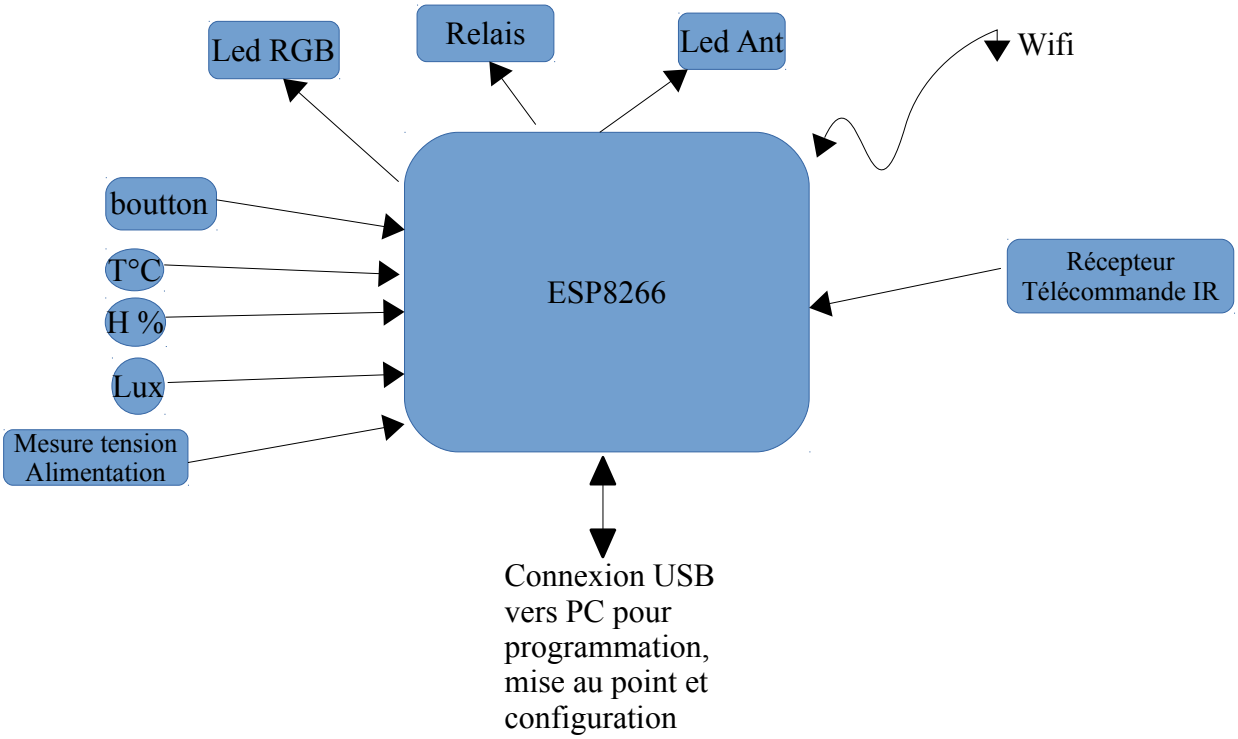
STNM1 2020
Objet Connecté

Ce tp a pour but de créer un objet connecté de démonstration offrant un serveur web permettant de visualiser les valeurs de différents capteurs (température,humidité,luminosité,…) et de commander des actions (par l'intermédiaire d'un relais). L'objet va être également connecté à un *broker mqtt* pour permettre la prise de contrôle par une application sur téléphone.

Architecture matérielle

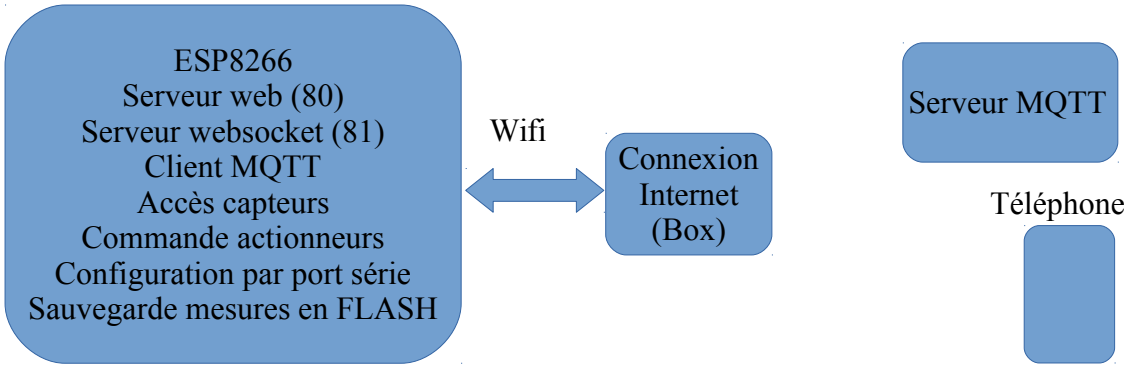



```
#define RELAY D0    // LED inverted logic / RELAY normal logic
#define LED_ANT D4  // inverted logic
#define BTN_FLASH  D3  // pushed 0 / released 1
#define RED_LED D6
#define GREEN_LED D7
#define BLUE_LED D8
#define DHTPIN D3
```

