



X-CUBE-AWS

rev1.0 09/06/2020

GOAL

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

PREREQUISITES

Software needed:

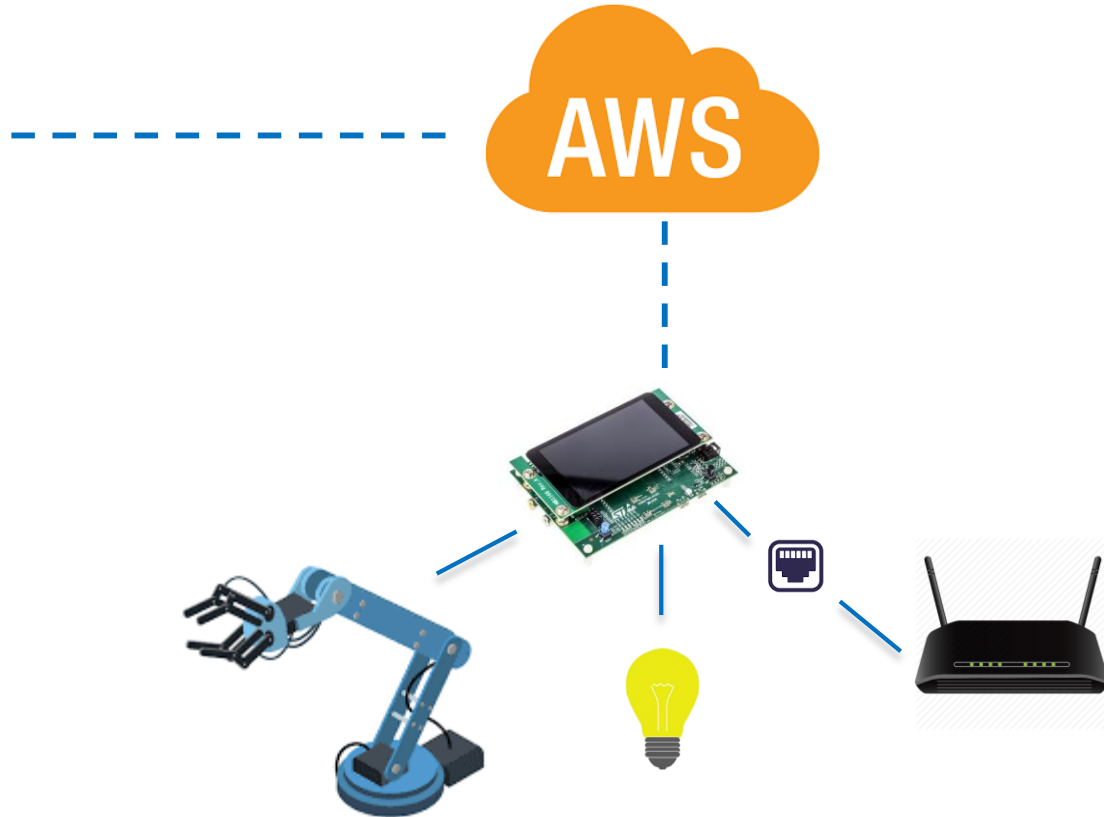
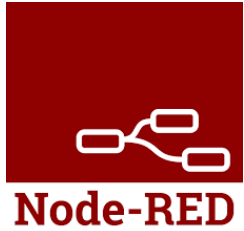
- STM32IDE
- X-CUBE-AWS
- TeraTerm

Hardware used in this example:

- 32F769IDISCOVERY

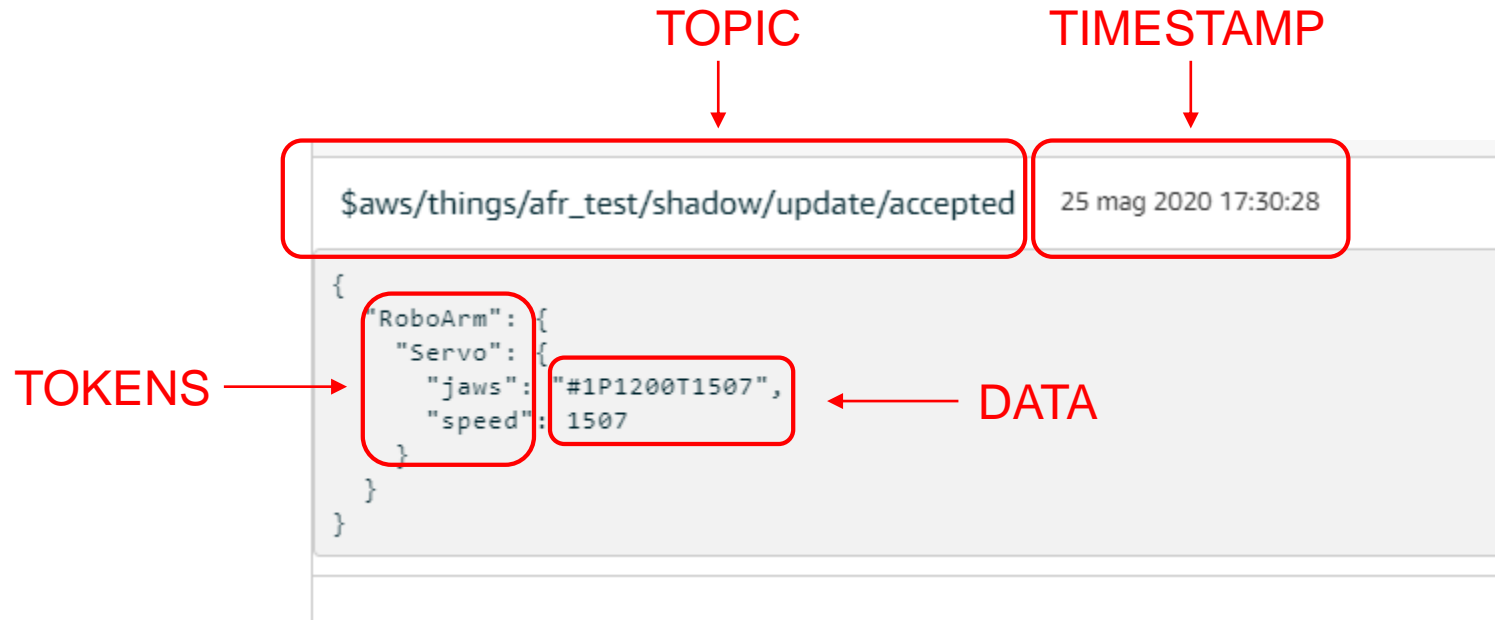
P.S. Some parts, like the communication with the robotic arm, can be unclear. Keep in mind that the main aim of these slides is to modify the given example from ST.

