a robotic arm

PREREQUISITES

Software needed:

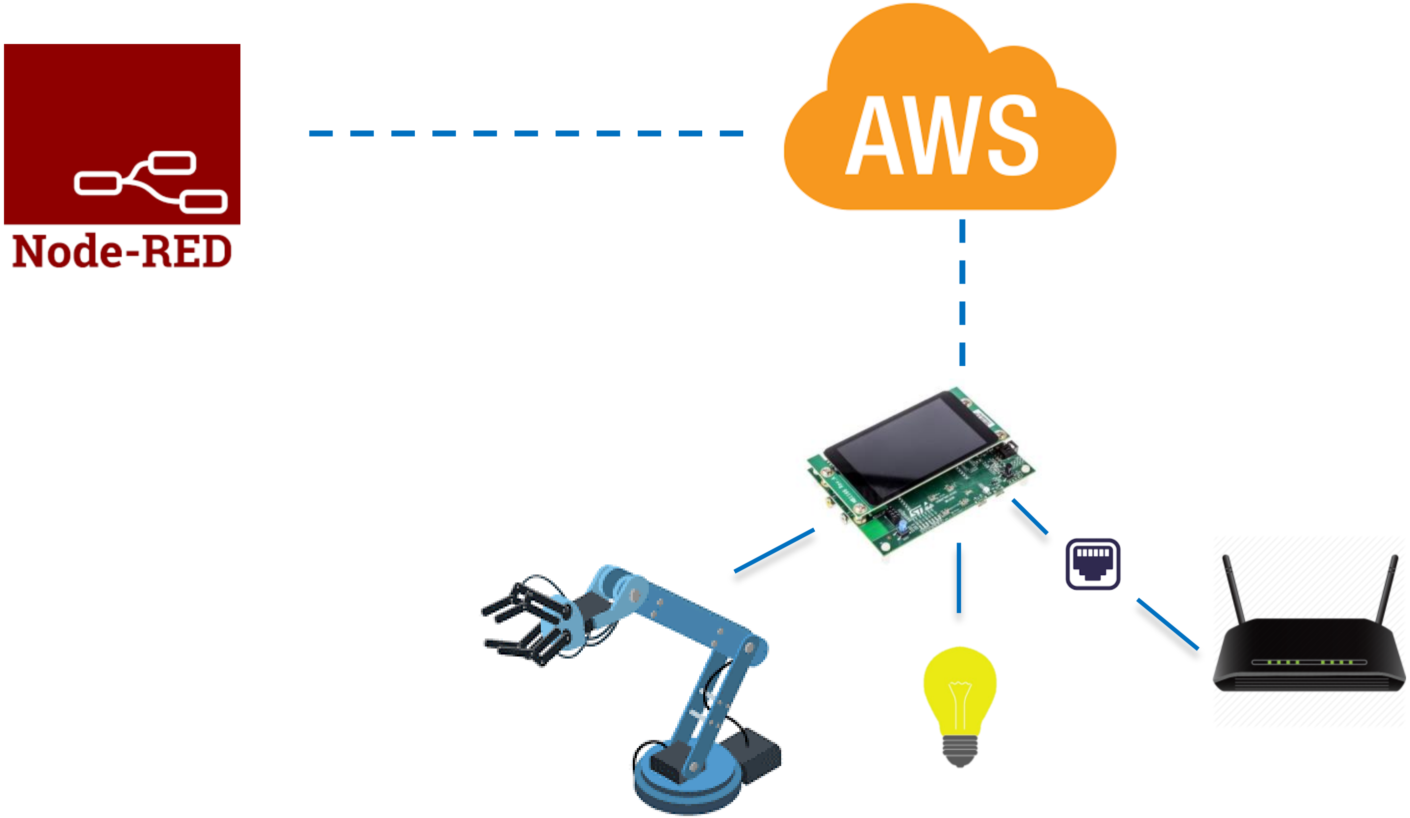
- STM32IDE
- X-CUBE-AWS
- TeraTerm

Hardware used in this example:

- 32F769IDISCOVERY

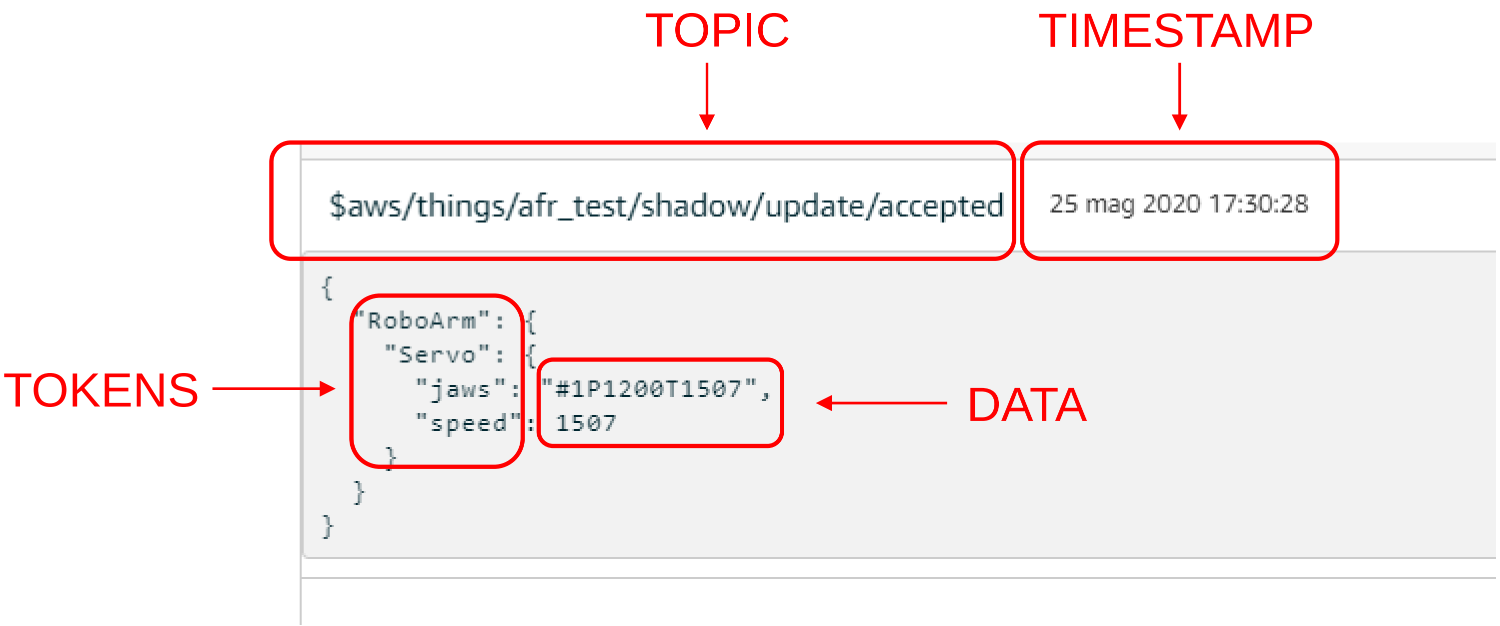
P.S. Some parts, like the communication with the robotic arm, can be unclear. Keep in mind that the main aim of these slides is to modify the given example from ST.