Documentations :

https://arduino-esp8266.readthedocs.io
https://www.arduino.cc/reference/en/
https://tttapa.github.io/ESP8266/Chap01%20-%20ESP8266.html

Architecture logicielle



Le principe général du programme de l'ESP8266 est le suivant :
Au démarrage, il va chercher à se connecter à un réseau wifi connu.
S'il n'en trouve pas il va attendre une configuration par le port série puis redémarrer.
Il va ensuite démarrer un serveur web (port 80) et un serveur websocket (port 81)
Il va périodiquement relever l'état de capteurs et stocker les valeurs dans des fichiers CSV stockés en mémoire FLASH.
Sur le serveur web une page permettra de visualiser l'état des capteurs et de commander les différents actionneur.
Il va se connecter à un serveur MQTT pour publier les valeurs de capteurs et permettre l'action sur les actionneurs (relais, LED).

Structure du code :
De manière à bien séparer les différentes fonctionnalités, on les codera dans des fichiers différents. Un fichier principal appellera les initialisations (*setup()*) et les fonctions *loop()* de tous les autres :
`iot.ino` [1] : fichier principal
`hardware.h` : définition des E/S
`config.h` : définition de la configuration
`wifi.h / wifi.cpp` : connexion wifi
`sensors.h / sensors.cpp` : lecture des capteurs / sauvegarde de données
`webserver.h / webserver.cpp` : serveur web
`websocket.h / websocket.cpp` : serveur websocket
`mqtt.h / mqtt.cpp` : client mqtt

Le fichier principal aura la structure suivante :

```
void setup(){
      setup_wifi();
      setup_sensors();
      setup_webserver();
```

1 Attention pour l'IDE arduino le nom doit être *identique* au répertoire contenant le fichier.

```
      setup_websocket();
      setup_mqtt();
  }

void loop(){
      loop_sensors();
      loop_webserver();
      loop_websocket();
      loop_mqtt();
}
```

La configuration sera stockée dans la structure ci-dessous :

```
struct CONFIG {
      char ssid[32] ; // ssid of wifi network to connect to
      char passwd[64]  ;
      char myhostname[32] ; // the name of the ESP (used by .local domain)
      IPAddress mqttServerIp ;
      unsigned long sensorsPeriod ; // milliseconds
} ;
```

On ajoutera des champs au fur et à mesure des besoins.
Au début on déclarera la variable de configuration dans le fichier principal et on la remplira dans le setup(). Par la suite on verra comment la modifier et la relire d'une valeur stockée en mémoire non volatile.

1. Accéder aux capteurs et stocker les mesures

1.1/ Lire les valeurs des capteurs (lumière, humidité, température)

On utilisera la structure suivant pour stocker les données :

```
struct DATA {
float temperature ; // temperature °C
float humidity ;// humidity %
float light ;//ambient light lux
float supplyVoltage ;// Volt
uint32_t timeStamp ; // millis at time of reading
} ;
```

Lire périodiquement les valeurs des capteurs et remplir les données courante.
(inspirez vous de l'exemple de l'IDE Arduino "02.Digital→ BlinkWithoutDelay" pour faire une temporisation non bloquante).
La période est stockée dans le champ `sensorsPeriod` de la structure de configuration. Pour les essais on prendra une valeur de 30s. Dans l'application finale la période serait beaucoup plus grande (5 min par exemple).

1.2/ Utiliser un buffer circulaire pour sauver les N dernières valeurs

On souhaite conserver en RAM un historique des dernières valeurs mesurées. Une structure de données adaptée est un "buffer circulaire" pour implémenter un comportement de type FIFO.
- Installer et étudier la libraire CircularBuffer : https://github.com/rlogiacco/
Comment déclarer un buffer pour stocker les données de luminosité ? , comment écrire une donnée ? Comment lire les données de la plus ancienne à la plus récente ?

Avec cette librairie la taille du buffer est une constante définie à la compilation, on prendra :

```
#define HISTORY_SIZE 10 // number of values in the circular buffer
```

-Stocker les dernières valeurs des capteurs dans une fifo. Vérifier le bon fonctionnement en affichant le buffer à chaque insertion.