Project Setup

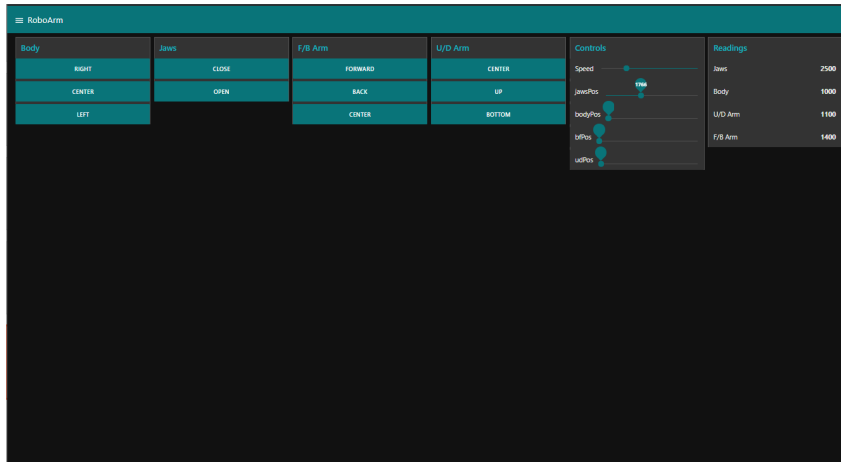
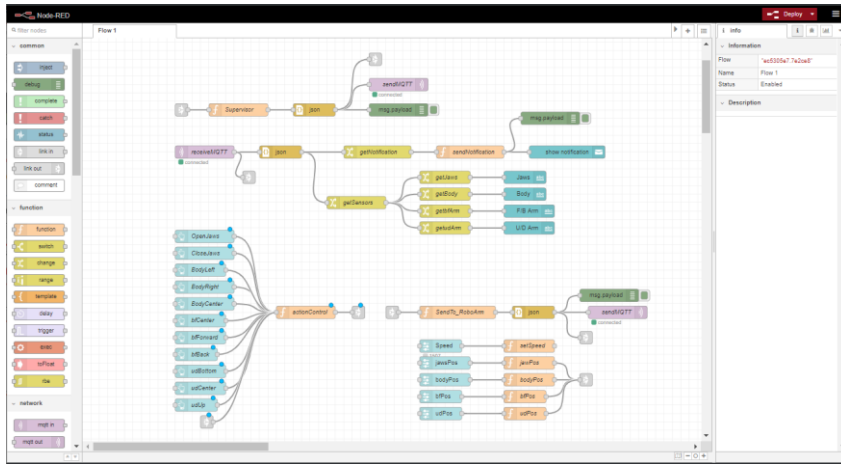


JSON Messages

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

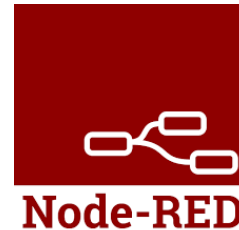


Node-RED



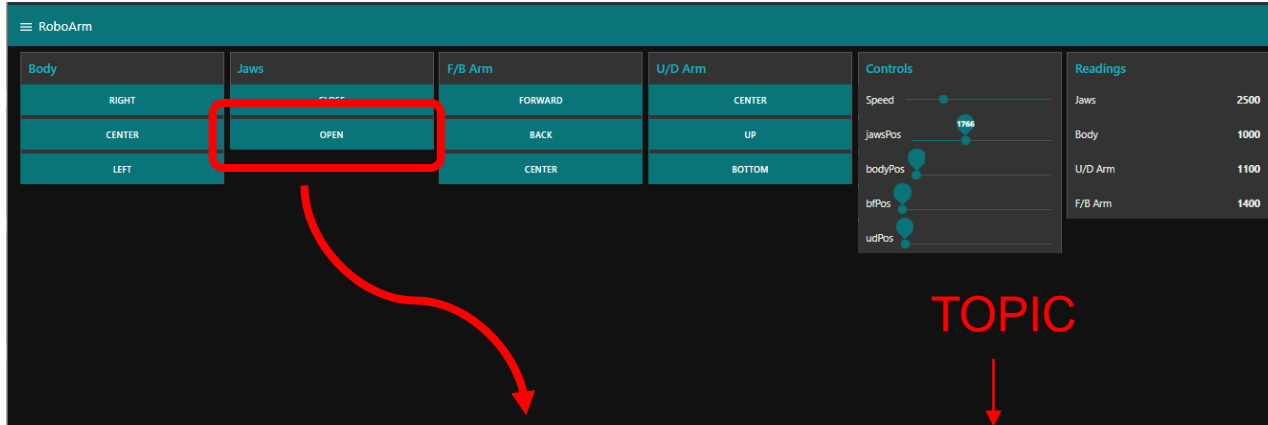
Node-RED server contains all the algorithms (flows) to manage the infrastructure and its function is to exchange messages with AWS.

