



Programmation D'un serveur WEB (module ESP32-ESP8266) Sous JAVASCIPT et le modèle AJAX





Mise en place d'un site WEB par l'utilisation de code Javascript et le modèle AJAX

1) présentation du modèle AJAX

AJAX (Asynchronous JavaScript + XML) n'est pas une technologie en soi, mais un terme désignant une « nouvelle » approche utilisant un ensemble de technologies existantes, dont : HTML ou XHTML, les feuilles de styles CSS, JavaScript, le modèle objet de document (DOM), XML, XSLT, et l'objet XMLHttpRequest. Lorsque ces technologies sont combinées dans le modèle AJAX, les applications Web sont capables de réaliser des mises à jour rapides et incrémentielles de l'interface utilisateur sans devoir recharger la page entière du navigateur. Les applications fonctionnent plus rapidement et sont plus réactives aux actions de l'utilisateur.

AJAX est un raccourci pour *Asynchronous JavaScript + XML* (JavaScript asynchrone plus XML). En simplifiant, il s'agit d'employer l'objet non standard XMLHttpRequest pour communiquer avec des scripts situés sur le serveur. L'objet permet d'échanger des informations sous différents formats (dont XML, HTML ou texte), mais son principal attrait est sa nature « asynchrone » qui implique qu'il peut communiquer avec le serveur, échanger des données et mettre à jour sans avoir à sans recharger la page.

Les deux fonctionnalités combinées vous permettent de :

- faire des requêtes vers le serveur sans avoir à recharger la page ;
- analyser et travailler avec des documents XML.

Pour faire une requête HTTP vers le serveur à l'aide de JavaScript, il faut disposer d'une instance d'une classe fournissant cette fonctionnalité. C'est ici que la classe XMLHttpRequest intervient. Une telle classe a d'abord été introduite dans Internet Explorer sous la forme d'un objet ActiveX appelé XMLHTTP. Par la suite, Mozilla, Safari et d'autres navigateurs ont suivi en implémentant une classe XMLHttpRequest qui fournit les mêmes méthodes et propriétés que l'objet ActiveX original de Microsoft. Dans l'intervalle, Microsoft a également implémenté XMLHttpRequest.

Par conséquent, pour créer une instance (un objet) de la classe désirée fonctionnant sur plusieurs navigateurs, vous pouvez utiliser :

```
// ancien code de compatibilité, aujourd'hui inutile
if (window.XMLHttpRequest) { // Mozilla, Safari, ...
    httpRequest = new XMLHttpRequest();
}
else if (window.ActiveXObject) { // IE
    httpRequest = new ActiveXObject("Microsoft.XMLHTTP");
```

La chose suivante à faire est de décider ce que vous ferez après avoir reçu la réponse du serveur.

Notez qu'il n'y a ni parenthèses après le nom de la fonction, ni paramètres fournis, car vous ne faites qu'assigner une référence à la fonction sans l'appeler effectivement.

xhttp.onreadystatechange = function

Par ailleurs, au lieu de donner un nom de fonction, vous pouvez utiliser la technique JavaScript de définition de fonctions au vol (ce qu'on appelle une fonction anonyme), et définir à cet endroit les actions à effectuer sur la réponse :

```
xhttp.onreadystatechange = function() {
if (this.readyState == 4 && this.status == 200)
{
 // instructions de traitement de la réponse
}
```

Ensuite, après avoir déclaré ce qui se produit lorsque la réponse est reçue, il s'agit de lancer effectivement la requête. Il faut pour cela appeler les méthodes open() et send() de la classe de requête HTTP, comme ceci :

```
httpRequest.open('GET', 'http://www.example.org/some.file', true);
httpRequest.send();
```

- Le premier paramètre de l'appel à open() est la méthode de requête HTTP — GET, POST, HEAD ou toute autre méthode que vous voulez utiliser et est gérée par le serveur. Laissez le nom de la méthode en majuscules comme spécifié par la norme HTTP ; autrement certains navigateurs (comme Firefox) peuvent ne pas traiter la requête.
- Le second paramètre est l'URL de la page que vous demandez. Pour des raisons de sécurité, il n'est pas possible d'appeler des pages se situant sur un autre domaine. Veillez à utiliser le nom de domaine exact sur toutes vos pages ou vous obtiendrez une erreur « permission refusée » à l'appel d'open(). Une erreur courante est de charger le site via domaine.tld, mais d'essayer d'appeler des pages avec www.domain.tld.
- Le troisième paramètre précise si la requête est asynchrone. Si mis à TRUE, l'exécution de la fonction JavaScript se poursuivra en attendant l'arrivée de la réponse du serveur. C'est le A d'AJAX

2) Implantation sur un serveur WEB ESP de requêtes asynchrones statiques

Le serveur WEB sera implanté sur un module ESP et mettra à disposition une page WEB, des images, des applets, et des scripts Javascript.

