



Le langage HTML





Le langage HTML

1) Présentation

HTML (HyperText Markup Language) est un langage de balisage qui sert à indiquer au navigateur comment structurer les pages web visitées.

Le HTML se compose d'une série d'éléments avec lesquels vous pouvez encadrer, envelopper ou baliser différentes parties du contenu pour les faire apparaître ou agir d'une certaine manière. Des balises encadrantes peuvent transformer une petite partie de contenu en un lien vers une autre page sur le Web, mettre des mots en italique, etc.

2) Le balisage

Nous pouvons dire d'un texte qu'il agit comme un paragraphe en l'enveloppant d'un élément paragraphe.

```
<p>Le soleil se lève</p>
```

Les principales parties de notre élément sont :

1. La balise ouvrante : il s'agit du nom de l'élément (dans ce cas, p), encadré par un chevron ouvrant (<) et un chevron fermant (>). Elle indique où l'élément commence ou commence à prendre effet — dans ce cas où commence le paragraphe.

2. La balise fermante : c'est la même que la balise ouvrante, sauf qu'elle comprend une barre oblique (/) avant le nom de l'élément. Elle indique la fin de l'élément — dans ce cas, la fin du paragraphe. Ne pas inclure une balise de fermeture est une erreur fréquente chez les débutants, et peut amener d'étranges résultats.

3. Le contenu : il s'agit du contenu de l'élément. Dans notre cas, c'est simplement du texte.

4. L'élément : l'ensemble balise ouvrante, balise fermante et contenu constituent l'élément.

3) Les éléments vides et l'insertion d'une image

Tous les éléments ne suivent pas le modèle ci-dessus d'ouverture de balise, puis contenu, puis fermeture de balise. Certains éléments ne sont composés que d'une balise. Ils servent généralement à insérer / incorporer quelque chose dans le document à l'endroit où ils sont mis. Par exemple, l'élément `<img>` insère une image dans une page à l'endroit où il est placé :

<IMG width="250" height="141" src="index_fichiers/radio.jpg">

4) Les attributs

Les éléments peuvent aussi avoir des attributs, qui comme suit:

```
<p class= "class-css">Le soleil se lève</p>
```

Les attributs contiennent des informations supplémentaires sur l'élément sans qu'elles n'apparaissent dans le contenu réel. Dans ce cas, l'attribut class vous permet de donner à l'élément un nom d'identification qui peut ensuite être utilisé pour cibler l'élément afin de lui attribuer un style CSS ou un comportement particulier, par exemple.

Pour créer un attribut, il faut :

1. insérer une espace entre cet attribut et le nom de l'élément (ou l'attribut précédent, si l'élément possède déjà un ou plusieurs attributs) ;

2. donner un nom à l'attribut, puis ajouter un signe égal ;

3. donner une valeur à l'attribut, entourée par des guillemets d'ouverture et de fermeture.

5) Décryptage de notre premier code HTML

Nous allons voir comment combiner des éléments individuels pour former une page HTML entière :

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="utf-8">
                <title>Ma page test</title>
        </head>
        <body>
        <p>Voici ma page web</p>
        </body>
</html>
```

<!DOCTYPE html> est la chaîne de caractères la plus courte qui soit un doctype valide. C'est tout ce que vous avez vraiment besoin de savoir.

<html></html> : l'élément <html> . Cet élément est le contenant de tout le code de la page et est parfois connu comme l'élément racine.

<head></head> : l'élément <head> . Cet élément a le rôle de conteneur pour toute chose que vous souhaitez inclure dans la page HTML qui ne soit pas du contenu à afficher aux visiteurs de la page

<meta charset="utf-8"> : cet élément définit que le jeu de caractères à utiliser pour votre document est UTF-8.

<title></title> : l'élément title . Il définit le titre de la page, celui qui s'affiche dans l'onglet du navigateur dans lequel la page est chargée et qui est utilisé pour décrire la page lorsque vous la marquez ou l'ajoutez aux favoris.

<body></body> : l'élément <body> . Il contient tout le contenu que vous souhaitez afficher aux internautes lorsqu'ils visitent votre page, que ce soit du texte, des images, des vidéos, des jeux, des pistes audios jouables ou autre.

5.1) L'en tête HTML

Le contenu de l'en-tête HTML <head> — à la différence du contenu de l'élément <body> (affiché quand la page est chargée par le navigateur) — n'est pas affiché dans la page du navigateur. Le travail de la balise <head> est de contenir les métadonnées à propos du document. Dans l'exemple ci-dessus, l'en-tête est plutôt petit :

Toutefois dans les pages plus importantes, l'en-tête peut contenir un grand nombre d'éléments — essayez d'aller sur certains de vos sites web préférés et utilisez les outils de développement pour vérifier le contenu de l'en-tête.