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



Dashboard Example

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



TOPIC

TIMESTAMP

`$aws/things/afr_test/shadow/update/accepted`

`25 mag 2020 17:30:28`

TOKENS

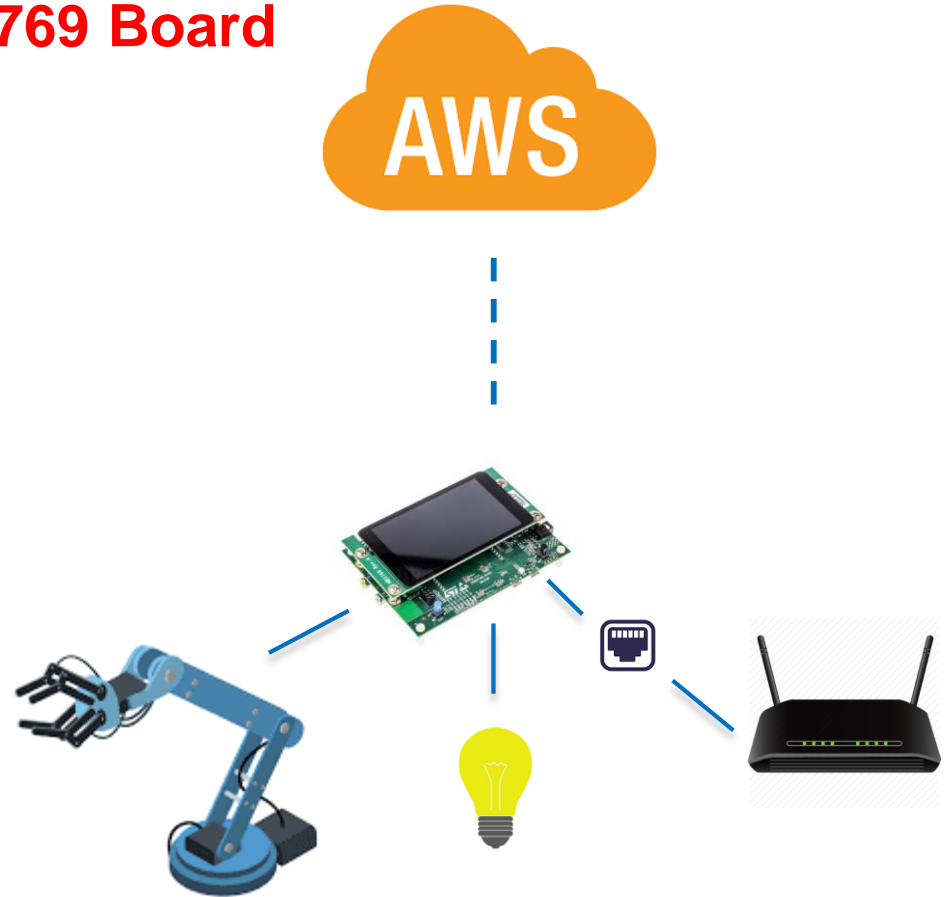
```
{
  "RoboArm": {
    "Servo": {
      "jaws": "#1P1200T1507",
      "speed": 1507
    }
  }
}
```

DATA

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
5400 /*
5401  * @brief Start Thread
5402  * @param argument not used
5403  * @return None
5404  */
5405 static void MainThread(void const * argument)
5406 {
5407     osThreadId CloudThreadId = NULL;
5408     osThreadId getDataThreadId = NULL;
5409     osTimerId TimerIsHandle;
5410
5411     UNUSED(argument);
5412     msg_info("\r\n Starting Main Thread ..\n");
5413
5414     platform_init();
5415
5416     /* Create Timer */
5417     osTimerDef(TimerIsDef, TimerCallback1SecExpired);
5418     TimerIsHandle = osTimerCreate(&os_timer_def_TimerIsDef, osTimerPeriodic, NULL);
5419
5420     /* Start application task */
5421     osTimerStart(TimerIsHandle, 1000);
5422
5423     /* Beware that the OS stack frames are allocated from the OS heap. May incur RAM overwrite in case of overflow. */
5424     osThreadDef(CLOUDNAME, &Cloud_run, osPriorityNormal, 0, configMINIMAL_STACK_SIZE + 0x1000);
5425     CloudThreadId = osThreadCreate(osThread(CLOUDNAME), NULL);
5426
5427     osThreadDef(GETDATANAME, &getSensorData, osPriorityNormal, 0, configMINIMAL_STACK_SIZE + 0x1000);
5428     getDataThreadId = osThreadCreate(osThread(GETDATANAME), NULL);
5429
5430     if (CloudThreadId == NULL || getDataThreadId == NULL)
5431     {
5432         Error_Handler();
5433     }
5434
5435     for(;;)
5436     {
5437         /* Delete the start Thread */
5438         osThreadTerminate(NULL);
5439     }
5440 }

aws_subscribe_publish_sensor_values.c
527 /*
528  */
529 void getSensorData(void const * argument)
530 {
531     for(;;){
532         /*Read data from sensors*/
533         voltage = rand()%100;
534         current = rand()%100;
535         temp = rand()%100;
536     }
537 }
538
539 /* TIMER Callback */
540 static void TimerCallback1SecExpired(void const * argument) {
541     if (!sendRequestFlag) {
542         sendRequestFlag=1;
543     }
544 }
545
```

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
951
952 /* create desired message */
953 if (!cPayload) {
954     cPayload = malloc(AWS_IOT_MQTT_TX_BUF_LEN);
955     if (!cPayload) {
956         msg_error("Unable to allocate memory for the Payload\n");
957     }
958 }
959
960 if (bp_pushed == BP_SINGLE_PUSH || sendRequestFlag) {
961
962     //printf("Sending Sensor Data to AWS.\n");
963     iot_create_payload(cPayload);
964
965     paramsQOS1.payload = cPayload;
966
967     // paramsQOS1.payloadLen = strlen(cPayload) + 1;
968     paramsQOS1.payloadLen = strlen(cPayload);
969
970     do {
971         rc = aws_iot_mqtt_publish(&client, cPTopicName,
972                                   strlen(cPTopicName), &paramsQOS1);
973
974         /*
975         if (rc == AWS_SUCCESS) {
976             printf("\nPublished to topic %s:", cPTopicName);
977             printf("%s\n", cPayload);
978         }*/
979
980     } while (MQTT_REQUEST_TIMEOUT_ERROR == rc);
981
982     if (sendRequestFlag) sendRequestFlag=0; //reset timer flag
983 }
984
985
```

Creating the Payload

The message sent to AWS is this:

[\\$aws/things/afr_test/shadow/update](#) [Esporta](#) [Annulla](#) [Sospendi](#)

Pubblicare

È possibile specificare un argomento e un messaggio da pubblicare con livello QoS pari a 0.

