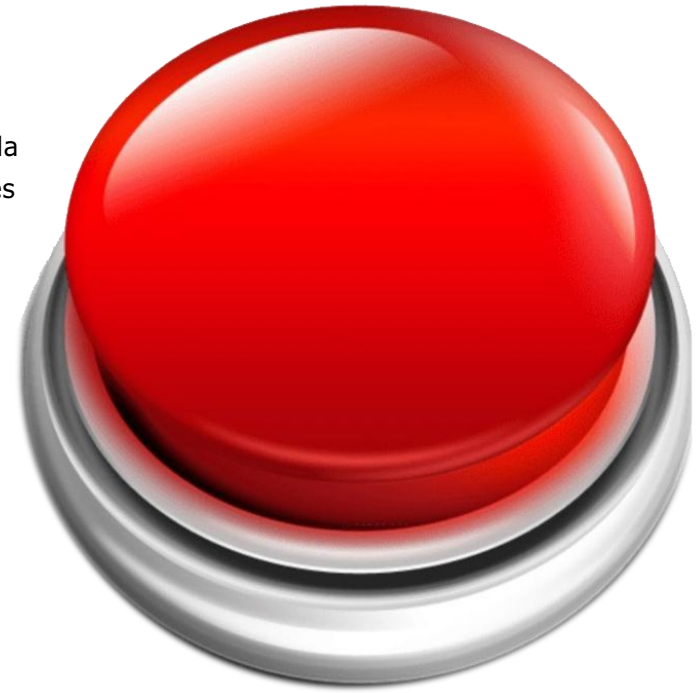
dessine-moi un bouton pour l'IdO

Boutonnière nº 2 : Prototypage entre la matière et le nuage

Veikko Krypczyk (Allemagne)



L'idée d'un bouton pour l'IdO repose sur le principe de la communication entre un matériel réduit à peu de choses et un service distant dans le nuage, virtuellement sans limite. Il faut pour cela établir une connexion au réseau local. Pour cette tâche, les petites cartes à microcontrôleurs pas chères comme l'ESP8266 NodeMCU sont idéales. Nous mettrons en lumière ici les aspects de l'IdO pertinents pour cet usage et présentons des idées pour des projets personnels. Tout est relativement facile à gérer et à la portée de chacun. À vous ensuite de prolonger...

L'internet des objets (IdO) rend potentiellement intelligents toutes sortes de dispositifs pourtant bien modestes, en les reliant au monde extérieur grâce à l'internet. Dans la première partie de cet article publiée dans le précédent numéro, nous avons décrit l'architecture de base d'un paysage d'IdO, composé de capteurs/actionneurs et d'une infrastructure (*backend*) de serveur. Vous n'aurez ni à construire ni à programmer vous-même l'infrastructure. Il suffira de tirer parti de l'offre diversifiée dématérialisée dans le nuage. La communication passe par l'internet. C'est simple si vous vous en tenez à des normes établies. Pour les demandes de renseignements (= requêtes), la communication basée sur *REST* a fait ses preuves. Le client, dans ce cas le dispositif IdO, peut transmettre des données à des services (sur un serveur) ou les interroger. Cette deuxième partie de l'article traitera du matériel et des logiciels du côté des dispositifs de l'IdO.

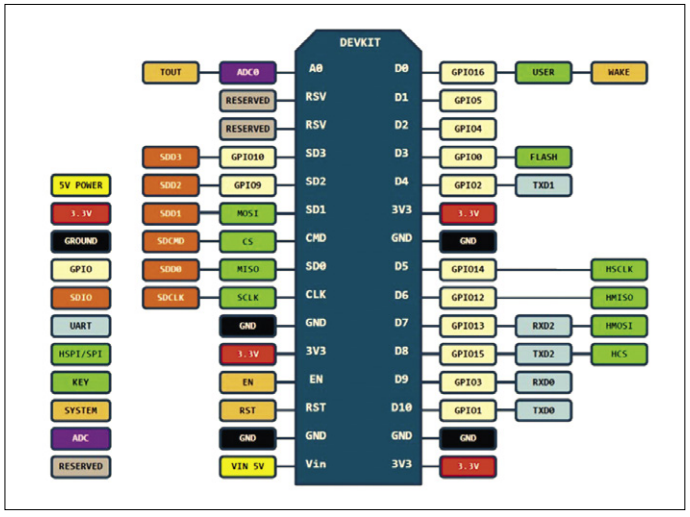


Figure 1. Affectation des broches de la carte microcontrôleur ESP8266 NodeMCU.