2. Un premier serveur web

- Modifier l'exemple `ESP8266WebServer->HelloServer` pour mettre les paramètres du Wifi de la salle ou de votre téléphone en mode partage de connexion. Charger le et vérifier son fonctionnement.

Remarques :
1. Les paramètres du wifi sont écrits ici "en dur" dans le programme (dans la structure de configuration). C'est acceptable pour un tp d'initiation mais pas pour une application commerciale. Pour résoudre le problème de la connexion initiale, l'Esp démarre d'abord son propre réseau wifi (mode acces point) avec un portail captif. Une page permet alors de rentrer les paramètres de son réseau wifi. L'Esp sauve cette configuration et redémarre en se connectant au résau wifi configuré (mode STA). Une librairie très complète pour gérer ce type de problème est WiFiManager : https://github.com/tzapu/WiFiManager
Cette méthode ne sera pas traitée dans ce tp. Cependant nous verrons au paragraphe 7 comment modifier les paramètres Wifi par la liaison série.

2. Il n'y aucune autre sécurité que la sécurité du réseau wifi. Toute personne connectée au réseau aura accès au serveur de l'Esp. C'est parfaitement acceptable pour une application domestique locale. Si on veut connecter directement l'Esp sur l'internet il est préférable de passer par un reverse-proxy.

- Adapter l'exemple pour l'inclure dans votre programme complet (dans les fichiers `wifi.h/wifi.cpp` pour la partie connexion et dans les fichier `webserver.h / webserver.cpp` pour la partie serveur.

- Modifier le nom DNS local[2] pour qu'il soit de la forme **geiiXXYY.local** où XXYY sont les deux octets poids faibles de l'adresse MAC de votre ESP.
(utiliser la méthode `macAddress()` de la classe Wifi. Attention le paramètre de la méthode `begin()` de la classe MDNS doit être une chaîne c (char *) et pas une *String*. Utiliser la méthode `c_str()` pour transformer une *String Arduino* en chaîne c).

-Créer une page d'accueil simple pour afficher l'état de la led bleue (LED_ANT) et du relais.

- Ajouter la possibilité de commander la led bleue LED_ANT et le relais par une requête de type GET de la forme : **Ip.De.Votre.Esp/execute?relay=0&led=**1
Le serveur exécute la commande et retourne une page simple indiquant l'action effectuée et le nouvel état de la led et du relais.

3. Une page HTML qui affiche les valeurs courantes des capteurs

En vous inspirant de l'exemple `ESP8266WebServer->AdvancedWebServer`, ajouter une fonction qui génère une page HTML affichant les valeurs de température, humidité, luminosité et tension d'alimentation. (On pourra supprimer le graphique en svg pour l'instant)
Dans l'exemple l'html est écrit dans une chaîne de caractères, ce qui oblige à "échapper" (mettre un \ devant) les guillemets et les fins de ligne. On aura avantage à utiliser les "raw string" c++ qui permettent d'insérer le html sans modification :

```
const char INDEX_HTML[] [PROGMEM = R"=====(
          <html>   "mettre le code html tel quel"
          </html>  )=====";
```

2 On utilise ici le protocole **mDNS** qui permet la résolution des adresses sur un petit réseau dépourvu de DNS local.

4. Une page (statique) en Flash

La page précédente était stockée dans la RAM de l'ESP. Il est ainsi facile d'en modifier le contenu mais on va vite être à court de RAM avec ce principe. D'autre part la mise au point est difficile car le code html doit être intégré directement dans le code c++.
L'ESP partage sa mémoire Flash en deux : une partie pour le code et une autre partie pour un système de fichier appelé SPIFFS (SPI Flash File System). Avec 4Mo de flash, on a le choix entre plusieurs répartitions code/SPIFFS : 4Mo/0Mo, 3Mo/1Mo, 2Mo/2Mo ou 1Mo/3Mo.
1Mo de code est souvent suffisant pour la plupart des applications.

Procédure de chargement en SPIFFS :
Même si vous utilisez un autre IDE pour développer votre projet, le plus simple est d'utiliser l'IDE arduino pour charger des fichiers. Il faut d'abord installer une commande additionnelle à l'IDE arduino à télécharger à https://github.com/esp8266/arduino-esp8266fs-plugin/releases
La procédure d'installation est décrite à https://github.com/esp8266/arduino-esp8266fs-plugin
Créer un sketch arduino quelconque et créer un répertoire nommé **data** dans le répertoire de ce sketch. Les fichiers présent dans ce répertoire seront chargés dans la mémoire Flash de l'ESP par la commande "**Sketch data upload**" de l'IDE arduino (attention, la fenêtre "moniteur série" doit être fermée pendant le téléchargement).