[Pubblic...](#)

```
1 {
2   "message": "Hello from AWS IoT console"
3 }
```

[\\$aws/things/afr_test/shadow/...](#) 09 giu 2020 12:00:49 [Espo...](#) [Nasco...](#)

```
{
  "device": {
    "sensors": {
      "jaws": 1200,
      "body": 1735,
      "bf": 1800,
      "ud": 2000
    }
  }
}
```

```
main.c  *aws_subscribe_publish_sensor_values.c
401
402 static void iot_create_payload(char *Payload) {
403
404
405     (void) snprintf(Payload, AWS_IOT_MQTT_TX_BUF_LEN, "%s", aws_json_device);
406     (void) snprintf(Payload + strlen(Payload),
407                     AWS_IOT_MQTT_TX_BUF_LEN - strlen(Payload), "%s", aws_json_sensors);
408
409     (void) snprintf(Payload + strlen(Payload),
410                     AWS_IOT_MQTT_TX_BUF_LEN - strlen(Payload), "%s%d%s",
411                     aws_json_jawsPos, Arm.jawsPos, aws_json_comma);
412     (void) snprintf(Payload + strlen(Payload),
413                     AWS_IOT_MQTT_TX_BUF_LEN - strlen(Payload), "%s%d%s",
414                     aws_json_bodyPos, Arm.bodyPos, aws_json_comma);
415     (void) snprintf(Payload + strlen(Payload),
416                     AWS_IOT_MQTT_TX_BUF_LEN - strlen(Payload), "%s%d%s",
417                     aws_json_bfPos, Arm.bfPos, aws_json_comma);
418     (void) snprintf(Payload + strlen(Payload),
419                     AWS_IOT_MQTT_TX_BUF_LEN - strlen(Payload), "%s%d", aws_json_udPos,
420                     Arm.udPos);
421
422     (void) snprintf(Payload + strlen(Payload),
423                     AWS_IOT_MQTT_TX_BUF_LEN - strlen(Payload), "%s", aws_json_endBrkt);
424     (void) snprintf(Payload + strlen(Payload),
425                     AWS_IOT_MQTT_TX_BUF_LEN - strlen(Payload), "%s", aws_json_endBrkt);
426     (void) snprintf(Payload + strlen(Payload),
427                     AWS_IOT_MQTT_TX_BUF_LEN - strlen(Payload), "%s", aws_json_endBrkt);
428 }
429
```

MQTTcallbackHandler

When any input is given by the dashboard, a message is published on the topic in which 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 be sent to the servo controller.

```
$aws/things/afr_test/shadow/update/accepte  
  
{  
  "RoboArm": {  
    "Servo": {  
      "jaws": "#1P1200T1507",  
      "speed": 1507  
    }  
  }  
}
```

```
main.c  aws_subscribe_publish_sensor_values.c  
250  
251     return;  
252  
253     device = findToken("RoboArm", params->payload, jsonTokenStruct);  
254  
255     if (device) {  
256  
257         servo = findToken("Servo", params->payload, device);  
258         if (servo)  
259         {  
260  
261             body = findToken("body", params->payload, servo);  
262             jaws = findToken("jaws", params->payload, servo);  
263             bf = findToken("bf", params->payload, servo);  
264             ud = findToken("ud", params->payload, servo);  
265  
266             if (body){  
267                 if (parseStringValue(buf, sizeof(buf), params->payload, body) == AWS_SUCCESS) {  
268                     //send body value to robotArm  
269                     char pos[4];  
270                     int p_index=(int)(strchr(buf,'P')-buf)+1;  
271                     int t_index=(int)(strchr(buf,'T')-buf);  
272                     strncpy(pos,buf+p_index,t_index-p_index);  
273                     Arm.bodyPos=atoi(pos);  
274                     msg_info("Move body: %d!\n",Arm.bodyPos);  
275  
276                     isDone=1;  
277                 }  
278                 else {  
279                     msg_error("Could not parse the body string.\n");  
280                     if (!isDone) isErr=1;  
281                 }  
282             }  
283  
284             if (jaws){  
285                 if (parseStringValue(buf, sizeof(buf), params->payload, jaws) == AWS_SUCCESS) {  
286                     //send body value to robotArm  
287                     char pos[4];  
288                     int p_index=(int)(strchr(buf,'P')-buf)+1;  
289                     int t_index=(int)(strchr(buf,'T')-buf);  
290                     strncpy(pos,buf+p_index,t_index-p_index);  
291                     Arm.jawsPos=atoi(pos);  
292                     msg_info("Move jaws: %d!\n", Arm.jawsPos);  
293                     isDone=1;  
294                 }  
295             }  
296         }  
297     }  
298 }
```