Project Setup

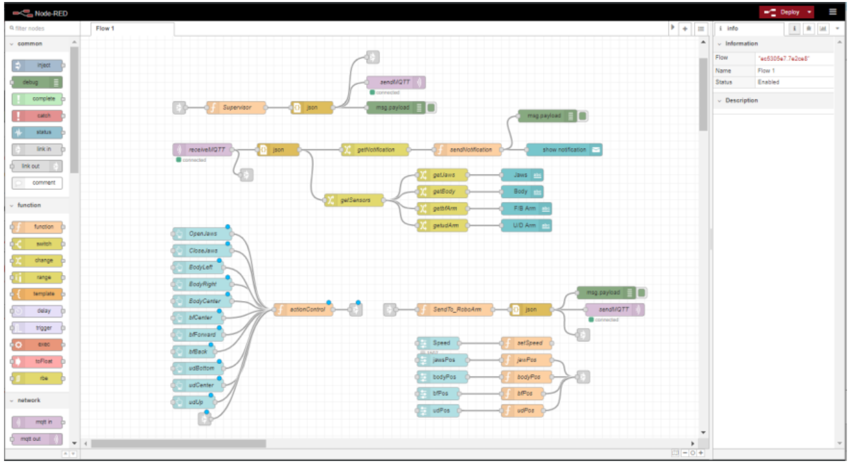


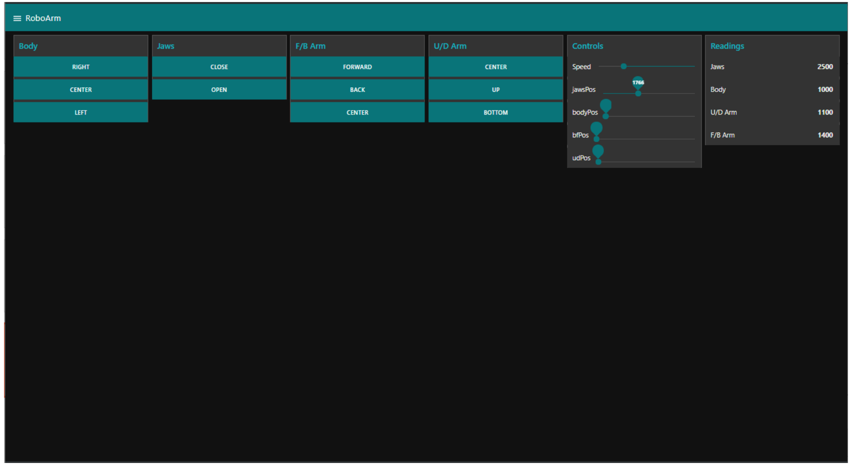
JSON Messages

Communication between devices takes place through the exchange of JSON messages to and from AWS.



Node-RED





Node-RED server contains all the algorithms (flows) to manage the infrastructure and its function is to exchange messages with AWS.

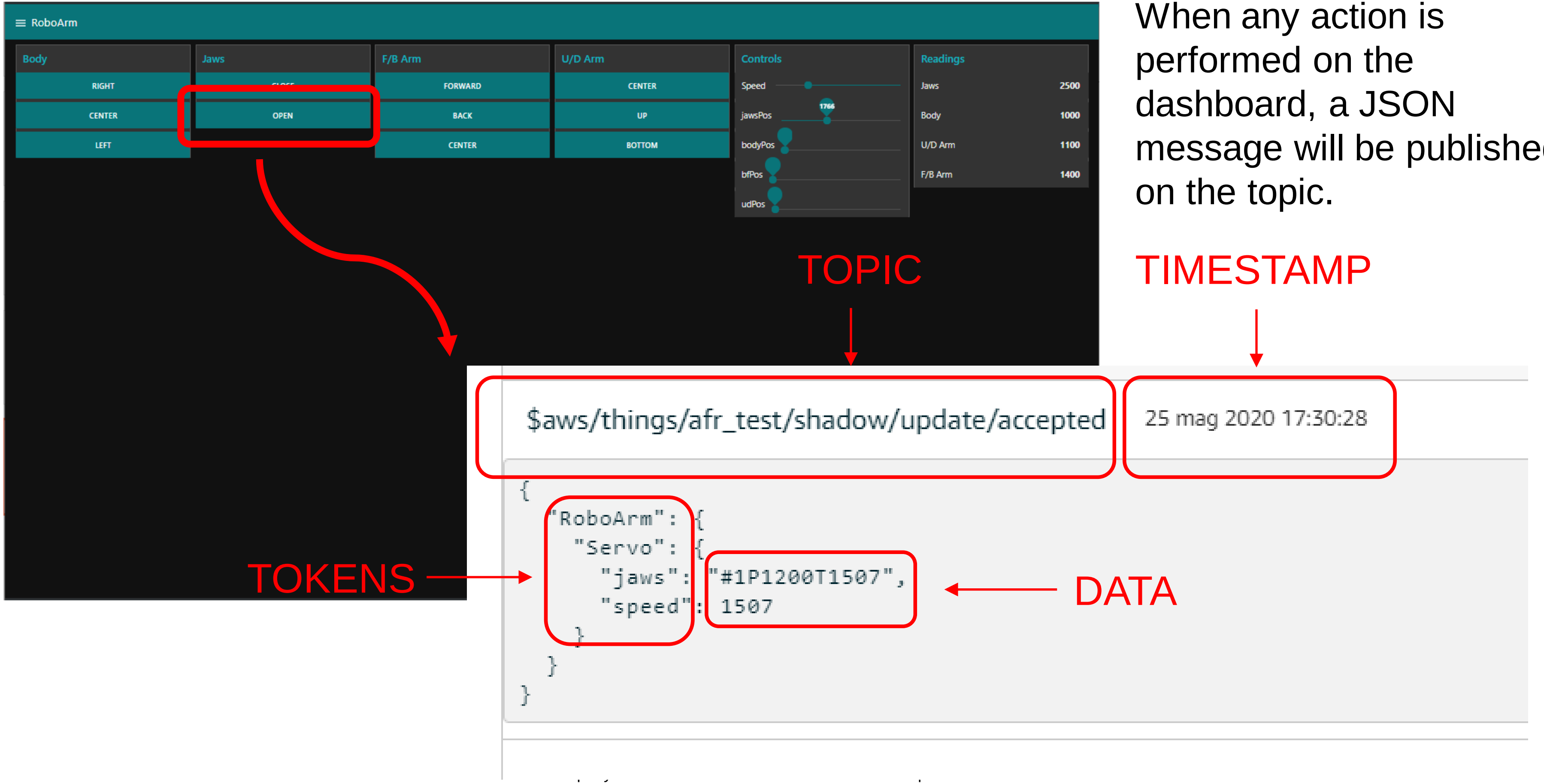
A dashboard running on Node-RED has been implemented as well.