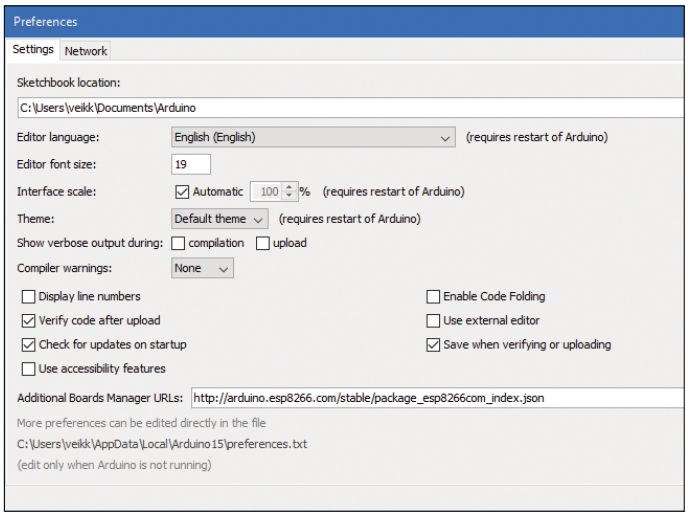


Figure 2. Configuration de l'EDI Arduino pour la programmation de l'ESP8266.

Sélection du matériel

Quel matériel utiliser ? Il existe diverses options, p. ex. un nano-ordinateur comme le **Raspberry Pi**. Outre le WLAN, le RPi offre une puissance de calcul suffisante pour toutes les applications. Si nous voulions actionner, dans un même lieu, plusieurs boutons, capteurs et actionneurs ayant des fonctions différentes, tous ces dispositifs seraient connectés aux broches GPIO du Raspberry Pi. Le RPi utiliserait le réseau pour se connecter au service dans le nuage. Pour des tâches réduites, un Raspberry Pi serait trop encombrant et il exigerait une alimentation permanente et un bloc d'alimentation. Trop gourmand.

Voyons l'**Arduino Uno** ! Comme on sait, ce n'est pas un ordinateur, mais une carte à µC. Ses possibilités matérielles et logicielles sont vastes et les bibliothèques et les exemples d'application sont innombrables. La programmation est aisée grâce à l'IDE Arduino. Cependant, l'Arduino ne sait rien faire du WLAN, de sorte que pour cette fonction cruciale pour l'IdO, il faudrait lui adjoindre un module supplémentaire. Trop lourd.

Voyons alors une carte à microcontrôleur **ESP8266 NodeMCU** ([1] et **encadré**). Elle réunit toutes les fonctions nécessaires, dont l'interface WLAN (**fig. 1**). Il suffit de connecter un poussoir (c'est notre bouton IdO) entre la borne D7 sur la carte et GND. Quand on appuie sur ce bouton, un signal doit être transmis par WLAN. Il y a une LED d'état sur la carte, nous n'avons donc besoin d'aucun autre composant. L'alimentation est fournie par le port micro-USB de l'ordinateur, pour l'instant.

Préparation du NodeMCU ESP8266

Comme les cartes Arduino, les µC ESP8266 peuvent être programmés dans l'IDE Arduino. Il faut d'abord connecter la carte avec un câble de données au PC via le port USB et, si nécessaire, installer le pilote pour le jeu de puces CH340 [2]. Si c'est déjà fait chez vous, sautez cette étape, sinon installez l'EDI Arduino sur votre ordinateur [3]. L'IDE doit être complété par un élément supplémentaire : sous *Fichier -> Préférences, entrez* le lien [4] pour les *URL supplémentaires des gestionnaires de cartes (***fig. 2).** Fermez la boîte de dialogue avec *OK*. L'installation proprement dite s'effectue sous le menu *Outils,* où se trouve l'entrée *Modules ESP8266 génériques.* Sélectionnez-la et cliquez sur le *gestionnaire de cartes* dans le menu contextuel en haut, réduisez la zone de recherche avec l'entrée *NodeMCU* et effectuez l'installation (**fig. 3**).

Un simple programme «Hello World» (**listage 1**) permet de vérifier si tout a bien fonctionné. Copiez le code et chargez-le dans le µC à l'aide de *Sketch|Upload.* La LED sur la carte est éteinte par la petite boucle avec `digitalWrite(LED, HIGH)` pendant 3 s et allumée par `digitalWrite(LED, LOW)` pendant 1 s.

Pour interroger un bouton (relié à la broche D7), un peu de code suffit. Celui-ci vérifie par exemple si le bouton a été fermé :

```
int bouton=D7 ;
status=digitalRead(button) ;
if (status==LOW)
{ ……  }
```

avant de déclencher des opérations ultérieures.