Les différents éléments (page WEB, images, applets, code Javascript) seront placés sur le système de fichier du module ESP. En effet le module ESP dispose d'une mémoire flash SPI que l'on peut exploiter grâce à la technologie SPIFFS et qui permettra d'accueillir tous ces éléments :

La déclaration des fichiers d'entêtes suivants sera nécessaire :

```
#include "WiFi.h"
#include "ESPAsyncWebServer.h"
#include "SPIFFS.h"
#include "FS.h"
```

La première chose à créer est notre serveur WEB issue de la classe AsyncWebServer pour des appels asynchrones :

```
AsyncWebServer server(80);// port 80 pour la connexion des clients :
```

Dans la fonction setup() on créera un point d'accès WIFI ainsi que l'objet permettant de générer notre système de fichier SPIFFS

```
if(!SPIFFS.begin(true))
{
   Serial.println("An Error has occurred while mounting SPIFFS");
  }
```

L'appel de la méthode on() de l'objet server nous permettra de gérer les différentes requêtes asynchrones du client WEB

Pour des éléments statiques comme des images, des pages Web statiques le principe reste simple :

Pour la requête HTTP_GET de la première page du site on retrouve le code suivant :

```
server.on("/", HTTP_GET, [](AsyncWebServerRequest *request){

   request->send(SPIFFS, "/index.html", "text/html");

  });
```

La méthode send de l'objet request nous permettra de renvoyer la réponse vers le client WEB.

- Le premier paramètre indique que l'élément à renvoyer se situe sur la mémoire flash du module ESP.
- Le deuxième paramètre indique le nom de cet élément
- Le troisième paramètre indique la nature de l'élément à renvoyer du texte formaté HTML ici.

Le client envoie la requête http_GET

```
Hypertext Transfer Protocol
  GET / HTTP/1.1\r\n
  Accept: text/html, application/xhtml+xml, image/jxr, */*\r\n
  Accept-Language: fr-FR\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko\r\n
  Accept-Encoding: gzip, deflate\r\n
  Host: 192.168.4.1\r\n
  Connection: Keep-Alive\r\n
  \r\n
```

Le serveur ESP répond et fournit par la suite le code de la page index.html:

```
Hypertext Transfer Protocol
  HTTP/1.1 200 OK\r\n
  Content-Length: 7465\r\n
  Content-Type: text/html\r\n
  Content-Disposition: inline; filename="index.html"\r\n
  Connection: close\r\n
  Accept-Ranges: none\r\n
  \r\n
```

Dans le cas d'une image on obtient :

```
server.on("/CO2-icon.jpg", HTTP_GET, [](AsyncWebServerRequest *request){

   request->send(SPIFFS, "/CO2-icon.jpg", "image/jpeg");

 });
```

Le client envoie la requête http_GET

```
Hypertext Transfer Protocol
  GET /CO2-icon.jpg HTTP/1.1\r\n
  Accept: image/png, image/svg+xml, image/jxr, image/*;q=0.8, */*;q=0.5\r\n
  Referer: http://192.168.4.1/\r\n
  Accept-Language: fr-FR\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko\r\n
  Accept-Encoding: gzip, deflate\r\n
  Host: 192.168.4.1\r\n
  Connection: Keep-Alive\r\n
  \r\n
```

Le serveur ESP répond et fournit par la suite le contenu binaire de l'image:

```
Hypertext Transfer Protocol
  > HTTP/1.1 200 OK\r\n
  > Content-Length: 80418\r\n
    Content-Type: image/jpeg\r\n
    Content-Disposition: inline; filename="CO2-icon.jpg"\r\n
    Connection: close\r\n
    Accept-Ranges: none\r\n
    \r\n
    [HTTP response 1/1]
    [Time since request: 0.514450000 seconds]
    [Request in frame: 131]
    File Data: 80418 bytes
> JPEG File Interchange Format
```

Dans le cas d'une APPLET JAVA

```
server.on("/web.jar", HTTP_GET, [](AsyncWebServerRequest *request){

   request->send(SPIFFS, "/web.jar", "application/x-java-applet");

 });
```

Le fichier web.jar correspond à notre archive JAVA compressée

Le client envoie la requête http_GET

```
Hypertext Transfer Protocol
  > GET /web.jar HTTP/1.1\r\n
    accept-encoding: pack200-gzip, gzip\r\n
    content-type: application/x-java-archive\r\n
    User-Agent: Mozilla/4.0 (Windows 10 10.0) Java/1.8.0_162\r\n
    Host: 192.168.4.1\r\n
    Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2\r\n
    Connection: keep-alive\r\n
    \r\n
    [Full request URI: http://192.168.4.1/web.jar]
    [HTTP request 1/1]
    [Response in frame: 506]
```

Le serveur ESP répond et fournit par la suite le contenu binaire de l'applet:

```
Hypertext Transfer Protocol
  > GET /web.jar HTTP/1.1\r\n
    accept-encoding: pack200-gzip, gzip\r\n
    content-type: application/x-java-archive\r\n
    User-Agent: Mozilla/4.0 (Windows 10 10.0) Java/1.8.0_162\r\n
    Host: 192.168.4.1\r\n
    Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2\r\n
    Connection: keep-alive\r\n
    \r\n
    [Full request URI: http://192.168.4.1/web.jar]
    [HTTP request 1/1]
    [Response in frame: 506]
```

Dans le cas d'un code javascript

```
server.on("/RGraph.thermometer.js", HTTP_GET, [](AsyncWebServerRequest *request){
   request->send(SPIFFS, "/RGraph.thermometer.js", "text/javascript");
 });
```