Node-RED

AWS

Dashboard Example

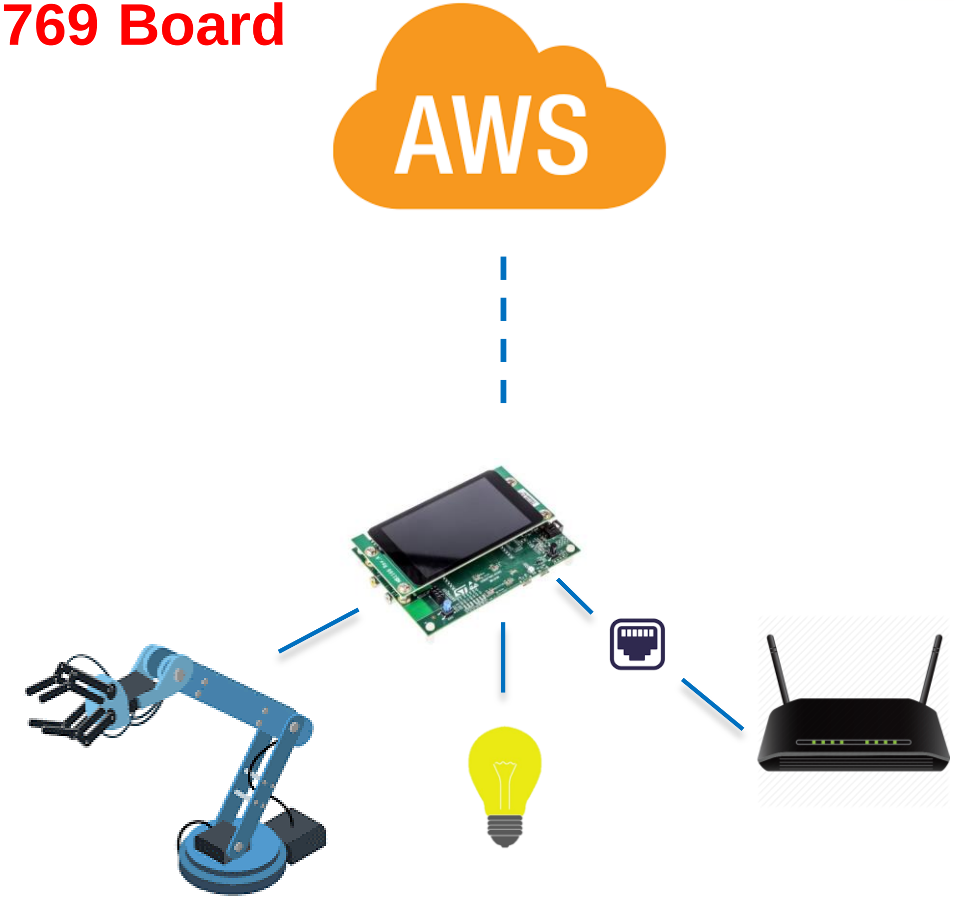
When any action is performed on the dashboard, a JSON message will be published on the topic.



STM32F769 Board

Features:

- **Internet Connection over Ethernet**
- **Controls a robotic Arm via UART**
- **Toggle on board LED via AWS**
- **Send notifications on Node-RED dashboard when an action has been performed.**
- **Console logging, for an easy troubleshooting and AWS Credential update.**



MainThread

For a real device, is usefull to send data at time intervals (like data acquired from sensors): a *timer* is needed for this task.

For more infos about timers and FreeRTOS have a look [here](#).

Sometimes creating other threads that can run indipendently is needed, for this pay attention to *red* boxes.

main.c aws_subscribe_publish_sensor_values.c

```c
 */

void getSensorData(void const * argument){

    for (;;){
        /*Read data from sensors*/
        voltage = rand()%100;
        current = rand()%100;
        temp    = rand()%100;
    }
}

/* TIMER Callback*/

static void TimerCallback1SecExpired(void const * argument) {
    if (!sendRequestFlag) {
        sendRequestFlag=1;
    }
}
```

main.c aws_subscribe_publish_sensor_values.c

```c
/**
  * @brief  Start Thread
  * @param  argument not used
  * @retval None
  */
static void MainThread(void const * argument)
{
  osThreadId CloudThreadId = NULL;
  osThreadId getDataThreadId = NULL;

  osTimerId Timer1sHandle;

  UNUSED(argument);
  msg_info("\r\n Starting Main Thread...\n");

  platform_init();

  /* Create Timer*/
  osTimerDef(Timer1sDef,TimerCallback1SecExpired );
  Timer1sHandle= osTimerCreate(&os_timer_def_Timer1sDef, osTimerPeriodic, NULL);

  /* Start application task */
  osTimerStart(Timer1sHandle, 1000);

  /* Beware that the OS stack frames are allocated from the OS heap. May incur RAM overwrite in case of overflow. */
  osThreadDef(CLOUDNAME, &cloud_run, osPriorityNormal, 0, configMINIMAL_STACK_SIZE + 0x1000);
  CloudThreadId = osThreadCreate (osThread(CLOUDNAME), NULL);

  osThreadDef(GETDATANAME, &getSensorData, osPriorityNormal, 0, configMINIMAL_STACK_SIZE + 0x1000);
  getDataThreadId = osThreadCreate (osThread(GETDATANAME), NULL);

  if (CloudThreadId == NULL || getDataThreadId==NULL)
  {
    Error_Handler();
  }

  for( ;; )
  {
    /* Delete the start Thread */
    osThreadTerminate(NULL);
  }
}
```

Cloud_run()

The main differences with the last example is in the aws_sub[..].c file.