- Étudier attentivement l'exemple "A-SPIFFS_File_server" du "ESP8266-beginer-guide" (https://github.com/tttapa/ESP8266) et le document "ESP8266 arduino Server Web SPIFFS et WebSocket" partie 1.

- Ajouter à votre programme la possibilité d'accéder à un fichier html statique stocké en SPIFFS.

- Ajouter la possibilité de servir correctement des images au format jpeg en ajoutant la prise en compte du type MIME correspondant pour les extensions jpg , JPG , jpeg et JPEG.

5. Stocker les valeurs des capteurs en SPIFFS

Il s'agit de lire périodiquement la valeur des capteurs et de stocker ces informations au format CSV dans la mémoire Flash de l'ESP.

Après chaque lecture de la valeurs des capteurs, on va ouvrir un fichier *"sensors.csv"* en mode "a", stocker une ligne de la forme

```
"ns,light,temperature,humidity\r\n"
```

et refermer le fichier. (`ns` est le nombre de secondes depuis le reset).
Exemple de ligne : "30,238.00,22.25,66.80\r\n"
Le mode "a" permet d'écrire après la dernière donnée présente dans le fichier.
Il est nécessaire de refermer le fichier après chaque écriture pour que l'écriture en Flash soit effective. On peut également ouvrir le fichier au départ (dans le `setup`) et utiliser la méthode `flush()` pour forcer l'écriture. Si vous souhaitez effacer les données au lancement du programme, il suffit d'ouvrir le fichier en mode "w".

- Implémenter l'écriture en SPIFFS et vérifier que vous pouvez télécharger le fichier par le serveur web.

Remarques : Dans une application réelle, il faudrait limiter la taille du fichier et prévoir une procédure supprimer les données trop anciennes. On pourrait par exemple changer de fichier chaque jour et supprimer les fichiers au bout de 15 jours.

6. Des graphiques montrant les valeurs des capteurs

On souhaite afficher un graphique montrant les valeurs stockées dans le fichier csv.

Il existe plusieurs manières pour générer un graphique dans un système embarqué. Certaines méthodes sont très puissantes mais impliquent une connexion internet (google chart,…), d'autres utilisent des bibliothèques non libres. Nous allons utiliser une bibliothèques java script open source qui ne nécessitent pas de connexion internet : **chart.js** (https://www.chartjs.org/). La génération du graphique est alors reportée côté client (dans le navigateur). L'objet connecté ne faisant que transmettre des données : une page html, des scripts en langage javascript et le fichier *csv* à afficher.

Étudier attentivement l'exemple fournit :

`chartCSVExample.html` et `ExampleData.csv`

Cette page html contient un script javascript (JS) qui exécute (dans le navigateur) au chargement de la page la fonction `window.onload()` . Celle ci demande le fichier csv (par une requête jquery) et génère le graphique.

- Charger l'exemple en SPIFFS et vérifier le fonctionnement.

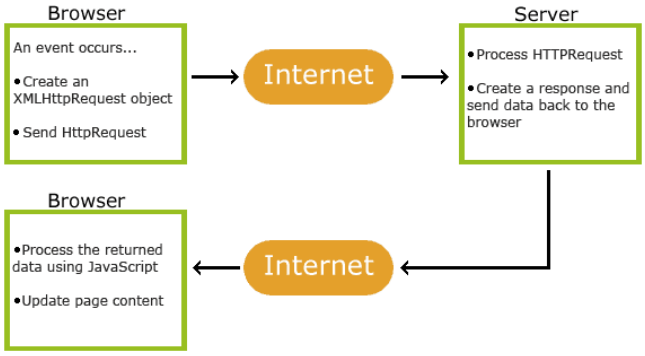
- En vous inspirant de l'exemple, créer une page qui affiche les valeurs des capteurs sous forme graphique.

7. Des pages dynamiques !

Jusqu'à présent les pages étaient soit "statiques" stockées en Flash (code ou SPIFFS), soit générées en RAM sur demande par le serveur. La seule manière de rafraîchir la page est de la recharger en totalité. C'est acceptable pour une petite page mais ce n'est pas très fluide ni très réactif. Nous allons voir deux méthodes qui permettent un rafraîchissement très fluide en ne transmettant uniquement les informations à modifier. Selon les applications, l'une ou l'autre de ces méthodes est mieux adaptée.