Il ne reste plus qu'à accéder à l'internet. Le matériel néces-

Listage 1. «Hello World» (LED clignotante) sur l'ESP8266

```
#define LED D4
void setup()
{
     pinMode(LED, OUTPUT);
}

void loop()
{
    digitalWrite(LED, HIGH);
    delay(3000);
    digitalWrite(LED, LOW);
    delay(1000);
}
```

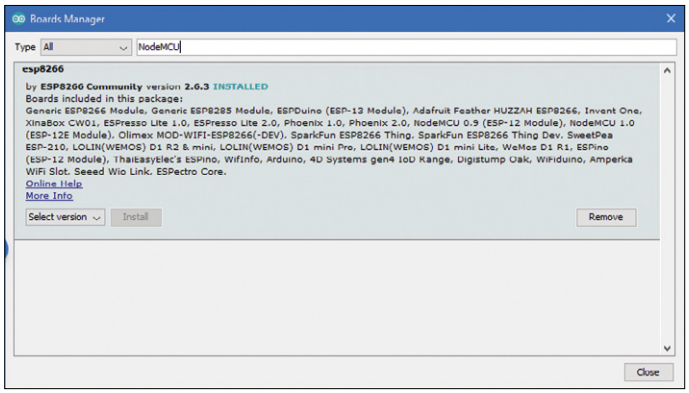


Figure 3. Installation des bibliothèques pour la carte ESP8266 NodeMCU.

La carte Node ESP8266MCU

Sur la carte ESP8266 NodeMCU on trouve la puce ESP8266 avec une vitesse d'horloge de 80 MHz et jusqu'à 4 Mo de mémoire flash. Le NodeMCU ESP8266 contient un module WLAN et une interface USB avec convertisseur UART/USB (CH340G).

La petite carte présente les caractéristiques techniques suivantes :

- soutient le WPA/WPA2
- puce : ESP8266
- processeur : Tensilica L106 – horloge : 80 MHz
- architecture du système : 32 bits, Flash : jusqu'à 4 Mo
- protocoles pris en charge : SPI, UART, I²C, I2S, IR Remote Control, PWM, GPIO
- niveau logique : 3,3 V
- consommation : < 10 µA en veille et jusqu'à 200 mA en charge
- alimentation et connexion de programmation via Micro-USB : 5 V
- puce de programmation : CH340G
- taille : 5,5 x 3,1 x 1,2 cm ; poids : 42 g

Ces spécifications montrent que la carte ESP8266 NodeMCU est bien adaptée pour faire fonctionner de petits dispositifs d'IdO particulièrement économes en énergie

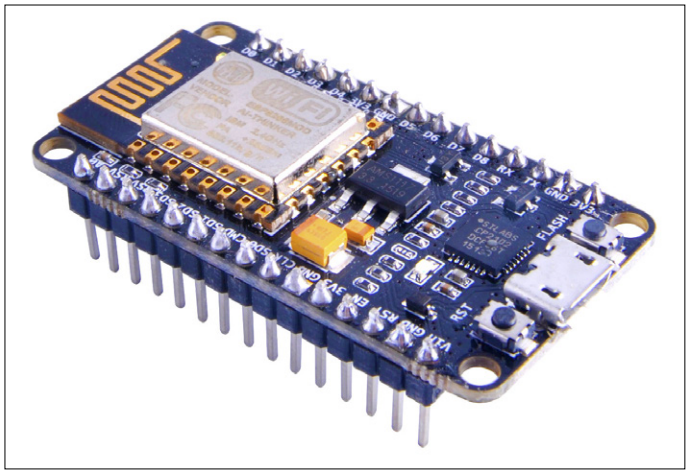


Figure 4. ESP8266 NodeMCU avec module WLAN

Listage 2. Se connecter à l'internet

```
include <ESP8266WiFi.h>
void setup()
{
    Serial.begin(115200);

    WiFi.begin(<SSID>, <Password>);

    while (WiFi.status() != WL_CONNECTED);
    {
        delay(1000);
        Serial.print("Connecting ...");
    }
}
```

saire se trouve déjà sur la carte de l'ESP8266 sous la forme d'un minuscule module WLAN (gris-vert) (**fig. 4**).
Si vous voulez utiliser le WLAN de l'ESP8266, il faut inclure une bibliothèque avec la clause `#include` *ESP8266WiFi.h* dans l'en-tête du code source. La bibliothèque attend les données correctes pour la connexion à l'internet, le SSID et le mot de passe. Ces données sont (d'abord statiquement) stockées dans le code source, plus tard l'utilisateur devrait bien sûr pouvoir configurer les données. L'établissement d'une connexion WLAN est merveilleusement simple (**listage 2**).