One important feature of our project is to send feedback data to AWS. Using the flag setted by the timer, each timer interval, the payload containing all the data is published to the topic.

For cleanness, the payload is built inside the function :

iot_create_payload(cPayload);

All the definitions needed are inside the same file.

main.c   *aws_subscribe_publish_sensor_values.c

```c

        /* create desired message */
        if (!cPayload) {
            cPayload = malloc(AWS_IOT_MQTT_TX_BUF_LEN);
            if (!cPayload) {
                msg_error("Unable to allocate memory for the Payload\n");
            }
        }

        if (bp_pushed == BP_SINGLE_PUSH || sendRequestFlag ) {


          //printf("Sending Sensor Data to AWS.\n");
            iot_create_payload(cPayload);

            paramsQOS1.payload = cPayload;

            // paramsQOS1.payloadLen = strlen(cPayload) + 1;
            paramsQOS1.payloadLen = strlen(cPayload);

            do {
                rc = aws_iot_mqtt_publish(&client, cPTopicName,
                        strlen(cPTopicName), &paramsQOS1);
                /*
                if (rc == AWS_SUCCESS) {
                    printf("\nPublished to topic %s:", cPTopicName);
                    printf("%s\n", cPayload);
                }*/


            } while (MQTT_REQUEST_TIMEOUT_ERROR == rc);

            if (sendRequestFlag) sendRequestFlag=0; //reset timer flag
        }

```

Creating the Payload

The message sent to AWS is this:

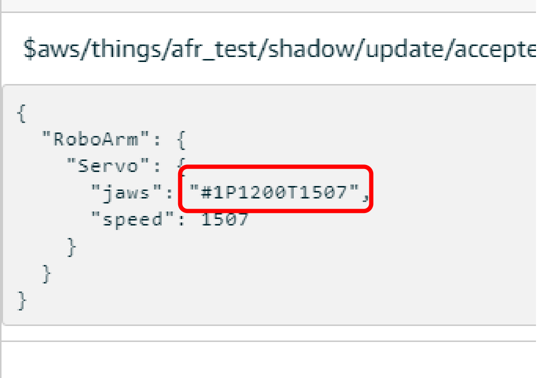
main.c   *aws_subscribe_publish_sensor_values.c

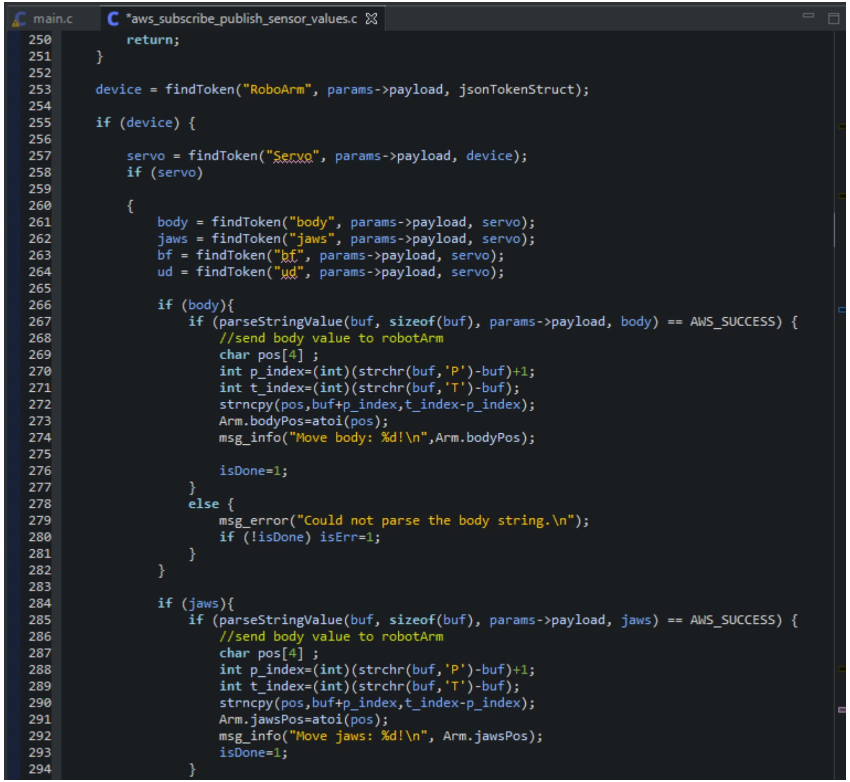
```c
static void iot_create_payload(char *Payload) {


    (void) snprintf(Payload, AWS_IOT_MQTT_TX_BUF_LEN, "%s", aws_json_device);
    (void) snprintf(Payload + strlen(Payload),
            AWS_IOT_MQTT_TX_BUF_LEN - strlen(Payload), "%s", aws_json_sensors);

    (void) snprintf(Payload + strlen(Payload),
            AWS_IOT_MQTT_TX_BUF_LEN - strlen(Payload), "%s%d%s",
            aws_json_jawsPos, Arm.jawsPos, aws_json_comma);
    (void) snprintf(Payload + strlen(Payload),
            AWS_IOT_MQTT_TX_BUF_LEN - strlen(Payload), "%s%d%s",
            aws_json_bodyPos, Arm.bodyPos, aws_json_comma);
    (void) snprintf(Payload + strlen(Payload),
                AWS_IOT_MQTT_TX_BUF_LEN - strlen(Payload), "%s%d%s",
                aws_json_bfPos, Arm.bfPos, aws_json_comma);
    (void) snprintf(Payload + strlen(Payload),
            AWS_IOT_MQTT_TX_BUF_LEN - strlen(Payload), "%s%d", aws_json_udPos,
            Arm.udPos);

    (void) snprintf(Payload + strlen(Payload),
            AWS_IOT_MQTT_TX_BUF_LEN - strlen(Payload), "%s", aws_json_endBrkt);
    (void) snprintf(Payload + strlen(Payload),
            AWS_IOT_MQTT_TX_BUF_LEN - strlen(Payload), "%s", aws_json_endBrkt);
    (void) snprintf(Payload + strlen(Payload),
            AWS_IOT_MQTT_TX_BUF_LEN - strlen(Payload), "%s", aws_json_endBrkt);
}
```

MQTTcallbackHandler

When any input is given by the dashboard, a message is published on the topic in wich the board is subscribed, so the handler will trigger.

At this point, the device parsed the message and split each token using the *jsmn parser* in order to get the string that have to been sent to the servo controller.