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
354     }
355
356     }
357
358     /* Set and report the operation state to the MQTT broker. */
359     if (isDone || isErr || (ledstateOn != ledstateOn_last) ) {
360
361         if(ledstateOn != ledstateOn_last){
362             ledstateOn_last=ledstateOn;
363             if (ledstateOn) {
364                 msg=msg_on; /*turn on led*/
365             } else {
366                 msg=msg_off; /*turn off led*/
367             }
368
369             msg_info("LED %s!\n", buf);
370             Led_SetState(ledstateOn);
371         }
372
373         if (isDone){
374             msg= msg_done;
375             msg_info("Position Sent %s!\n", buf);
376             /*send position to uart*/
377             strcat(buf, "\r\n");
378             roboArm_UART_send((uint8_t*)&buf);
379         }
380
381         if (isErr) msg= msg_err;
382         /*reset Flags*/
383         isDone = 0;
384         isErr = 0;
385
386         sendParams.payloadLen = strlen(msg);
387         sendParams.payload = (void*) msg;
388
389         IoT_Error_t rc = aws_iot_mqtt_publish(pClient, cPTopicName,
390             strlen(cPTopicName), &sendParams);
391
392         if (rc == AWS_SUCCESS) {
393             msg_info("\nPublished the new operation status to topic %s:", cPTopicName);
394             msg_info("%s\n", msg);
395         }
396     }
397 }
398
399 }
400
401 }
```

☰ RoboArm

\$aws/things/afr_test/shadow/update

Done!

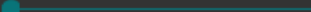
Body

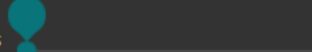
RIGHT

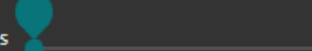
CENTER

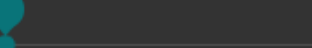
LEFT

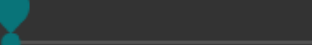
Controls

Speed 

jawsPos 

bodyPos 

bfPos 

udPos 

Jaws

CLOSE

OPEN

U/D Arm

CENTER

UP

BOTTOM

F/B Arm

FORWARD

BACK

CENTER

Readings

Jaws

Body

U/D Arm

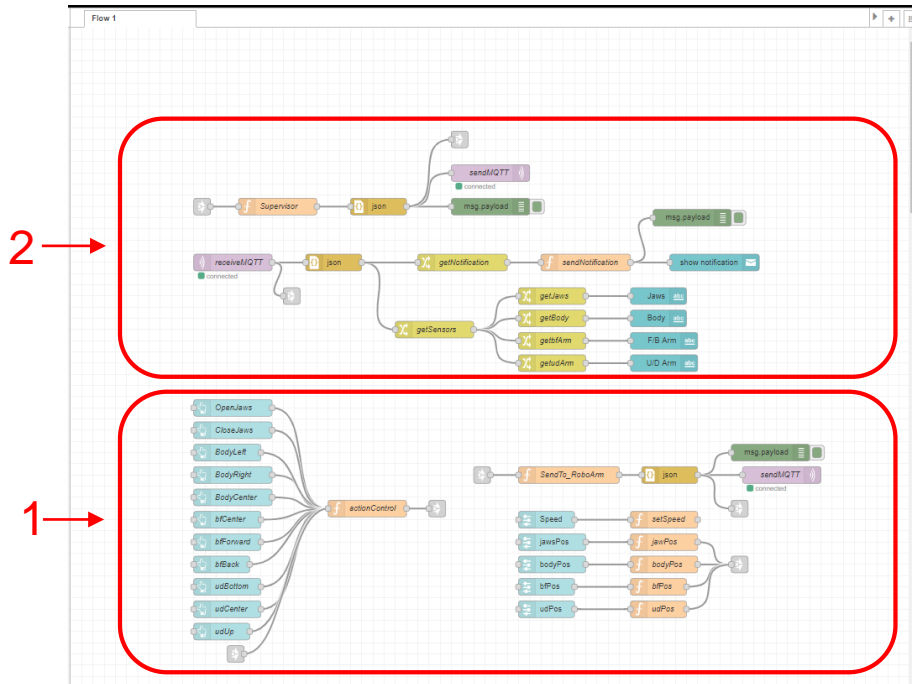
F/B Arm

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:

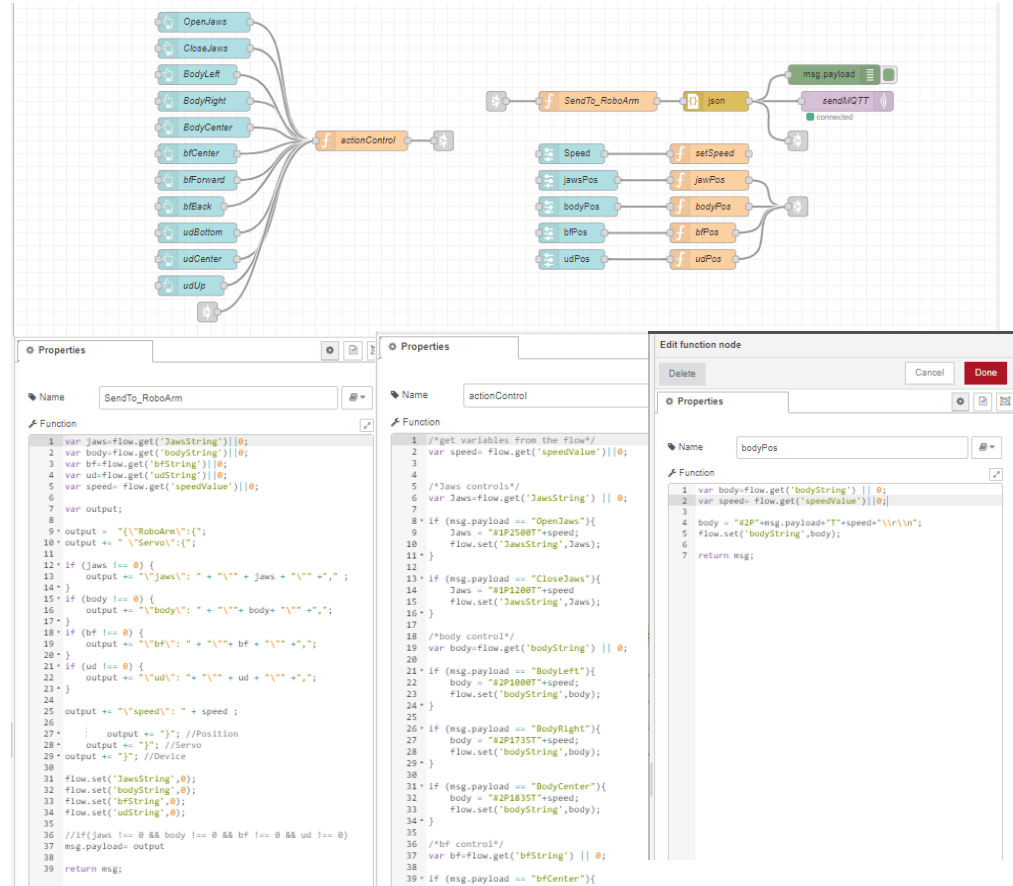
1. Get the inputs from the dashboard and publish them on the topic
2. Get the feedback data sent by the board and display them on the dashboard



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)
```