Tous ces éléments peuvent être lancé par le code HTML d'une ou de plusieurs Page WEB

Le client envoie la requête http_GET

```
Hypertext Transfer Protocol
  > GET /RGraph.thermometer.js HTTP/1.1\r\n
    Accept: application/javascript, */*;q=0.8\r\n
    Referer: http://192.168.4.1/\r\n
    Accept-Language: fr-FR\r\n
    User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko\r\n
    Accept-Encoding: gzip, deflate\r\n
    Host: 192.168.4.1\r\n
    Connection: Keep-Alive\r\n
    \r\n
    [Full request URI: http://192.168.4.1/RGraph.thermometer.js]
    [HTTP request 1/1]
```

Le serveur ESP répond et fournit par la suite le script:

```
Hypertext Transfer Protocol
  > HTTP/1.1 200 OK\r\n
  > Content-Length: 16553\r\n
    Content-Type: text/javascript\r\n
    Content-Disposition: inline; filename="RGraph.thermometer.js"\r\n
    Connection: close\r\n
    Accept-Ranges: none\r\n
    \r\n
    [HTTP response 1/1]
    [Time since request: 0.244575000 seconds]
    [Request in frame: 124]
    File Data: 16553 bytes
> Line-based text data: text/javascript (65 lines)
```

3) Implantation sur un serveur WEB ESP de requêtes asynchrones dynamiques

Les pages WEB sont dynamiques et le serveur WEB doit fournir les informations qui permettent de mettre à jour ces éléments dynamiques.

Pour que le client WEB puisse interroger de manière synchrone le serveur WEB, on doit écrire en javascript sur la page WEB principale le code suivant :

```
<script>

setInterval(function() {
  getfct1();
  getinput();
  run();
}, 2000);


</script>
```

L'appel de la fonction setInterval permettra de lancer les quatres fonctions (getfct1, getinput,run) toutes les deux secondes.

3.1) Requête GET d'une simple variable

La première fonction getfct1 permet de mettre à jour une variable dynamique sur la page WEB. Cette variable sera baptisée var1. On utilisera en code HTML une balise du type span et l'élément id pour référencer la variable ;

La valeur est de : temperature

Passage d'un élément du type texte vers le client WEB

Dans la page WEB le script AJAX de la fonction getfct1 devient le suivant :

```
function getfct1() {

var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function()
{
if (this.readyState == 4 && this.status == 200) {
document.getElementById("var1").innerHTML = this.responseText;
}
};
xhttp.open("GET", "envoid1", true);
```

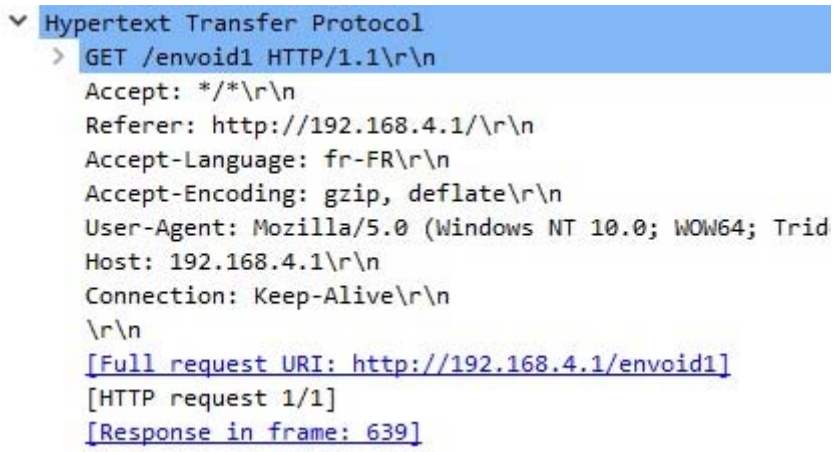
```
xhttp.send();
}
```

Le serveur WEB ESP doit gérer la requête GET précédente. La méthode server.on permettra cette gestion:

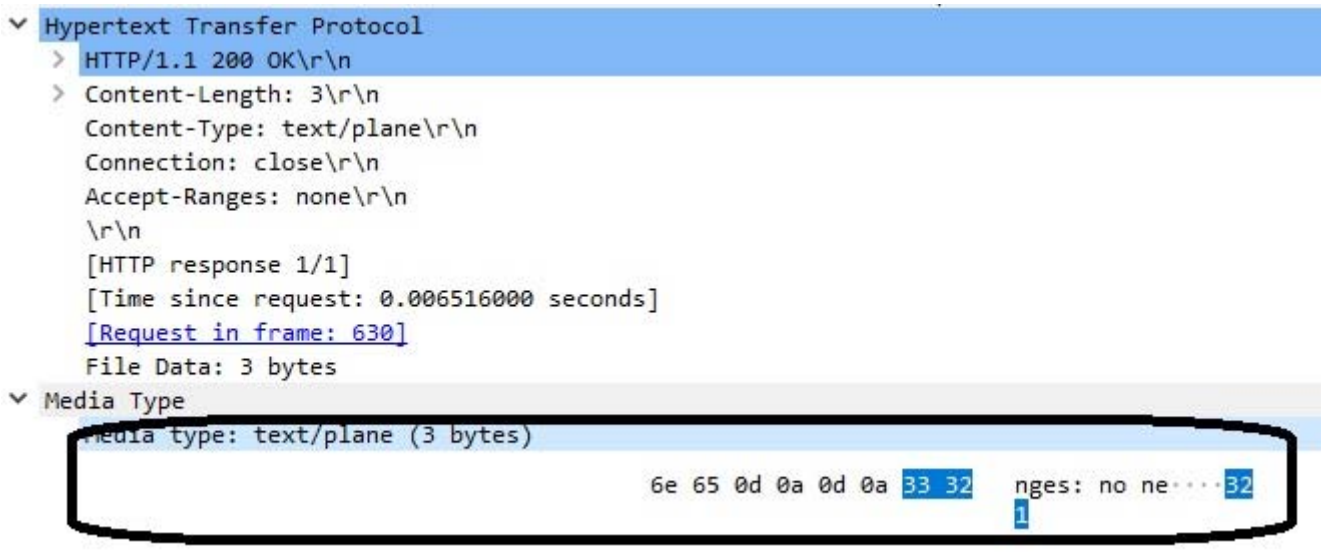
```
server.on("/envoiD1", HTTP_GET, [](AsyncWebServerRequest *request){
   request->send(200, "text/plane", eco2());
 });
```

La fonction send renverra sous forme de chaine de caractère la valeur de var1 qui sera reçue par le client et automatiquement mise à jour sur la page

Le client envoie la requête http_GET



Le serveur ESP répond et fournit par la suite le chiffre sous chaine de caractère qui sera analysée par le script au niveau du client web.



Var 1 peut être associée à une variable dynamique de notre page :

La valeur est :
Ce qui après mise à jour nous donne la valeur 321 dans le champ var1



3.2) Requête GET d'un ensemble de variable

On souhaite transmettre plusieurs variables à la fois vers le client web.

Dans ce cas précis une des solutions possibles est d'utiliser le format xml pour passer des variables entre le serveur et le client web.

Le fichier xml sera formatté de la manière suivante :

```
<?xml version = \"1.0\" ?>
<inputs>
<button1>ON</button1>
<button2>ON</button2>
</inputs>
```

Le script AJAX sera le suivant au niveau de la page web :

```
function getinput() {

var xhttp = new XMLHttpRequest();
 xhttp.onreadystatechange = function()
 {

if (this.readyState == 4 && this.status == 200)
{
document.getElementById("input1").innerHTML =
this.responseXML.getElementsByTagName('button1')[0].childNodes[0].nodeValue;
document.getElementById("input2").innerHTML =
this.responseXML.getElementsByTagName('button2')[0].childNodes[0].nodeValue;
}
};
xhttp.open("GET", "getinput", true);
xhttp.send();
}
```

Du coté du serveur web le code devient :

```
server.on("/getinput",HTTP_GET,  [](AsyncWebServerRequest *request){
```

```
  request->send(200, "text/xml", chaine());
 });
String chaine(void){
 String ch=String("<?xml version = \"1.0\" ?>")+String("<inputs>")+String("<button1>");
  ch=ch+String("ON");
  ch=ch+String("</button1>")+String("<button2>");
  ch=ch+String("ON");
  ch=ch+String("</button2>")+String("</inputs>");
 return ch;
}
```

La fonction chaine retournera le contenu du texte formaté en xml avec deux variables button1 et button2.

Du coté client le script récupérera la valeur de ces deux variables par la méthode getElementsByTagName

Le client envoie la requête http_GET

```
Hypertext Transfer Protocol
  GET /getinput HTTP/1.1\r\n
  Accept: */*\r\n
  Referer: http://192.168.4.1/\r\n
  Accept-Language: fr-FR\r\n
  Accept-Encoding: gzip, deflate\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko\r\n
  Host: 192.168.4.1\r\n
  Connection: Keep-Alive\r\n
  \r\n
  [Full request URI: http://192.168.4.1/getinput]
  [HTTP request 1/1]
  [Response in frame: 647]
```

Le serveur ESP répond et fournit par la suite le texte xml qui sera analysée par le script au niveau du client web.

```
Hypertext Transfer Protocol
  HTTP/1.1 200 OK\r\n
  Content-Length: 83\r\n
  Content-Type: text/xml\r\n
  Connection: close\r\n
  Accept-Ranges: none\r\n
  \r\n
  [HTTP response 1/1]
  [Time since request: 0.014047000 seconds]
  [Request in frame: 635]
  File Data: 83 bytes
eXtensible Markup Language
  <?xml
     version = "1.0"
     ?>
  <inputs>
    <button1>
       ON
       </button1>
    <button2>
       ON
       </button2>
    </inputs>
```

Button1 et button2 peuvent être associées à des variables dynamiques input1 et input2 de notre page :

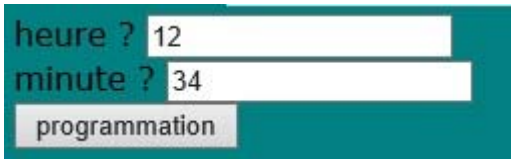
Ce qui après mise à jour nous donne les deux champs à ON et ON

ON
ON

3.3) Requête POST d'un ensemble de variable

On souhaite transmettre des données saisies sur la page web vers le serveur web. La méthode Post va nous permettre de poster ces données.