MQTTcallbackHandler

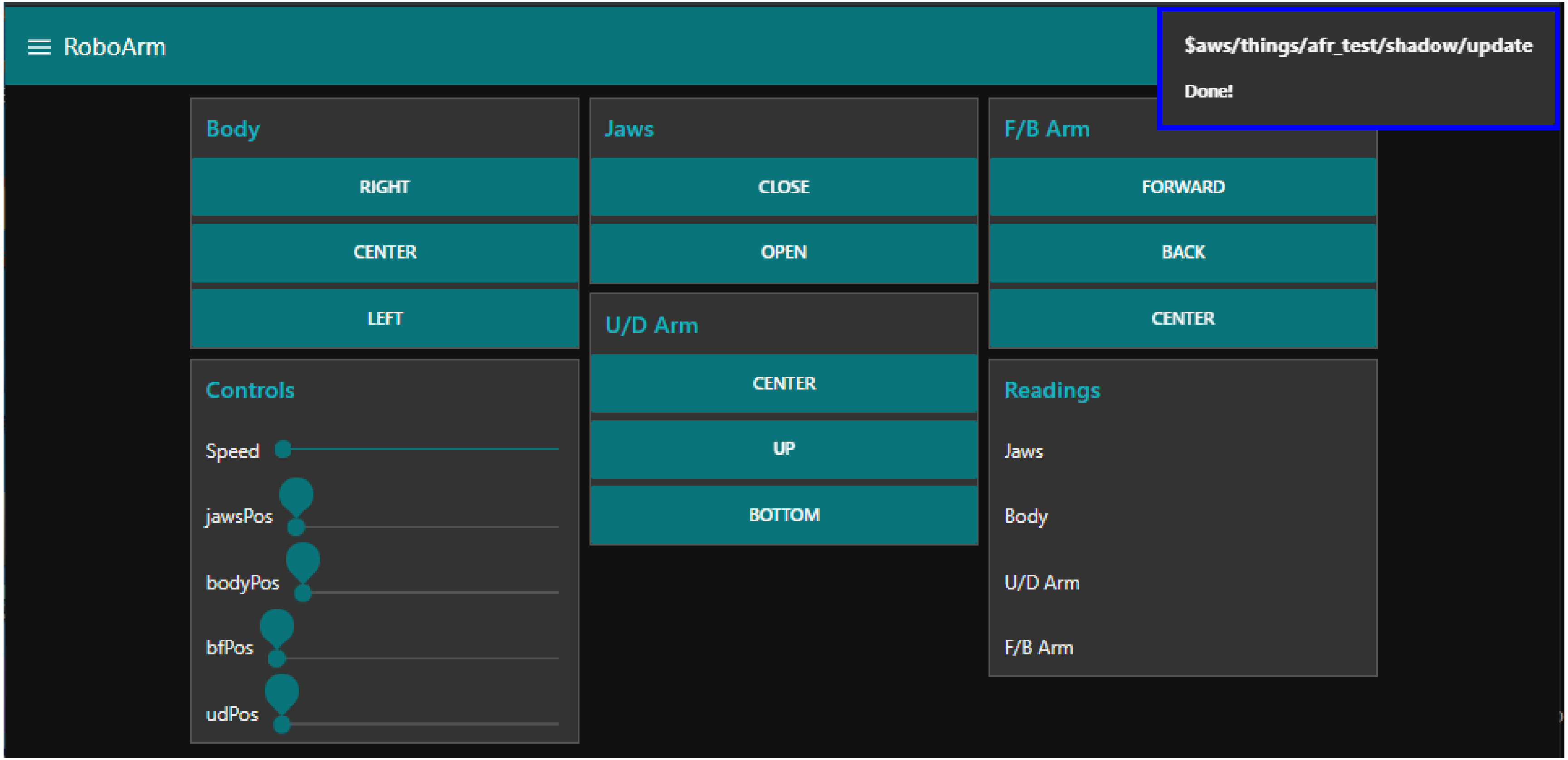
When the parsing is ended, the position or the led status is stored in ***buf*** .

If buf contains the position of the motors, it will be sent via UART to the motor controller.

At the end, an update on the operation status is published on the topic: doing this we will have live feedback of the operations done by the board, displaying them on the dashboard as notifications.

main.c   *aws_subscribe_publish_sensor_values.c

```c
        }

    }

    /* Set and report the operation state to the MQTT broker. */
    if (isDone || isErr || (ledstateOn != ledstateOn_last) ) {

        if(ledstateOn != ledstateOn_last){
            ledstateOn_last=ledstateOn;
            if (ledstateOn) {
                msg=msg_on; /*turn on led*/
            } else {
                msg=msg_off;/*turn off led*/
            }

            msg_info("LED %s!\n", buf);
            Led_SetState(ledstateOn);
        }


        if (isDone){
            msg= msg_done;
            msg_info("Position Sent %s!\n", buf);
            /*send position to uart*/
            strcat(buf, "\r\n");
            roboArm_UART_send((uint8_t*)&buf);
        }

        if (isErr) msg= msg_err;
        /*reset Flags*/
        isDone = 0;
        isErr = 0;

        sendParams.payloadLen = strlen(msg);
        sendParams.payload = (void*) msg;

        IoT_Error_t rc = aws_iot_mqtt_publish(pClient, cPTopicName,
                strlen(cPTopicName), &sendParams);

        if (rc == AWS_SUCCESS) {
            msg_info("\nPublished the new operation status to topic %s:", cPTopicName);

            msg_info("%s\n", msg);
        }
    }
}
```