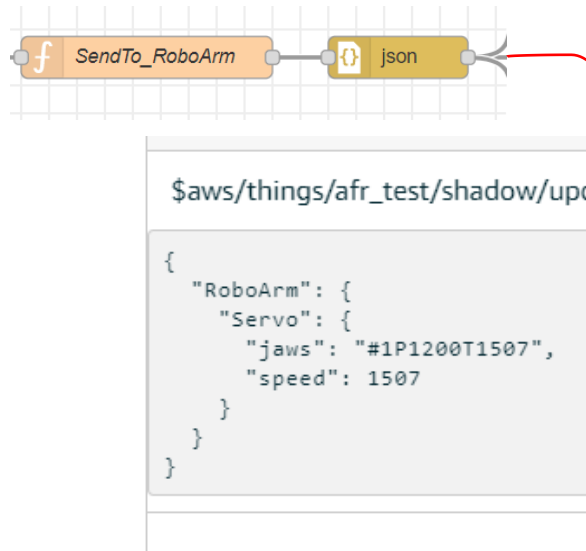
To find more information to manage variables in Node-RED see [here](#).



```
1 var jaws=flow.get('JawsString')||0;
2 var body=flow.get('bodyString')||0;
3 var bf=flow.get('bfString')||0;
4 var ud=flow.get('udString')||0;
5 var speed= flow.get('speedValue')||0;
6
7 var output;
8
9 output = "{\RoboArm\":{";
10 output += "  \Servo\":";
11
12 if (jaws !== 0) {
13   output += "  \jaws\":" + "\"+ jaws + "\"+ "," ";
14 }
15 if (body !== 0) {
16   output += "  \body\":" + "\"+ body+ "\"+ "," ";
17 }
18 if (bf !== 0) {
19   output += "  \bf\":" + "\"+ bf + "\"+ "," ";
20 }
21 if (ud !== 0) {
22   output += "  \ud\":" + "\"+ ud + "\"+ "," ";
23 }
24
25 output += "  \speed\":" + speed ;
26
27   output += "}; //Position
28   output += "}; //Servo
29   output += "}; //Device
30
31 flow.set('JawsString',0);
32 flow.set('bodyString',0);
33 flow.set('bfString',0);
34 flow.set('udString',0);
35
36 //if(jaws !== 0 && body !== 0 && bf !== 0 && ud !== 0)
37 msg.payload= output
38
39 return msg;
```

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
1 var jaws=flow.get('JawsString')||0;
2 var body=flow.get('bodyString')||0;
3 var bf=flow.get('bfString')||0;
4 var ud=flow.get('udString')||0;
5 var speed= flow.get('speedValue')||0;
6
7 var output;
8
9 output = "{\"RoboArm\":{\"";
10 output += " \"Servo\":{\"";
11
12 if (jaws !== 0) {
13   output += "\"jaws\": " + "\"" + jaws + "\" + "," ";
14 }
15 if (body !== 0) {
16   output += "\"body\": " + "\"" + body + "\" + "," ";
17 }
18 if (bf !== 0) {
19   output += "\"bf\": " + "\"" + bf + "\" + "," ";
20 }
21 if (ud !== 0) {
22   output += "\"ud\": " + "\"" + ud + "\" + "," ";
23 }
24
25 output += "\"speed\": " + speed ;
26
27   output += "}\""; //Position
28   output += "}\""; //Servo
29 output += "}\""; //Device
30
31 flow.set('JawsString',0);
32 flow.set('bodyString',0);
33 flow.set('bfString',0);
34 flow.set('udString',0);
35
36 //If(jaws !== 0 && body !== 0 && bf !== 0 && ud !== 0)
37 msg.payload= output
38
39 return msg;
```

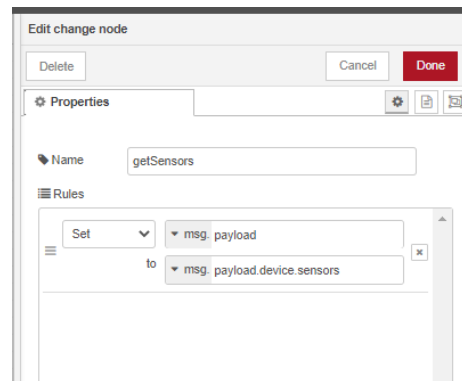
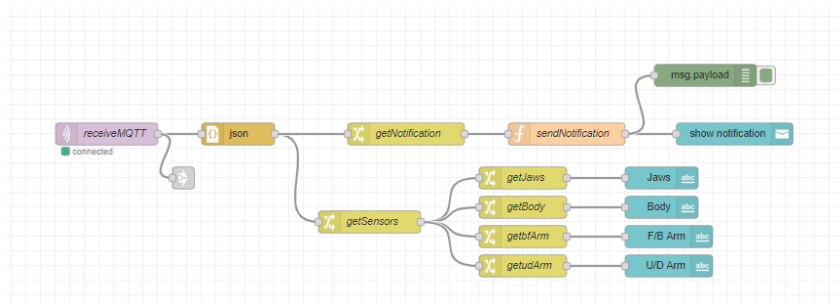
Get sensors values and notifications

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

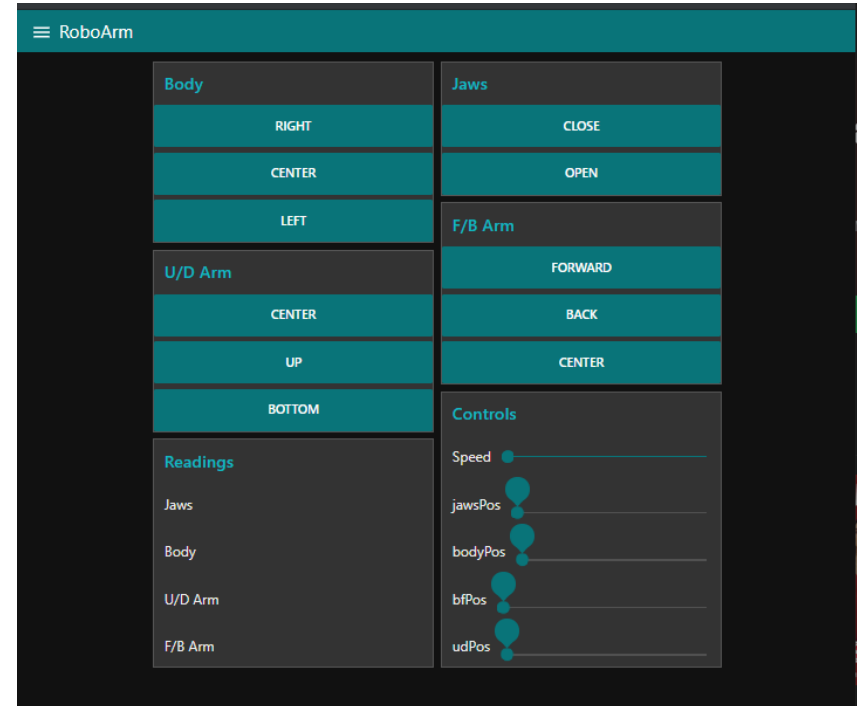
For doing this is better to use a **change node** so if the message is **undefined** (for example we send the position of only one motor and the message does not contain all the motors positions the payload will be undefined) nothing will be triggered.



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.