7.1 AJAX : Asynchronous JavaScript and XML

AJAX est une technique qui permet de rafraîchir rapidement un page web en échangeant de petites quantités de données au format XML entre le client et le serveur par l'intermédiaire de requête HTTP. Voir : https://www.w3schools.com/asp/asp_ajax_intro.asp



Le navigateur charge la page HTML contenant un code javascript qui envoie une requête XMLHttp. Le serveur traite cette requête (lit un capteur, change un sortie,…) et crée une

réponse qu'il retourne au navigateur. Dans le navigateur, le script JS met à jour la page en fonction de la réponse.

- Étudier l'exemple : `ajaxExemple.html`

Ce fichier contient une page html simple et un code JS qui exécute périodiquement la fonction `getUptime()` ci- dessous :

```
function getUptime() {
        var xhttp = new XMLHttpRequest();
        xhttp.onreadystatechange = function() {
                if (this.readyState == 4 && this.status == 200) {
                // l'élement uptime de la page prend la valeur de la réponse
                document.getElementById("uptime").innerHTML = this.responseText;
                }
        };
        xhttp.open("GET", "getUptime", true);
        xhttp.send();
}
```

La requête html de type GET, `getUptime` est envoyé au serveur. Lorsque la réponse revient la fonction enregistrée dans le champ `onreadystatechange` de la variable `xhttp` s'exécute. Celle-ci récupère l'élément `uptime` de la page et y place le texte reçu `responseText.`

Dans le serveur, il faut enregistrer une fonction `handleUptime` qui répond à la requête `getUptime :`

```
server.on("/getUptime", handleUptime);
```

La fonction lit le uptime (millis) , le transforme en texte et l'envoie en mode texte au serveur :

```
void handleUptime() {
        String uptime = String(millis()/1000) ;
        server.send(200, "text/plain", uptime);
}
```

- Charger l'exemple en SPIFFS, ajouter le code qui répond à la requête `uptime` et vérifier le fonctionnement.
- Ajouter le code qui répond à la requête de commande de la LED. (Inspirez vous de code de commande de la LED du paragraphe 2).

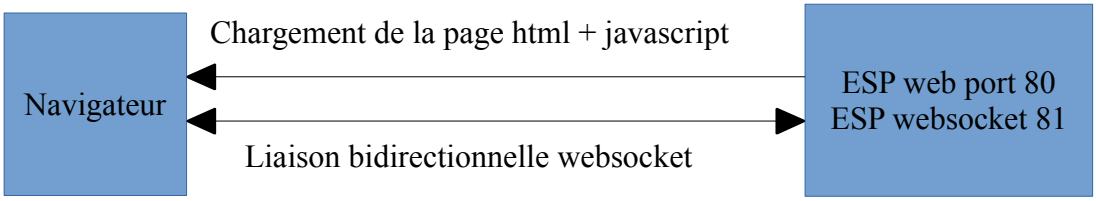
-Écrire une page dynamique affichant l'état des capteurs et permettant de commander la LED et le relais avec deux boutons.

7.2 Étude de la librairie arduinoWebSockets

Nous allons mettre en place un serveur *webSocket* de manière à pouvoir demander des information et passer des commandes sans avoir à recharger la page entière. Comme pour la méthode précédente, le programme javascript embarqué dans la page demandera périodiquement des mises à jour et pourra modifier uniquement les éléments nécessaires de la page. Ici contrairement à la méthode précédente, les différentes modifications pourront être répercutées automatiquement aux différents clients connectés (broadcast).

- Installer la librairie https://github.com/Links2004/arduinoWebSockets
- Charger et étudier l'exemple : https://gist.github.com/bbx10/667e3d4f5f2c0831d00b
- Modifier le pour contrôler la Led RGB de l'ESP.
- Vérifier que si vous chargez la page dans un ou plusieurs autres navigateurs (éventuellement sur des machines différentes), l'état de la led est automatiquement mis à jour.

Principe de fonctionnement :



Le navigateur obtient la page qui contient le html et un code javascript par une requête sur le port 80. Une fonction javascript s'exécute (sur le navigateur) au chargement de la page pour établir une connexion avec le serveur websocket sur le port 81. Le navigateur peut alors communiquer avec le serveur websocket par des fonctions javascript et modifier éventuellement le contenu de la page. Le serveur websocket peut envoyer des informations à tous les clients connectés pour effectuer les mises à jour de l'état des capteurs/actionneurs.
On peut ainsi avoir un affichage fluide et très réactif.

Un mot sur la sécurité : Ici la sécurité du système repose sur l'accès au réseau wifi. Dès qu'on a accès au réseau wifi, on a accès aux différentes pages de l'ESP et au serveur websocket. Le serveur websocket ne vérifie pas l'origine de la connexion. Il est donc illusoire de mettre en place un accès par mot de passe à la page web. Ce fonctionnement est tout à fait acceptable pour une utilisation domotique privée dans lequel tout les membres d'une famille ont accès aux différents objets connectés dès lors qu'ils ont accès au réseau wifi. Il ne faut surtout pas que l'ESP soit directement visible depuis l'internet (par une redirection de port sur la box par exemple) car dans ce cas n'importe qui peut y avoir accès. Si on veut y accéder depuis l'extérieur de son réseau on peut utiliser le support SSL de l'ESP mais celui ci est assez limité. La solution la plus utilisée semble être de mettre en place un reverse proxy.