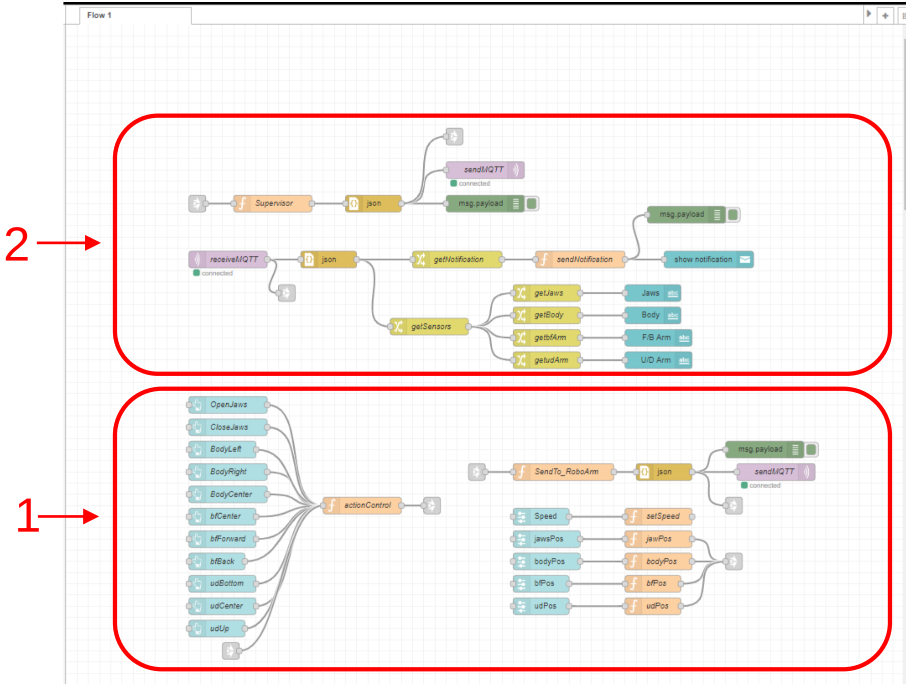

RoboArm
$aws/things/afr_test/shadow/update
Done!
Body
RIGHT
CENTER
LEFT
Jaws
CLOSE
OPEN
F/B Arm
FORWARD
BACK
CENTER
U/D Arm
CENTER
UP
BOTTOM
Controls
Speed
jawsPos
bodyPos
bfPos
udPos
Readings
Jaws
Body
U/D Arm
F/B Arm

Node-Red

This is the structure of the flow used in this example: let’s analyze one section at the time.

The flow is mainly split in two parts:

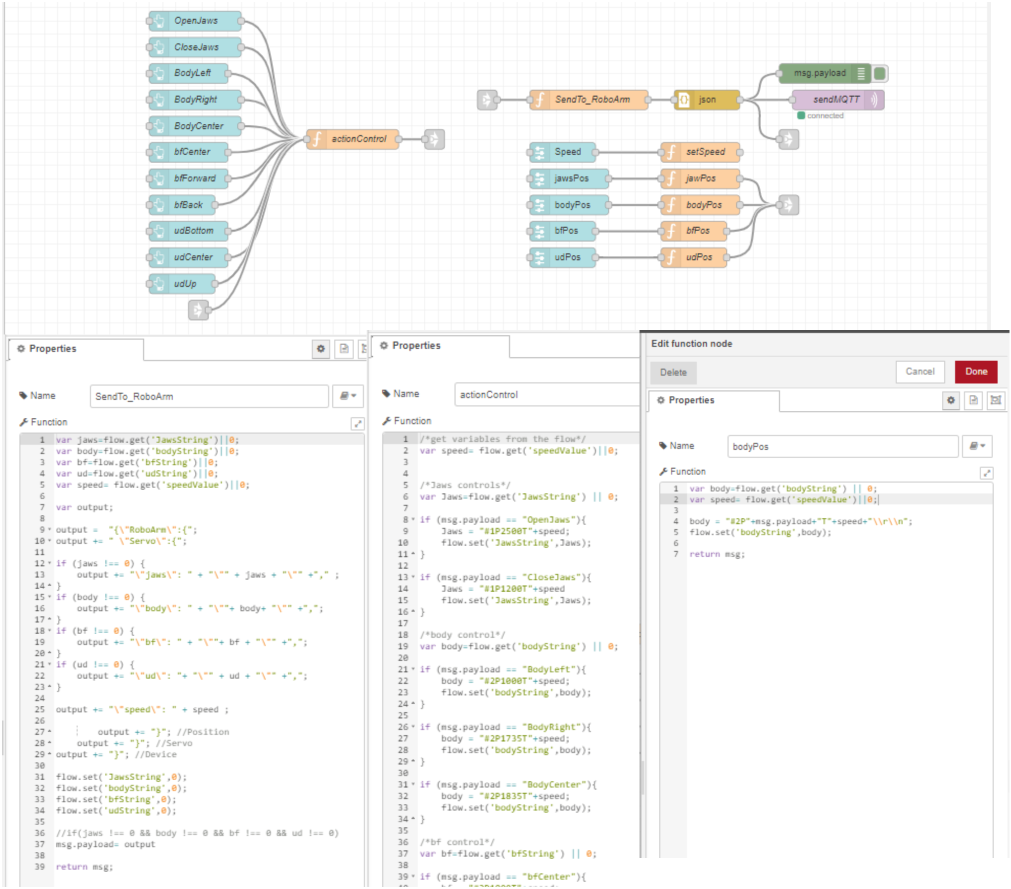
1. Get the inputs from the dashboard and publish them on the topic
2. Get the feedback data sent by the board and display them on the dashboard



Dashboard Inputs

The nodes used in this section are the following:

1. *Buttons* to send a fixed position
2. *Sliders* to modify value and motor positions
3. *Functions* to get, set variables and creating JSON messages
4. JSON parser to convert string in JSON payloads
5. *Link nodes* to connect button inputs and sliders value to the *SendTo_RoboArm* function
6. *Debug* node
7. *SendMQTT* node to publish on the topic



Get and set Variables in Node-RED

Variables are a little different in node-RED:

There are no specific types and in order to store variables and read the last value set special functions need to be called.

To get the value of a variable:

context.get(‘varName’)

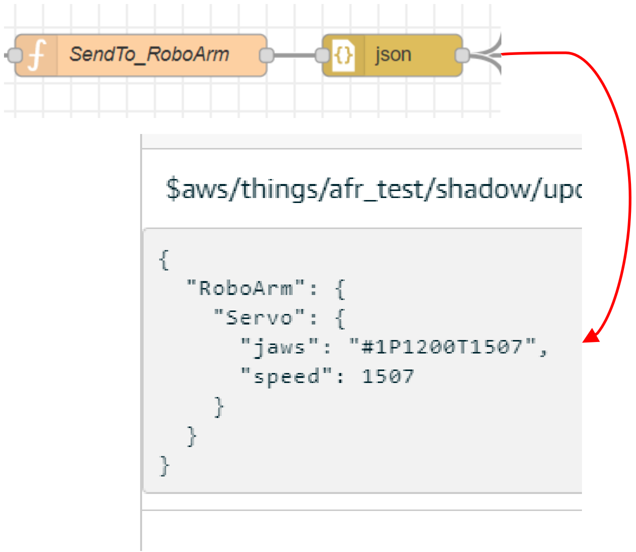
And to set the value of a variable:

context.set(‘varName’ , value)

To find more information to manage variables in Node-RED see here.

JSON Parser

Using the JSON Parser is possible to convert the msg object in a valid JSON Payload without writing any code.