Il faut maintenant procéder de telle sorte que lorsqu'un événement se produit, nous envoyons un message sur le réseau. Un terminal REST, tel que décrit dans la première partie de cet article, servira de récepteur dans cette communication sur le réseau. Ce point de terminaison reçoit les données ou le message et peut en faire un traitement ultérieur et déclencher une réaction. Avec un simple bouton IdO, il n'y a aucune autre donnée à transmettre que l'événement «bouton actionné». Si l'application était étendue en conséquence, des données supplémentaires pourraient être transmises en tant que paramètres par la requête REST. Essayons !

Données dans le nuage

Un terminal REST est nécessaire pour envoyer des données dans le nuage, c'est-à-dire vers un serveur distant. Vous n'avez pas besoin de programmer cela vous-même, vous pouvez utiliser les services existants. Pour les tests et les applications privées, ces services en nuage sont généralement gratuits. Nous voulons procéder de manière expérimentale, étape par étape. Sous [5], vous pouvez créer une API REST gratuite à des fins de test, à laquelle vous pouvez accéder à partir de notre µC via le WLAN. Cela fonctionne en quelques clics de souris, même sans enregistrement préalable. Regardez [6] ; nous avons même déjà créé un point de terminaison (**fig. 5**) ! Pour vérifier si et comment ce point d'arrivée est atteint à

Listage 3. Envoyer la demande GET au point de terminaison REST

```
void loop()
{
  if (WiFi.status() == WL_CONNECTED)
  {
     status=digitalRead(button);
     if (status==LOW)
     {
      digitalWrite(LED, LOW);

      BearSSL::WiFiClientSecure client;
      client.setInsecure();
      HTTPClient https;
      https.begin(client, "https://iotbutton.free.
                           beeceptor.com");
      int httpCode = https.GET();
      if (httpCode == 200)
      {
        String payload = https.getString();
        Serial.println(payload);
      }
      else
      {
        Serial.print("Erreur: ");
        Serial.println(httpCode);
      }
      https.end();

     }
     else
     {
      digitalWrite(LED, HIGH);
     }

  }
}
```

partir du µC, nous envoyons une demande GET au service lorsque le bouton est enfoncé. Il est inséré dans la boucle du programme (**listage 3**).
La ligne de code suivante vérifie si le µC est connecté à l'internet :

```
if (WiFi.status() == WL_CONNECTED)
{  }……
```

Toutes les lignes incluses du code source ne sont donc traitées que si une connexion à l'internet a été établie.
L'étape suivante

```
status=digitalRead(button) ;
     if (status==LOW)
```

vérifie si le bouton a été actionné. Si c'est le cas, un client http est initialisé et l'option de transmission de *commandes https est* donc activée. Cela nécessite une bibliothèque, qui doit être incluse dans l'en-tête avec `#include.` Pour spécifier l'URL de notre point terminal REST, la valeur `https://iotbutton.free.beeceptor.com` est définie. Elle est suivie d'une simple requête GET sur le point final, dont le résultat peut être spécifié par la ligne

```
String payload = https.getString() ;
```

comme une chaîne de caractères.
En réaction directe à l'appui sur le bouton, la LED de la carte doit être allumée, puis éteinte à nouveau dès que le bouton sera relâché (voir ci-dessus). Il est possible de vérifier en ligne si les données sont également arrivées dans le nuage. Allez au point terminal de la page Beeceptor [6] et suivez-y les requêtes au point terminal (**fig. 6**).

Notez que l'envoi gratuit de requêtes par ce service est limité à 50 par jour. Au cours de nos essais, nous avons dépassé ce nombre et avons reçu un message d'erreur avec le code 429 sur la requête GET comme valeur de retour. Toutefois, tant que vous ne dépassez pas le nombre maximum de requêtes, le code de statut 200 (OK) et les données de la demande REST sont renvoyés au µC. Dans l'environnement de développement, vous pouvez suivre dans le moniteur série *(Outils|Moniteur série)* les valeurs de retour reçues par le µC (**fig. 7**).
En appuyant sur un bouton, nous avons donc transmis une demande REST par l'internet. C'est le côté technique du bouton IdO. Le **listage 4** indique le code source complet. Les zones à adapter individuellement sont indiquées en gras.

Courant et sommeil

Nous avons choisi la carte ESP8266 en raison notamment de sa faible consommation. On peut se fier aux valeurs de consommation mentionnées dans [7]. Avec un simple programme Arduino, le courant est d'environ 50 à 70 mA, en fonctionnement WLAN la consommation atteint environ 80…170 mA et pendant quelques millisecondes peut même culminer à 400 mA. D'où l'intérêt d'une réduction drastique de la consommation en mode de sommeil profond. Selon diverses sources, on parle de quelques microampères. Une batterie (rechargeable) devrait tenir des mois. La tension d'alimentation est connectée aux bornes +5V et GND. Il existe deux variantes du mode de sommeil profond.