Prenant les consignes heure et minute d'une page web associées à un bouton programmation :



Ce qui nous donne en code html le formulaire suivant :

```
<form action="http://192.168.4.1/horodatage" method="post">
<div>
<label for="hour">heure ?</label>
<input name="hour" id="hour" value="12">
</div>
<div>
<label for="min">minute ?</label>
<input name="min" value="34">
</div>
<div>
<button>programmation</button>
</div>
</form>
```

A chaque appui sur le bouton programmation une requete post de type horodatage est acheminée du client vers le serveur.

Coté serveur web celui-ci doit gérer la requête post :

```
server.on("/horodatage", HTTP_POST,[](AsyncWebServerRequest *request){
handlehorodatage(request);
  });
```

```
void handlehorodatage(AsyncWebServerRequest *request)
{
Serial.println(request->arg("hour"));
Serial.println(request->arg("min"));
request->send(200, "text/plane", "ok");
}
```

A la réception de la requête Le serveur lancera la fonction handlehorodatage qui récupère les différents champs de celle-ci

Le client envoie la requête http_POST

```
Hypertext Transfer Protocol
  POST /horodatage HTTP/1.1\r\n
  Accept: text/html, application/xhtml+xml, image/jxr, */*\r\n
  Referer: http://192.168.4.1/\r\n
  Accept-Language: fr-FR\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko\r\n
  Content-Type: application/x-www-form-urlencoded\r\n
  Accept-Encoding: gzip, deflate\r\n
  Host: 192.168.4.1\r\n
  Content-Length: 14\r\n
  Connection: Keep-Alive\r\n
  Cache-Control: no-cache\r\n
  \r\n
  [Full request URI: http://192.168.4.1/horodatage]
  [HTTP request 1/1]
  [Response in frame: 37]
  File Data: 14 bytes
HTML Form URL Encoded: application/x-www-form-urlencoded
  Form item: "hour" = "15"
  Form item: "min" = "25"
```

Le serveur web répond ok :

```
Hypertext Transfer Protocol
  HTTP/1.1 200 OK\r\n
  Content-Length: 2\r\n
  Content-Type: text/plane\r\n
  Connection: close\r\n
  Accept-Ranges: none\r\n
  \r\n
  [HTTP response 1/1]
  [Time since request: 0.005724000 seconds]
  [Request in frame: 36]
  File Data: 2 bytes
Media Type
  Media type: text/plane (2 bytes)   6e 65 0d 0a 0d 0a 6f 6b   nges: no ne····ok
```

3.4) Requête GET et POST d'une applet JAVA

On a vu comment charger de manière statique une APPLET JAVA à partir d'un serveur web vers un client web.

Dans la page web on retrouve le code HTML suivant :

```
<applet archive="/web.jar" code="WEB.class" width=430 height=267></applet>
```

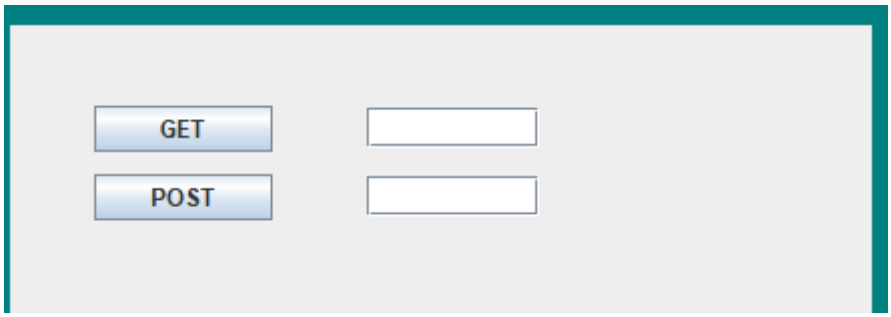
Du coté serveur web :

```
server.on("/web.jar", HTTP_GET, [](AsyncWebServerRequest *request){

  request->send(SPIFFS, "/web.jar", "application/x-java-applet");

 });
```

Une fois son chargement et son exécution accomplies la page web nous affiche l'IHM de l'applet.



Il est possible en code JAVA de pouvoir lancer vers un serveur web une requête du type GET ou POST.