Properties

Name: SendTo_RoboArm

Function

```
var jaws=flow.get('JawsString')||0;
var body=flow.get('bodyString')||0;
var bf=flow.get('bfString')||0;
var ud=flow.get('udString')||0;
var speed= flow.get('speedValue')||0;

var output;

output =  "{\"RoboArm\":{";
output += " \"Servo\":{";

if (jaws !== 0) {
    output += "\"jaws\": " + "\"" + jaws + "\"" +"," ;
}
if (body !== 0) {
    output += "\"body\": " + "\""+ body+ "\"" +",";
}
if (bf !== 0) {
    output += "\"bf\": " + "\""+ bf + "\"" +",";
}
if (ud !== 0) {
    output += "\"ud\": "+ "\"" + ud + "\"" +",";
}

output += "\"speed\": " + speed ;

        output += "}"; //Position
    output += "}"; //Servo
output += "}"; //Device

flow.set('JawsString',0);
flow.set('bodyString',0);
flow.set('bfString',0);
flow.set('udString',0);

//if(jaws !== 0 && body !== 0 && bf !== 0 && ud !== 0)
msg.payload= output

return msg;
```

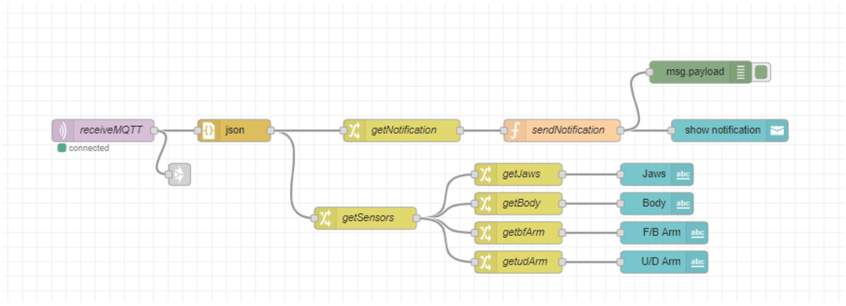
Get sensors values and notifications

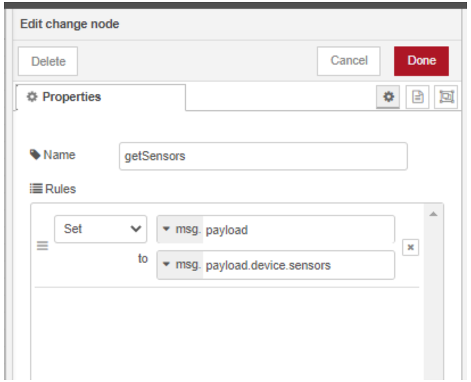
When a message is received it’s converted in json format.

Getting the different tokens is really easy: they are now stored in the payload of the msg object, coming from the receiving block, and they are accessible by typing

msg.payload.token.subtoken

For doing this is better to use a ***change node*** so if the message is ***undefined*** (for example we send the position of only one motor an the message does not contain all the motors positions the payload will be undefined) nothing will be triggered.

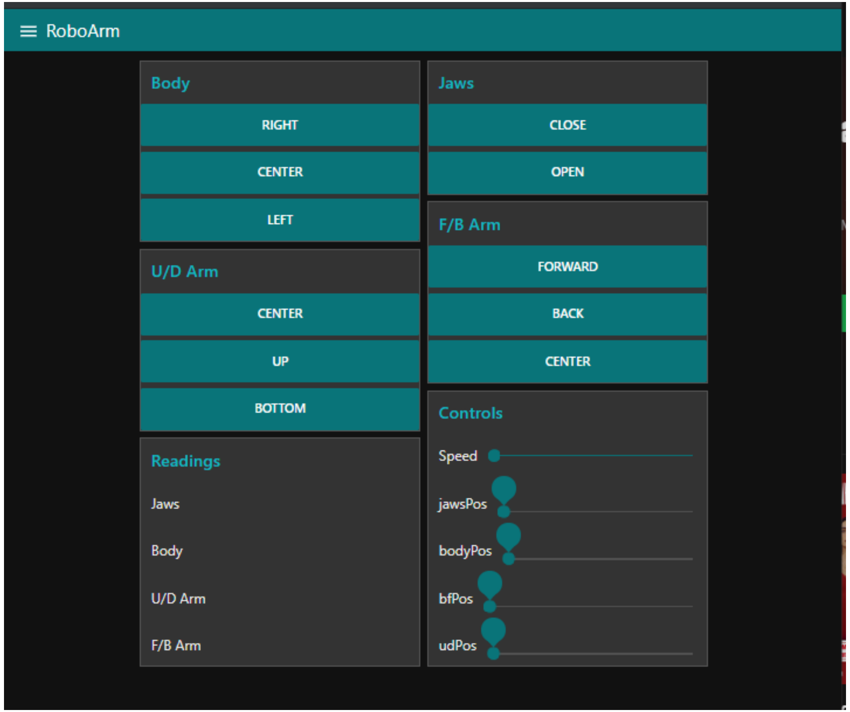




Dashboard

The Dashboard is the following one with this features:

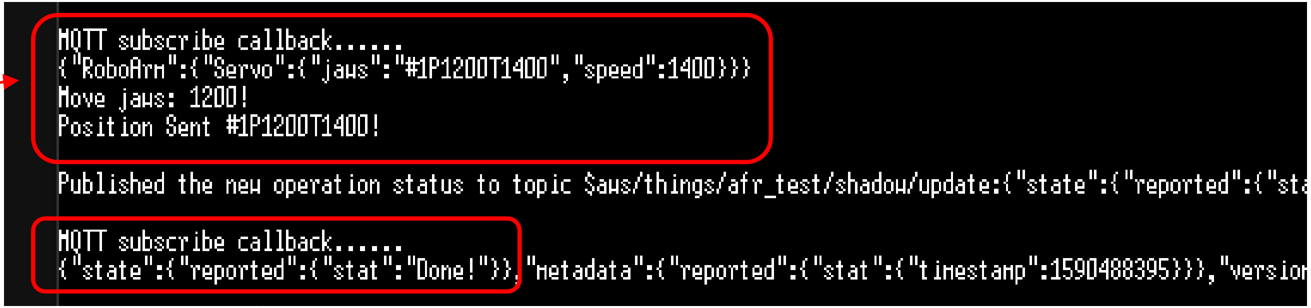
- On a button press a message with a fix position is published on the topic
- Moving *Position sliders,* we send a message with the motor position equal to the slider value
- *Readings* are the feedback values read from the board
- When an action is performed by the board a notification will appear



Callback Example

In this example we pushed the button on the dashboard to open the Jaws:

The board is capable of interpreting the message published on the topic, containing the position and the speed of the motors and after the action has been performed a notification of «Done» is published to the *update* topic resulting in a notification on the dashboard.