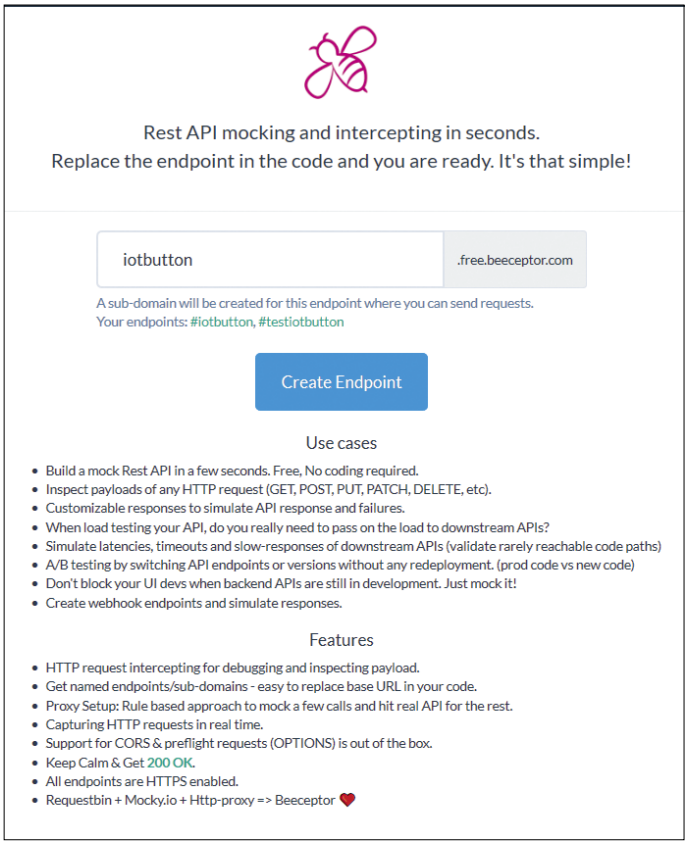


Figure 5. Point final REST pour le test sur le nuage de *Beeceptor*.

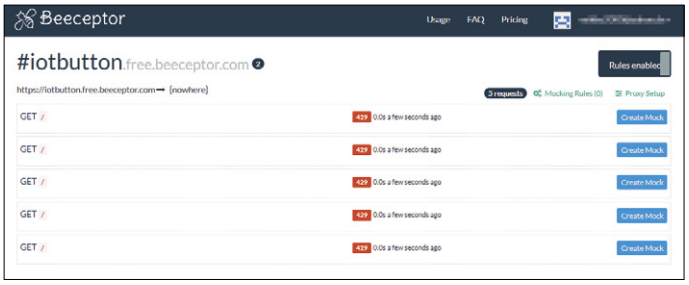


Figure 6. Demandes enregistrées vers l'extrémité REST sur le nuage *Beeceptor*

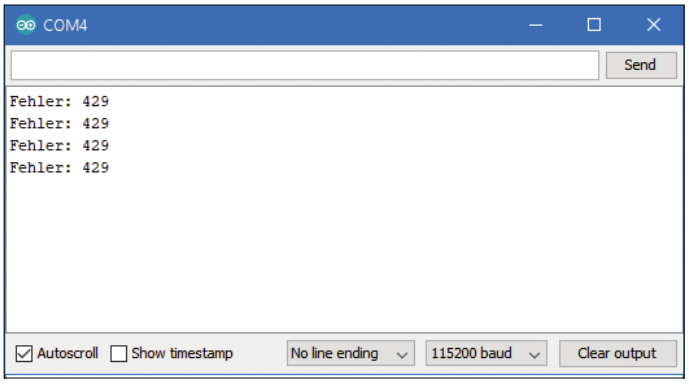


Figure 7. Le Serial Monitor enregistre les réponses du point final REST. Ici, le message d'erreur 429 est émis parce que le nombre de demandes a été dépassé.