Comme déjà expliqué précédemment le serveur doit gérer les deux types de requête par la méthode on() de l'objet server. Cette gestion est réalisée par le code suivant sur l'ESP :

```
server.on("/data",HTTP_GET,  [](AsyncWebServerRequest *request){

   request->send(200, "text/plane", "325");

 });
server.on("/data1", HTTP_POST,[](AsyncWebServerRequest *request){

 // handleLogin(request);

  request->send(200, "text/plane", "bonjour");

 });
```

Le code suivant en JAVA permet de lancer une requête http_get

```
try {
                        URL myurl = new URL(url);
                        con = (HttpURLConnection) myurl.openConnection();
                        con.setRequestMethod("GET");
                        StringBuilder content;
                        BufferedReader in = new BufferedReader(new
InputStreamReader(con.getInputStream()));

                        String line;
                        content = new StringBuilder();
                        while ((line = in.readLine()) != null) {
                        content.append(line);
                        content.append(System.lineSeparator());

                        }
                        System.out.println(content.toString());
                        textField.setText(content.toString());
                        } catch (Exception e){
                              e.printStackTrace();
                        }

                        finally {
                        con.disconnect();
                        }
```

Le code suivant en JAVA permet de lancer une requête http_post

```
String url1 = "http://192.168.4.1/data1";
String urlParameters = "name=Jack&occupation=programmer";
byte[] postData = urlParameters.getBytes(StandardCharsets.UTF_8);
                        try {
                              URL myurl = new URL(url1);
                              con = (HttpURLConnection) myurl.openConnection();
                              con.setDoOutput(true);
                              con.setRequestMethod("POST");
                              con.setRequestProperty("User-Agent", "Java client");
      con.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
      DataOutputStream wr = new DataOutputStream(con.getOutputStream());
      wr.write(postData);
      StringBuilder content;
      BufferedReader in = new BufferedReader(
      new InputStreamReader(con.getInputStream()));
      String line;
      content = new StringBuilder();
      while ((line = in.readLine()) != null)    {
      content.append(line);
      content.append(System.lineSeparator());
                                                }
      textFieldpost.setText(content.toString());
                        } catch (Exception e){
                              e.printStackTrace();
                        }
                        finally {
                              con.disconnect();
                              }
```

Lors de l'appuie sur le bouton GET, l'applet du client web lance une requête du type http_get vers le serveur

```
Hypertext Transfer Protocol
  GET /data HTTP/1.1\r\n
  User-Agent: Mozilla/4.0 (Windows 10 10.0) Java/1.8.0_162\r\n
  Host: 192.168.4.1\r\n
  Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2\r\n
  Connection: keep-alive\r\n
  \r\n
  [Full request URI: http://192.168.4.1/data]
  [HTTP request 1/1]
  [Response in frame: 7037]
```

Le serveur web ESP répond en fournissant la valeur 6 au client

```
Hypertext Transfer Protocol
  HTTP/1.1 200 OK\r\n
  Content-Length: 1\r\n
  Content-Type: text/plane\r\n
  Connection: close\r\n
  Accept-Ranges: none\r\n
  \r\n
  [HTTP response 1/1]
  [Time since request: 0.028504000 seconds]
  [Request in frame: 7036]
  File Data: 1 bytes
Media Type
  Media type: text/plane (1 byte)    6e 65 0d 0a 0d 0a 36      nges: no ne····6
```

L'applet (donc le client) affiche la valeur 6

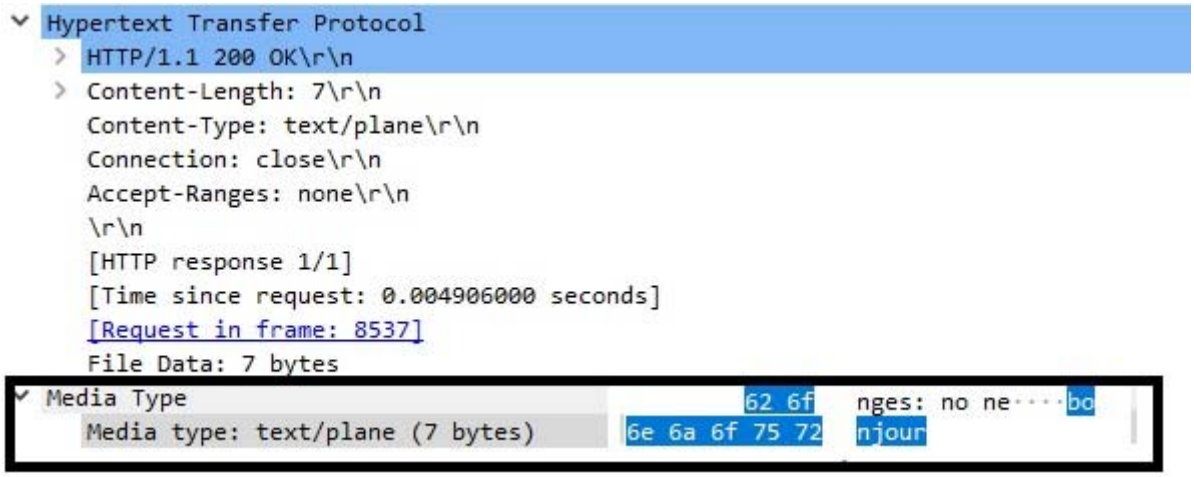


Lors de l'appuie sur le bouton POST, l'applet du client web lance une requête du type http_POST vers le serveur

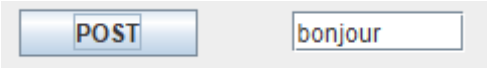
```
Hypertext Transfer Protocol
  POST /data1 HTTP/1.1\r\n
  User-Agent: Java client\r\n
  Content-Type: application/x-www-form-urlencoded\r\n
  Host: 192.168.4.1\r\n
  Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2\r\n
  Connection: keep-alive\r\n
  Content-Length: 31\r\n
  \r\n
  [Full request URI: http://192.168.4.1/data1]
  [HTTP request 1/1]
  [Response in frame: 8539]
  File Data: 31 bytes
HTML Form URL Encoded: application/x-www-form-urlencoded
  Form item: "name" = "Jack"
  Form item: "occupation" = "programmer"
```

On voit les deux paramètres qui sont postés name et programmer

Le serveur web ESP répond en fournissant la chaine bonjour au client web



L'applet (donc le client) affiche bonjour



3.5) Requête get d'un objet graphique du type javascript

Rgragh est une librairie open free permettant de disposer d'objet graphique d'instrumentation de bonne qualité. Le script javascript RGraph.thermometer.js permet de disposer d'un objet graphique du type thermomètre.