Motor Position

Status Notification

Access nodered from outside the network

When accessing nodered the address to reach it is

http://localhost:1880

Localhost is the equivalent to the IP address of the machine running Nodered.

```
Scheda LAN wireless Wi-Fi:

   Suffisso DNS specifico per connessione:
   Indirizzo IPv6 locale rispetto al collegamento . : fe80::d5ab:fad3:6f88:95af%15
   Indirizzo IPv4. . . . . . . . . . . . : 192.168.1.110
   Subnet mask . . . . . . . . . . . . . : 255.255.255.0
   Gateway predefinito . . . . . . . . . : 192.168.1.1
```

In our case, the equivalent is

http://192.168.1.110:1880/

This address is reachable only by the *local network.* (To know your local ip address, use *ipconfig* command on the cmd)

So, how can i reach the computer running nodered from outside the local network ?

The answer is simple, just enable the remote access on your router and perform a port forwarding.

This process is different for each modem, so check the manufacturer guide.

The basic idea is to reach the modem's ip address from the outside network, which will automatically redirect our request to the local ip address of the pc running Node-RED.

An example of port forwarding for a TP-LINK modem here.

Securing Nodered

Now we can access nodered (and the dashboard) from anywhere in the world.

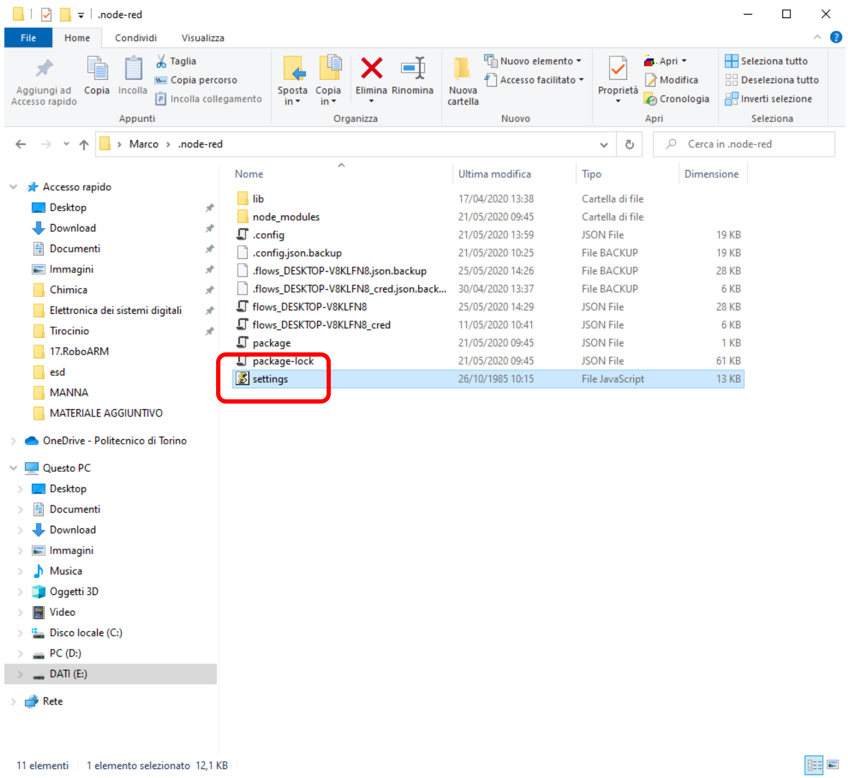
There is still a big problem, ***security***.

It's possible to protect nodered canvas and dashboard with some passwords.

Go to nodered folder

C:\Users\UserName\.node-red

And open the ***settings*** file (use notepad or any equivalent editor).



Securing Nodered

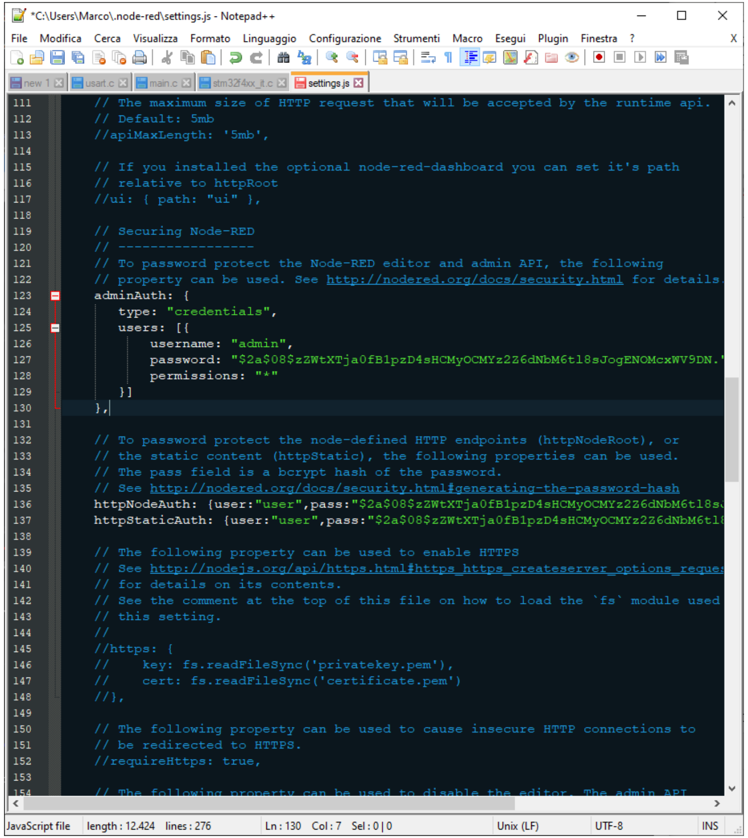
Find the Securing Node-RED section and uncomment the lines as in the picture.

The password is encripted and to set a new one, we need to create a new ***hash.***

Open the terminal and install node-red-admin by typing the line

> ***npm install –g node-red-admin***

it will take a while.



Create an hash password

In the same window type the line

node-red-admin hash-pw

Enter the password and press enter (the password will be transparent in the window).

Copy the generated hash in pass field of the settings file.

As you can see, there are multiple users:

- ***AdminAuth***: secures the canvas, it's possible to add some users with only reading permissions (here)
- ***httpAuth***: secures the dashboard