Listage 4. Envoi d'une demande GET via REST

```
#include <ESP8266WiFi.h>
#include <ESP8266HTTPClient.h>
#define LED D4
int button=D7;
int status=1;
void setup()
{
    Serial.begin(115200);
    pinMode(LED, OUTPUT);
    digitalWrite(LED, HIGH);
    WiFi.begin(<SSID>, <Password>);
    while (WiFi.status() != WL_CONNECTED)
    {
        delay(1000);
        Serial.print("Connecting ...");
    }
}

void loop()
{
  if (WiFi.status() == WL_CONNECTED)
  {
     status=digitalRead(button);
     if (status==LOW)
     {
      digitalWrite(LED, LOW);
      BearSSL::WiFiClientSecure client;
      client.setInsecure();
      HTTPClient https;
      https.begin(client, "https://iotbutton.free.
                           beeceptor.com");
      int httpCode = https.GET();
      if (httpCode == 200)
      {
        String payload = https.getString();
        Serial.println(payload);
      }
      else
      {
        Serial.print("Erreur: ");
        Serial.println(httpCode);
      }
      https.end();

     }
     else
     {
      digitalWrite(LED, HIGH);
     }
  }
}
```

Listing 5. Envoi d'une demande GET par le mode REST et le mode sommeil profond

```
#include <ESP8266WiFi.h>
#include <ESP8266HTTPClient.h>

#define LED D4
void setup()
{
    Serial.begin(115200);
    pinMode(LED, OUTPUT);
    digitalWrite(LED, HIGH);
    WiFi.begin(<SSID>, <Password>);
    while (WiFi.status() != WL_CONNECTED)
    {
        delay(1000);
        Serial.print("Connecting ...");
    }
}

void loop()
{
  if (WiFi.status() == WL_CONNECTED)
  {
      digitalWrite(LED, LOW);
      BearSSL::WiFiClientSecure client;
      client.setInsecure();
      HTTPClient https;
      https.begin(client, "https://iotbutton.free.
                           beeceptor.com");
      int httpCode = https.GET();
      if (httpCode == 200)
      {
        String payload = https.getString();
        Serial.println(payload);
        digitalWrite(LED, HIGH);
        delay(2000);
        ESP.deepSleep(0);
      }
      else
      {
        Serial.print("Erreur: ");
        Serial.println(httpCode);
      }
      https.end();
  }
}
```

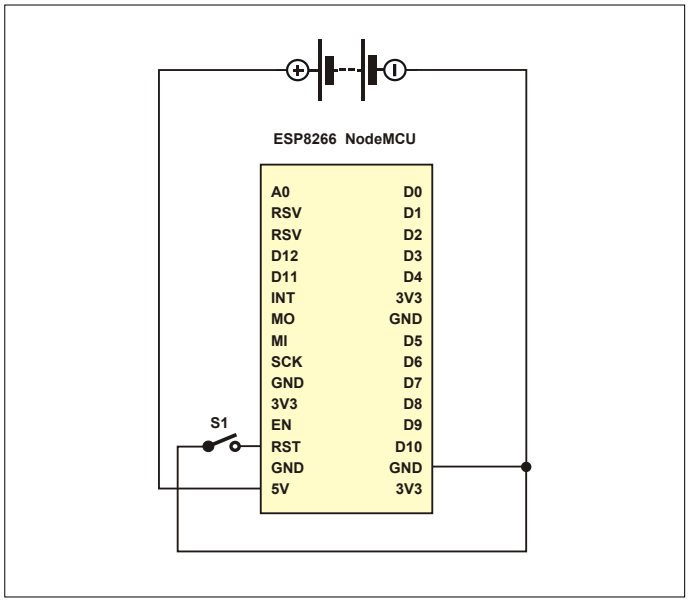


Figure 8. Le schéma de circuit ne contient qu'un seul bouton poussoir en plus du microcontrôleur.

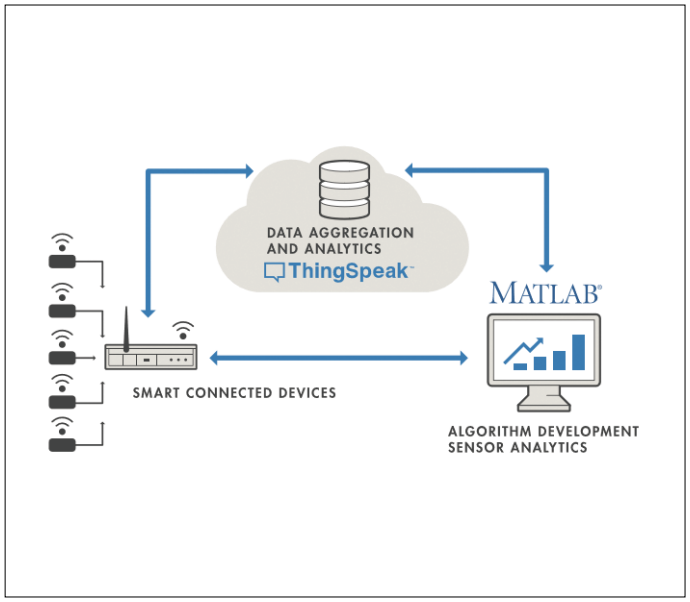


Figure 9. Collaboration entre l'IdO-Device, l'IdO-Cloud et d'autres services en utilisant l'exemple de *ThingSpeak*.