Dans le fichier html dans le script javascript on doit déclarer la fonction permettant de créer cet objet

```
function thermometer(d)
{
   var thermometer = new RGraph.Thermometer({
      id: 'cvs',
      min: 0,
      max: 100,
      value: d,
      options: {
      }
   }).draw();
}
```

Cette fonction recevra un entier permettant de faire varier le curseur.

En langage HTML, un canevas ayant comme id **cvs (idem que dans la fontion thermometer)** permettra d'afficher sur la page web cet objet

```
<canvas id="cvs" width="100" height="400">
   [No canvas support]
</canvas>
```

Les scripts suivants devront être déclarés dans la page HTML afin que le serveur WEB puissent les fournir aux client WEBs

```
<script src="/RGraph.svg.common.core.js" ></script>
<script src="/RGraph.svg.common.tooltips.js" ></script>
<script src="/RGraph.svg.hbar.js" ></script>
<script src="/RGraph.common.core.js"></script>
<script src="/RGraph.thermometer.js"></script>
```

La requête get sera lancée toutes les 200ms (setInterval) grâce à la fonction run dans le script de la page HTML. Cette requête permettra du coté du serveur ESP de récupérer la valeur de la température.

```
function run()
{
        var xhttp = new XMLHttpRequest();
 xhttp.onreadystatechange = function() {
if (this.readyState == 4 && this.status == 200) {
 var newstring = new String(this.responseText);
analog=parseInt(this.responseText);
}
};
        xhttp.open("GET", "data", true);
        xhttp.send();
                        if (d1[0]!=analog){
                        d1[0]=analog;
                        thermometer(analog);
}
```

Cette fonction permettra d'alimenter la variable analog fournie par le serveur ESP à chaque requête GET data.

La requête du type GET data sera envoyée vers le serveur WEB ESP. Le dessin du graphe sera lancé dans la fonction run ainsi que la mise à jour du dessin graphique (thermometer(analog);)

Les variables analog et d1 seront des variables globales du script.

```
analog=1;
 d1  = [3];
```

Du côté du serveur WEB ESP on doit pourvoir fournir les différents scripts aux navigateurs WEB suite aux requêtes GET. Pour le script RGraph.common.core.js on aura donc le code suivant :

```
server.on("/RGraph.common.core.js", HTTP_GET, [](AsyncWebServerRequest *request){
   request->send(SPIFFS, "/RGraph.common.core.js", "text/javascript");
 });
```

Le principe restera le même pour les autres scripts.

Les différentes requêtes demandées par le client WEB suite au chargement de la page WEB principale apparaissent, ainsi que la réponse du serveur WEB et leur contenu javascript

http

No.	Time	Source	Destination	Protocol	Length	Info
206	16.862711	192.168.4.2	192.168.4.1	HTTP	312	GET / HTTP/1.1
223	16.909497	192.168.4.1	192.168.4.2	HTTP	493	HTTP/1.1 200 OK (text/html)
240	16.933098	192.168.4.2	192.168.4.1	HTTP	352	GET /RGraph.svg.common.core.js HTTP/1.1
320	17.060833	192.168.4.1	192.168.4.2	HTTP	495	HTTP/1.1 200 OK (text/javascript)
327	17.108092	192.168.4.2	192.168.4.1	HTTP	356	GET /RGraph.svg.common.tooltips.js HTTP/1.1
331	17.117052	192.168.4.1	192.168.4.2	HTTP	603	HTTP/1.1 200 OK (text/javascript)
338	17.128519	192.168.4.2	192.168.4.1	HTTP	345	GET /RGraph.svg.hbar.js HTTP/1.1
366	17.157421	192.168.4.1	192.168.4.2	HTTP	895	HTTP/1.1 200 OK (text/javascript)
373	17.194803	192.168.4.2	192.168.4.1	HTTP	348	GET /RGraph.common.core.js HTTP/1.1
474	17.352690	192.168.4.1	192.168.4.2	HTTP	971	HTTP/1.1 200 OK (text/javascript)
506	17.859030	192.168.4.2	192.168.4.1	HTTP	348	GET /RGraph.thermometer.js HTTP/1.1
525	17.897598	192.168.4.1	192.168.4.2	HTTP	987	HTTP/1.1 200 OK (text/javascript)

L'objet graphique s'affiche :