```
MQTT subscribe callback.....  
{ "RoboArm":{ "Servo":{"jaws": "#1P1200T1400", "speed":1400}}}   
Move jaws: 1200!  
Position Sent #1P1200T1400!  
  
Published the new operation status to topic $aus/things/afr_test/shadow/update:{"state":{"reported":{"stat":  
  
MQTT subscribe callback.....  
{"state":{"reported":{"stat":"Done!";},"metadata":{"reported":{"stat":{"timestamp":1590488395}}},"version
```

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.

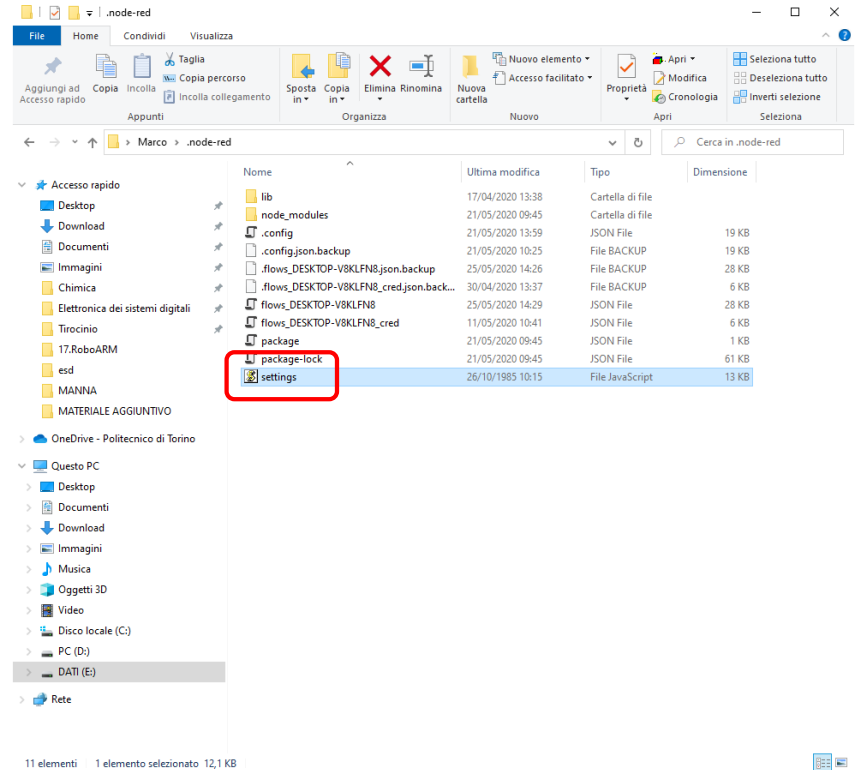
There is still a big problem, **security**.

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

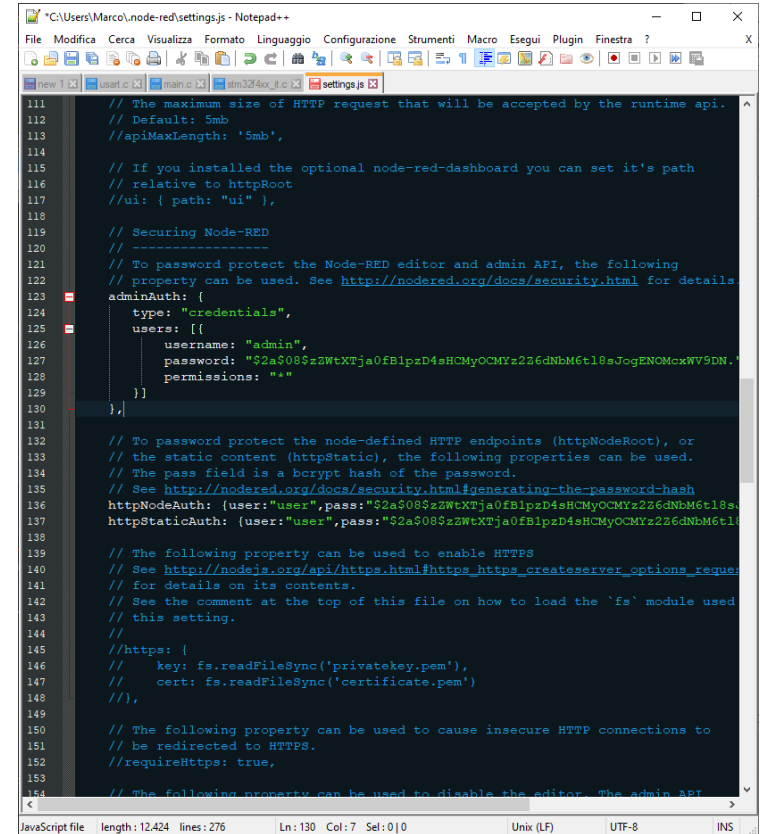
Find the Securing Node-RED section and uncomment the lines as in the picture.

The password is encrypted 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.