- *Le sommeil profond des logiciels* : ESP8266 se réveille automatiquement après un certain temps. Pour ce faire, une connexion doit être établie entre le GPIO16 (D2) et le connecteur RST. Après le délai fixé, un signal est envoyé qui redémarre automatiquement l'ESP. Toutefois, la durée maximale est limitée à environ 71 minutes. Cette option est un bon choix dans les scénarios qui transmettent des valeurs mesurées de manière cyclique. La commande correspondante est : `ESP.deepSleep(SLEEP_TIME * 1000000)`
- *Interrompre le sommeil profond* : Le ESP8266 ne redémarre pas automatiquement, c'est une interruption qui le tire de son sommeil profond. Ce mode de sommeil profond est appelé avec : `ESP.deepSleep (0)`. Toutefois, GPIO16 et RST ne doivent pas être reliées. Dès que l'événement est déclenché, on envoie un signal bas (*LOW*) à la broche RST. Cela permet de redémarrer l'ESP8266 et de faire exécuter le logiciel.

Pour le bouton IdO, cela signifie que le bouton n'est plus connecté à une broche GPIO, mais à une broche RST (et à la terre) (**fig. 8).** Dans le code, la vérification de la frappe peut donc être omise. Au lieu de cela, le µC est mis en mode veille (**listage 5**) après avoir envoyé avec succès les informations par l'interface REST. Compilez le code source (*sketch*) et chargez-le dans le contrôleur. Après le démarrage, il se connecte au réseau, envoie un message et se met en mode veille. Si le microcontrôleur est réveillé par le bouton (*Reset*), le processus est relancé. Vous pouvez suivre en ligne comment l'API REST a été appelée avec succès par la demande GET.

Encore une fois dans le nuage

Techniquement, le bouton IdO est en place. Il est possible à présent de configurer en quelques étapes les actions effectuées à la réception du message du µC. Pour ces scénarios également, il existe différents services en nuage, tels que *Microsoft*

Liens

[1] Manuel de l'utilisateur du NodeMCU ESP8266 : www.handsontec.com/pdf_learn/esp8266-V10.pdf

[2] Pilote CH340 : https://sparks.gogo.co.nz/ch340.html

[3] IDE Arduino : http://www.arduino.cc/en/main/software

[4] URL du gestionnaire de cartes : http://arduino.esp8266.com/stable/package_esp8266com_index.json

[5] Beeceptor: https://beeceptor.com/

[6] Console IoT-Button: https://beeceptor.com/console/iotbutton

[7] Consommtion de l'ESP8266: https://arduino-hannover.de/2018/07/25/die-tuecken-der-esp32-stromversorgung/

[8] Nuage IdO Microsoft : https://azure.microsoft.com/en-en/services/iot-hub/

[9] ESP8266 dans le nuage Azure : https://sandervandevelde.wordpress.com/2019/05/07/connection-a-cheap-esp8266-to-azure-iot-central/

[10] NodeMCU dans le nuage Azure : https://www.thingforward.io/techblog/2018-05-22-connecting-nodemcu-to-microsoft-azure-iot-hub-part-1.html

[11] Thingspeak : https://thingspeak.com/

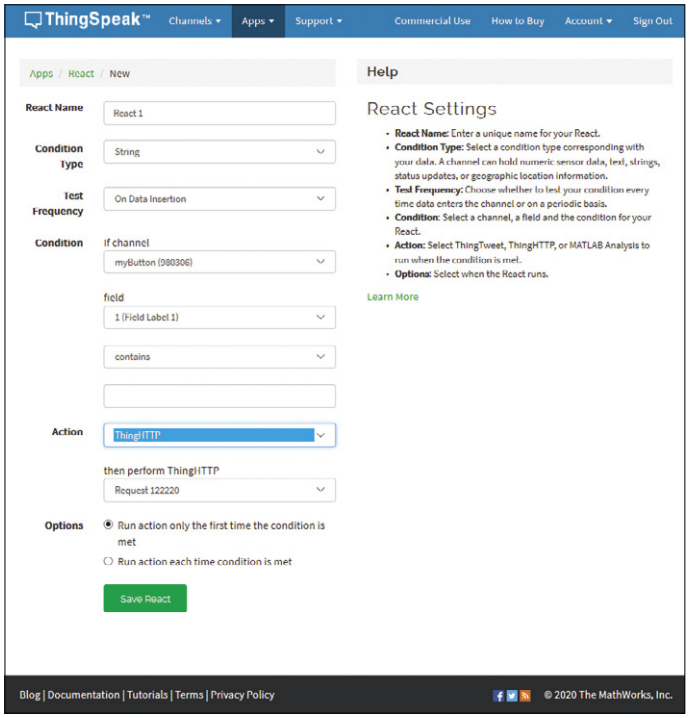


Figure 10. Configuration du point final REST (canal) pour *ThingSpeak*.