7.3 Mise au point du serveur webSocket

Pour mettre au point le serveur websocket, nous allons utiliser l'utilitaire **`wscat`** qui permet de se connecter en ligne de commande à un serveur. On pourra ainsi envoyer des données au serveur comme le fera plus tard notre programme javascript et vérifier le résultat.

- Installer l'outil **`wscat`** (https://www.npmjs.com/package/wscat)
- Vérifier que vous pouvez vous connecter par :

```
$wscat -c ws://ip_du_serveur:port
```

-Contrôler la led de l'exemple en ligne de commande

Nous allons mettre en place un protocole très simple pour communiquer avec le serveur.

Commandes "ESP" ("&?" si non reconnue ou paramètre erronée):

Fonction	**Commande**	**Réponse**	**Commentaire**
Lire upTime	**!U**	**&U;01:54:34;ok**	Temps écoulé depuis le reset
Relais	**!R1 / !R0**	**&R;1;ok**	Change état du relais ESP
Led ANT	**!L1 / !L0**	**&L;1;ok**	Change état de la LED ANT
Couleur led RGB	**!C064;122;010**	**&C;ok**	Change couleur (ordre R,G,B)
Lire capteurs	**!S ;**	**&S;22.4;58.2;155;ok**	Température,humidité,lux

- Ajouter un serveur websocket sur le port 81.

- Implémenter les différentes commandes.

Les réponses sont envoyées à tous les clients connectés (*broadcast*), pour qu'ils puissent mettre à jour l'état du système (sauf pour les commandes de lecture du uptime et des capteurs).

7.4 Un page en spiffs qui communique avec le serveur websocket

En vous inspirant de l'exemple, écrire une page html et un code javascript qui informe de l'état des capteurs et qui permet de commander les différents actionneurs.
Cette page sera stockée en flash dans le SPIFFS.

Nb : pour la mise au point, il n'est pas nécessaire de charger la page en flash. Lors de la création de l'objet WebSocket en javascript, il suffit de remplacer l'adresse de connexion par l'adresse de l'ESP pour que la communication s'établisse même si la page se trouve sur votre machine de développement :
remplacer :

```
websock = new WebSocket('ws://' + window.location.hostname + ':81/');
```

par :

```
websock = new WebSocket('ws://' + '192.168.0.44' + ':81/');
```

8. Configuration par le port série

On peut modifier la configuration de l'objet (paramètre Wifi, …) par des commandes sur le port série.
Étudier l'exemple "Communication->Serial Event". On y trouve une méthode pour récupérer une commande entrée par le port série dans une variable de type String. Ensuite il est possible d'analyser cette variable pour exécuter les commandes reçues. Attention dans l'ESP8266, la fonction *serialEvent()* n'est pas appelée automatiquement. Il faut l'appeler régulièrement dans la fonction loop().

Au démarrage, L'ESP attend une seconde puis fait clignoter la LED ANT et attend ensuite pendant cinq secondes. Si pendant ces cinq secondes, on appuie sur le bouton FLASH[3], on accepte les commandes sur le port série.
La sortie de ce mode particulier se fait par un nouvel appui sur le bouton ou par une commande "exit".
Pendant la configuration il n'est pas nécessaire d'effectuer les autres fonctionnalités.
A la sortie de ce mode la configuration est sauvegardée en mémoire non volatile[4] (EEPROM) et la LED ANT s'éteint et on fait un reset. Au prochain démarrage la configuration est lue à partir de l'EEPROM.

Implémenter les commandes suivantes :

Fonction	**Commande**	**Réponse**	**Commentaire**
ssid	**!ssid=gewifitp1**	**&ok**	
mot de passe	**!pass=geiigeii**	**&ok**	
serveur mqtt	**!mqtt=192.168.0.99**	**&ok**	
hostname	**!host=username**	**&ok**	nom mdns (à la place adr mac)
période capteurs	**!period=2000**	**&ok**	valeur en millisecondes
fin de configuration	**!exit**	**Affiche la configuration**	sauvegarde EEPROM et reset
non reconnue		**&?**	aucune action

Chaque commande doit être terminée par le caractère newline ('\n' code 0x0A).
Si vous avez ajouté d'autres champ dans la structure de configuration, il faut implémenter les commandes permettant de les modifier.
Comme les valeurs sont stockées en EEPROM en cas de problème un reset n'est pas efficace. Il faut donc prévoir la possibilité de retour à un mode par défaut. Par exemple un appui long sur le bouton FLASH ou par une commande "default".