Pour la mise à jour du curseur du thermomètre on aura le code suivant

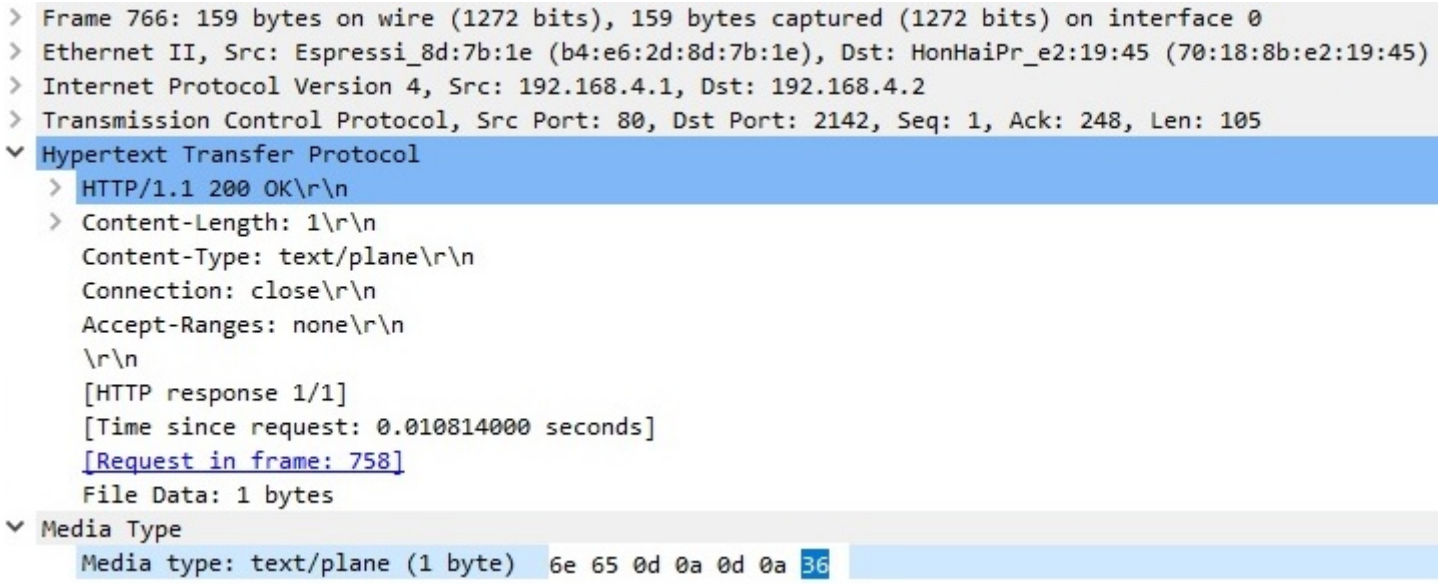
```
server.on("/data",HTTP_GET,  [](AsyncWebServerRequest *request){
   request->send(200, "text/plane", temperature());
 });
```

La fonction temperature renverra au format chaine de caractère la valeur de la température suite à la requête GET data

La requête WEB GET data est demandée par le client WEB

```
> Frame 758: 301 bytes on wire (2408 bits), 301 bytes captured (2408 bits) on interface 0
> Ethernet II, Src: HonHaiPr_e2:19:45 (70:18:8b:e2:19:45), Dst: Espressi_8d:7b:1e (b4:e6:2d:8d:7b:1e)
> Internet Protocol Version 4, Src: 192.168.4.2, Dst: 192.168.4.1
> Transmission Control Protocol, Src Port: 2142, Dst Port: 80, Seq: 1, Ack: 1, Len: 247
v Hypertext Transfer Protocol
  > GET /data HTTP/1.1\r\n
    Accept: */*\r\n
    Referer: http://192.168.4.1/\r\n
    Accept-Language: fr-FR\r\n
    Accept-Encoding: gzip, deflate\r\n
    User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; Trident/7.0; rv:11.0) like Gecko\r\n
    Host: 192.168.4.1\r\n
    Connection: Keep-Alive\r\n
    \r\n
    [Full request URI: http://192.168.4.1/data]
    [HTTP request 1/1]
    [Response in frame: 766]
```

La réponse du serveur WEB à la requête GET DATA est donnée ci-dessous. La valeur de la température est de 6 au format chaine de caractère



available()

Returns the number of bytes available for reading

(that is, the amount of data that has been written to the client by the server it is connected to).

connected()

Whether or not the client is connected.

Note that a client is considered connected if the connection has been closed but there is still unread data.

Returns true if the client is connected, false if not.

```
if (client.available()) {
    char c = client.read();
    Serial.print(c);
  }
if (!client.connected()) {
    Serial.println();
    Serial.println("disconnecting.");
    client.stop();
    for(;;)
      ;
  }
```

stop()

Disconnect from the server

server.available()

Gets a client that is connected to the server and has data available for reading.

The connection persists when the returned client object goes out of scope; you can close it by calling client.stop().

```
void loop() {
  // listen for incoming clients
  WiFiClient client = server.available();
  if (client) {
```

```
    if (client.connected()) {
      Serial.println("Connected to client");
    }

    // close the connection:
    client.stop();
  }
}
```

- http-equiv=’refresh’ : c’est une page que le navigateur devra rafraichir. Pour plus de types, allez ici content=’10’ : toutes les 10 secondes