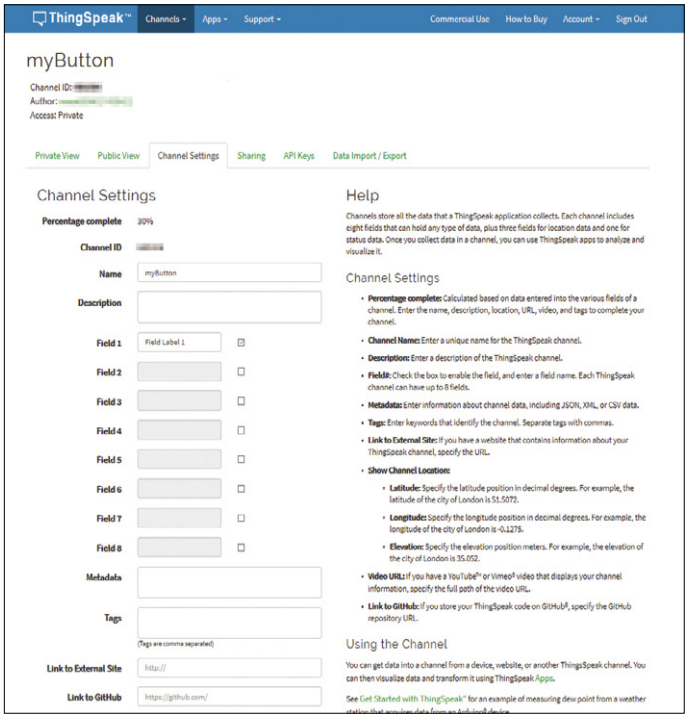


Figure 11. Les applications déterminent les actions de *ThingSpeak* pour une chaîne.

Azure IoT [8][9][10] ou *ThingSpeak* [11]. L'approche consiste essentiellement à configurer un point de terminaison en ligne directement dans le service en nuage et à l'adresser via l'internet à partir du dispositif IdO. Les actions souhaitées peuvent ensuite être lancées par le service (**fig. 9**).

Créez un compte gratuit avec le service *ThingSpeak*, lequel ne nécessite qu'une vérification par adresse électronique, puis créez un nouveau point de terminaison REST (*canal*).

Remplissez le formulaire détaillé (**fig. 10**) suivant les explications données dans la partie droite, ou donnez-lui d'abord un nom. Chaque canal se voit attribuer une URL unique et la possibilité d'envoyer des données supplémentaires au-delà de la demande à l'aide de paramètres URL. Pour un simple bouton IdO, aucun autre paramètre n'est nécessaire, mais sous *Mes canaux/clés API, vous pouvez*trouver les clés de lecture et d'écriture (et des exemples sur la façon d'inclure les clés dans les requêtes GET) du canal. Si un canal est correctement configuré, vous pouvez utiliser l'élément de menu *Apps* pour définir les actions à déclencher (**fig. 11**), p. ex. l'envoi d'un e-mail par un service web ou la publication d'un micromessage sur *Twitter*. L'infrastructure (*backend*) de l'IdO sert pour ainsi dire de couche intermédiaire.

Attention : pour communiquer avec les services du réseau, les dispositifs IdO doivent forcément avoir accès à ces services. Pour cela, ils doivent bien sûr s'identifier, p. ex. avec un nom d'utilisateur et un mot de passe ou avec d'autres caractéristiques uniques (clés). Ces valeurs doivent être stockées dans le code source. Veillez donc à protéger ces informations, notamment lorsque vous copiez le code source ou que vous le publiez sur l'internet (pour un usage gratuit).

Conclusion et perspectives

Ne sont-elles pas fascinantes, ces expériences avec le µC ESP8266 ? Que vous manque-t-il pour mettre en œuvre des scénarios de l'IdO ? Pour rendre opérationnel le projet du bouton IdO ou pour le transformer en application commerciale, il ne manque plus grand-chose : mécanique (boîtier), alimentation (connexion permanente de la batterie). Des extensions sont envisageables : si par exemple quelqu'un s'approche sans autorisation d'un objet sécurisé, notre bouton IdO déclenché par un détecteur de mouvement pourrait directement déclencher l'alarme et communiquer via l'internet. Ce travail-là sera effectué en ligne, par une configuration appropriée dans le nuage.

190303-B-02