3 Attention, l'appui sur le bouton FLASH dès le reset fait entrer l'ESP dans le mode "bootloader". Il faut donc attendre l'allumage de la LED avant d'appuyer sur le bouton.

4 Utiliser les méthodes `put` et `get` de la classe EEPROM pour sauver et relire facilement une structure. (Voir le fichier source EEPROM.h).

9. Connexion à un serveur MQTT

9.1 Installation

Dans un environnement linux, il suffit d'utiliser le gestionnaire de paquet de la distribution. Par exemple pour Ubuntu :

```
$ sudo apt install mosquitto mosquitto-clients
```

Tester l'installation :
- Dans un terminal s'abonner à un topic *test* :

```
$ mosquitto_sub -v -t test
```

-Dans un autre terminal publier sur le topic test :

```
$ mosquitto_pub -m hello – t test
```

- Vérifier que l'accès est possible depuis une autre machine en rajoutant l'option :

```
-h ip_du_serveur
```

- Vérifier qu'en publiant un message avec le mode *retain* (option **-r**), le message est mémorisé.

Utilisation des "wilcards" # et + :

- # permet d'écouter toute la sous-arborescence qui suit. Par exemple

```
$ mosquitto_sub -t /esp2E08/# -v
```

donne accès à tous les topics commençant par /esp2E08/

- + permet de remplacer un niveau de la sous-arborescence. Par exemple

```
$ mosquitto_sub -t /+/temperature -v
```

donne accès à toutes les températures (publiées après un seul niveau).

Sur un téléphone *Android* télécharger l'application "MQTT Dash".
- Vérifier que vous savez publier et recevoir des messages.

Installer la libraire PubSubClient :

```
https://github.com/knolleary/pubsubclient
```

Cette librairie permet de publier et de s'abonner à un serveur MQTT. La taille totale des messages, header+topic+message, est de 128 octets par défaut. Comme le header est sur 5 octets maximum, il reste au minimum 123 octets pour le topic **et** le message.

9.2 Utiliser et analyser l'exemple fourni

Le programme ci dessous se connecte au réseau Wifi et au serveur MQTT spécifié et s'abonne au topic "led". Toutes les 5 secondes il publie un message sur le topic test/ESP01/valeur. On peut commander la led en envoyant le caractère 1 ou 0 sur le topic led.
C'est une excellent base de départ. Vous devez faire fonctionner et bien comprendre ce programme avant d'attaquer la suite.

```
#include <ESP8266WiFi.h>
#include <ESP8266mDNS.h>
#include <EEPROM.h>
#include <PubSubClient.h>

#define MQTT_PORT 1883
#define HOST_NAME "ESP01"
```

```
#define LED_ANT D4  // Led antenne sur ESP12

// nom (mDNS) ou ip de la machine ayant le broker
const char* mqtt_server = "192.168.0.9";

// objet pour la connexion
WiFiClient espClient;
// connexion MQTT
PubSubClient client(espClient);
// état de la led
bool led_state = 0 ;

// cette fonction s'exécute lorsqu'un message MQTT est reçu :
// le topic est toujours une chaîne de caractères (terminée par '\0')
// la payload pas forcément, la longueur de la payload est dans length
void callback(char* topic, char* payload, unsigned int length) {
  Serial.print("Topic [");
  Serial.print(topic);
  Serial.print("] ");
  Serial.print("Payload : ");
  // affichage du payload
  for (int i = 0; i < length; i++) {
    Serial.print((char)payload[i]);
  }
  Serial.println();

  // la suite dépend du topic
payload[length] = '\0' ; //on s'assure que la payload est bien une chaîne C
String sPayload(payload); // passe la payload en String arduino
String sTopic(topic); // passe le topic en String arduino
// On peut donc travailler au choix avec les chaînes C payload et topic
// ou bien avec les String arduino sPayload et sTopic
  if ( sTopic == String("led") ) {

    // le caractère '1' est-il le premier du payload ?
    if ((char)payload[0] == '1') {
      // oui led = on
      digitalWrite(BUILTIN_LED, LOW);
      led_state = 1 ;
    } else  if ((char)payload[0] == '0') {
      // non led = off
      digitalWrite(BUILTIN_LED, HIGH);
      led_state = 0 ;
    }
    return ;
  }
  // traiter les autres topic
  if ( sTopic == String("autre_topic") ) {
    … ;
    return ;
  }
}
// connexion ou reconnexion au broker
// ici bloquant si connexion impossible (il vaudrait mieux sortir au bout
d'un timeout)
// dès qu'on est connecté, on s'abonne au(x) topic(s) souhaité(s)
void reconnect() {
  // Connecté au broker ?
  while (!client.connected()) {
    // non. On se connecte.
    if (!client.connect(mqtt_server)) {
      Serial.print("Erreur connexion MQTT, rc=");
```

```
      Serial.println(client.state());
      delay(5000);
      continue;
    }
    Serial.println("Connexion serveur MQTT ok");
    // connecté.
    // on s'abonne au topic "led"
    client.subscribe("led");

  }
}

void setup() {
  // configuration sortie
  pinMode(BUILTIN_LED, OUTPUT);
  pinMode(LED_ANT, OUTPUT);
  digitalWrite(BUILTIN_LED,1); // logique inverse sur ESP12
  digitalWrite(LED_ANT,1);

  Serial.begin(115200); // configuration moniteur série

  setup_wifi(); // configuration wifi

  client.setServer(mqtt_server, MQTT_PORT); // configuration broker

  client.setCallback(callback);  // configuration callback
}

void loop() {

  char msg[32]; //  array pour le message
  char topic[64];// array pour le topic
  static uint8_t val = 0;
  // Sommes-nous connecté ?
  if (!client.connected()) {
        reconnect();// non. Connexion
        // après chaque reconnexion on publie l'état courant de la led
       // pour que toutes les clients MQTT soient à jour.
       client.publish("led", led_state?"1":"0" );
  }
  // gestion MQTT
  client.loop();

  // temporisation
  static long lastMsg = 0;
  long now = millis();
  if (now - lastMsg >= 5000) {
    lastMsg = now;
    val++;
    // construction du message
    snprintf(msg, 32, "hello world #%d", val);
    // construction topic
    snprintf(topic, 64, "test/%s/valeur", HOST_NAME );
    // publication message sur topic
    Serial.printf("publishing topic : %s , message : %s\n", topic, msg);
    client.publish(topic, msg);
  }
}
```