```
111 // The maximum size of HTTP request that will be accepted by the runtime api.
112 // Default: 5mb
113 //apiMaxLength: '5mb',
114
115 // If you installed the optional node-red-dashboard you can set it's path
116 // relative to httpRoot
117 //ui: { path: "ui" },
118
119 // Securing Node-RED
120 // -----
121 // To password protect the Node-RED editor and admin API, the following
122 // property can be used. See http://nodered.org/docs/security.html for details
123 adminAuth: {
124   type: "credentials",
125   users: [
126     {
127       username: "admin",
128       password: "$2a$08$z2WtXTja0fB1pzD4sHcMyOCMyz226dNbM6t18aJogENOMcxWV9DN.",
129       permissions: "*"
130     }
131   ],
132
133   // To password protect the node-defined HTTP endpoints (httpNodeRoot), or
134   // the static content (httpStatic), the following properties can be used.
135   // The pass field is a bcrypt hash of the password.
136   // See http://nodered.org/docs/security.html#generating-the-password-hash
137   httpNodeAuth: {user:"user",pass:"$2a$08$z2WtXTja0fB1pzD4sHcMyOCMyz226dNbM6t18aJogENOMcxWV9DN."},
138   httpStaticAuth: {user:"user",pass:"$2a$08$z2WtXTja0fB1pzD4sHcMyOCMyz226dNbM6t18aJogENOMcxWV9DN."},
139
140   // The following property can be used to enable HTTPS
141   // See http://nodejs.org/api/https.html#https\_https\_createserver\_options\_requestlistener
142   // for details on its contents.
143   // See the comment at the top of this file on how to load the 'fs' module used
144   // this setting.
145   //
146   //https: {
147   //   key: fs.readFileSync('privatekey.pem'),
148   //   cert: fs.readFileSync('certificate.pem')
149   //},
150
151   // The following property can be used to cause insecure HTTP connections to
152   // be redirected to HTTPS.
153   //requireHttps: true,
154
155   // The following property can be used to disable the editor. The admin API
```


Create an hash password

In the same window type the line

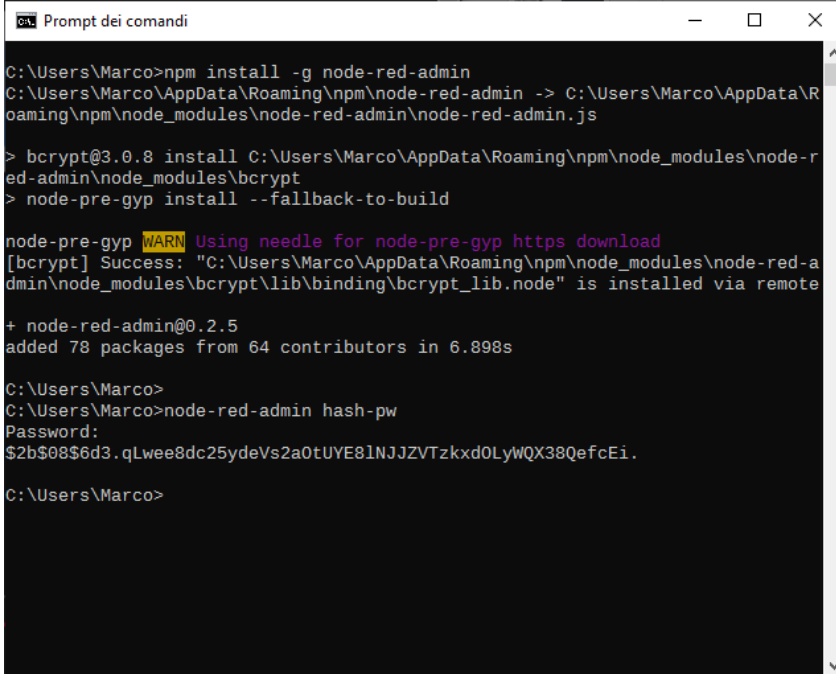
node-red-admin hash-pw

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



```

C:\Users\Marco>npm install -g node-red-admin
C:\Users\Marco\AppData\Roaming\npm\node-red-admin -> C:\Users\Marco\AppData\Roaming\npm\node_modules\node-red-admin\node-red-admin.js

> bcrypt@3.0.8 install C:\Users\Marco\AppData\Roaming\npm\node_modules\node-red-admin\node_modules\bcrypt
> node-pre-gyp install --fallback-to-build

node-pre-gyp WARN Using needle for node-pre-gyp https download
[bcrypt] Success: "C:\Users\Marco\AppData\Roaming\npm\node_modules\node-red-admin\node_modules\bcrypt\lib\binding\bcrypt_lib.node" is installed via remote
+ node-red-admin@0.2.5
added 78 packages from 64 contributors in 6.898s

C:\Users\Marco>
C:\Users\Marco>node-red-admin hash-pw
Password:
$2b$08$6d3.qLwee8dc25ydeVs2a0tUYE8lNJZVTzkxd0LyWQX38QefcEi.

C:\Users\Marco>
```