Attention : Les fonctions *publish* et *subscribe* n'acceptent que les chaînes C. Si on dispose d'une String arduino il faut la convertir en chaîne C par la méthode `c_str()` pour la passer à *publish* ou *subscribe*.
Exemple :

```
String message = "ON" ;
String topic = "/home/light/state" ;
client.publish( topic.c_str(), msg.c_str() );
```

9.3 Connaître l'état des capteurs en se connectant au serveur MQTT

L'ESP publie périodiquement l'état de chaque capteur (toutes les 30 secondes par exemple, ajouter un champ dans la structure de configuration).

Schéma des topics capteurs (XXYY sont les deux derniers octets de l'adresse MAC de l'ESP)

Topic	Format des messages
/espXXYY/temperature	21.4
/espXXYY/humdity	62.2
/espXXYY/light	243

9.4 Commande d'actionneur en passant par un serveur MQTT

On souhaite commander le relais, la LED_ANT et la couleur de la Led RGB par des message MQTT. Quand on commande un actionneur il est important de s'assurer que la commande a bien été exécutée. Pour cela on va créer deux topics par actionneur :

un topic ***set*** qui permet de commander l'actionneur,
un topic ***state*** qui permet de connaître l'état courant de l'actionneur.

L'ESP va s'abonner au topic ***set*** de chaque actionneur. S'il reçoit une commande valide sur ce topic, il exécute cette commande et publie le nouvel état de l'actionneur sur le topic ***state*** correspondant. En l'absence de commande, il publie périodiquement l'état de l'actionneur sur la topic ***state***.

Schéma des topics actionneurs (XXYY sont les deux derniers octets de l'adresse MAC de l'ESP)

Topic	Format des messages
/espXXYY/led/set	0 , 1
/espXXYY/led/state	0 , 1
/espXXYY/relay/set	0 , 1
/espXXYY/relay/state	0 , 1
/espXXYY/color/red/set	rrr taux de r en décimal
/espXXYY/color/red/state	rrr
/espXXYY/color/green/set	ggg taux de g en décimal
/espXXYY/color/green/state	ggg
/espXXYY/color/blue/set	bbb taux de b en décimal

/espXXYY/color/blue/state	bbb
/espXXYY/color/set	Modifie la couleur avec le format html en décimal. Color html héxa: 0xRRGGBB
/espXXYY/color/state	Retourne la couleur html dec

Les topics séparés pour fixer chaque couleur sont plus faciles à programmer.
Si vous utilisez l'application Android **MQTT dash**, la couleur html peut être envoyée par l'objet color en choisissant le format "Numeric string".