```
<head>
        <meta charset="utf-8">
        <title>Ma page test</title>
</head>
```

5.2) Le titre

Nous avons déjà vu l'élément <title> — qui peut être utilisé pour ajouter un intitulé au document. Il peut toutefois être confondu avec l'élément <h1> , pour ajouter un en-tête de haut niveau au contenu de votre page dans l'élément <body> — quelquefois désigné comme étant le « titre de la page ». Mais ce sont des choses différentes !

• L'élément <h1> apparaît dans la page quand elle est chargée dans le navigateur — généralement, il devrait être utilisé une fois par page, pour marquer le titre du contenu de votre page (le titre d'une histoire, ou d'une actualité, ou tout ce qui vous paraît approprié).

• L'élément <title> est une métadonnée qui représente l'intitulé du document HTML global (non le contenu du document).

5.3) Les métadonnées

Les métadonnées sont des données qui décrivent des données, et le langage HTML a une manière « officielle » d'ajouter des métadonnées à un document — l'élément <meta> .

Les balises **meta** vont nous permettre de donner au navigateur des informations sur la page web à afficher. On dit que ce sont des **métadonnées** (du grec meta "après" et du latin data "informations", soit une donnée servant à définir ou décrire une autre donnée).

Les balises **meta** seront toujours placée entre les balises <head> et </head>. En effet, nous avons vu précédemment que l'entête d'un document HTML contient des informations sur la page alors que le corps (body) contient tout ce qui doit être affiché.

On distingue **2 types** de balises méta :

- Les **balises méta "name"** qui permette de décrire la page.
- Les **balises méta "http-equiv"** qui envoie des informations supplémentaires via le protocole HTTP.

De nombreux éléments <meta> contiennent les attributs name et content :

• name définit le type de méta élément ; le type d'informations contenu.

• content définit le contenu réel de la métadonnée.

Les balises méta "name"

Elles s'utilisent toujours comme ceci :
<meta name="mot clé ici**" content="**valeur ici**" />**

Elles nous permettent notamment de préciser :

- L'auteur de la page grâce au mot clé **author** :

```
<meta name="author" content="Nom de la personne qui a réalisé la page" />
```

- Une description de la page grâce au mot clé **description** :

```
<meta name="description" content="Brève description de la page ici" />
```

- Une liste de mots clés pour qualifier la page grâce au mot clé **keywords**. Ces mots clés seront séparés par des virgules :

```
<meta name="keywords" content="mot clé 1, autre mot clé, blabla" />
```

- Le logiciel utilisé pour réaliser la page grâce au mot clé **generator** :

```
<meta name="generator" content="Adobe Dreamweaver" />
```

Les balises méta "http-equiv"

Elles s'utilisent comme ceci :
<meta http-equiv="mot clé ici**" content="**valeur ici**" />**

Elles nous permettent :

- de préciser l'encodage utilisé pour la page grâce au mot clé **Content-Type**. Vous pouvez en savoir plus l'encodage des fichiers en cliquant sur L'encodage et les caractères spéciaux. Exemple pour un encodage au format UTF-8 :

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
```

- de préciser si la page ne doit pas être sauvegardée dans la cache du navigateur (c'est à dire que la page ne sera pas mémorisée dans les fichiers temporaires du visiteur) grâce au mot clé **pragma** :

```
<meta http-equiv="pragma" content="no-cache" />
```

- de préciser une date d'expiration pour la page grâce au mot clé **expires** :

```
<meta http-equiv="expires" content="Tue, 20 Aug 1996 14:25:27 GMT" />
```

- de demander à la page de se raffraichir (= se réactualiser) toutes les N secondes grâce au mot clé **refresh**. Par exemple, pour que la page se réactualise toutes les 30 secondes :

```
<meta http-equiv="refresh" content="30" />
```

- de rediriger le visiteur vers une autre page au bout de N secondes grâce au mot clé **refresh** et à la précision **URL=chemin-vers-ma-page**. Par exemple, pour rediriger le visiteur sur le site alistapart.com au bout de 2 secondes :

```
meta http-equiv="refresh" content="2;URL=http://www.alistapart.com/" />
```

6) Les caractères spéciaux

En HTML, les caractères < , > , " , ' et & sont des caractères spéciaux. Ils font partie de la syntaxe HTML, comment inclure un de ces caractères dans du texte ?

Nous devons utiliser les références des caractères — codes spéciaux qui représentent des caractères et peuvent être utilisés dans ces circonstances exactes. Chaque référence de caractère est démarrée avec une esperluette (&), et se termine par un point-virgule (;).

Le caractère	Référence équivalente
<	<
>	>
"	"
'	'
&	&

```
<p>En HTML, un paragraphe se définit avec l'élément &lt;p&gt;.</p>
```

7) Les commentaires en HTML

En HTML, comme pour la plupart des langages de programmation, il existe un mécanisme permettant d'écrire des commentaires dans le code. Les commentaires sont ignorés par le navigateur et invisibles à l'utilisateur. Leur but est de permettre d'inclure des commentaires dans le code pour dire comment il fonctionne, que font les diverses parties du code, etc. Cela peut s'avérer très utile si vous revenez à un codage que vous n'avez pas travaillé depuis 6 mois et que vous ne pouvez pas vous rappeler ce que vous avez fait — ou si vous donnez votre code à quelqu'un d'autre pour qu'il y travaille.

Pour transformer une section de contenu dans votre fichier HTML en commentaire, vous devez la mettre dans les marqueurs spéciaux <!-- et --> , par exemple :

```
<!-- <p>Je suis dans un commmentaire!</p> -->
```

CSS est le langage utilisé pour composer vos pages Web (par exemple, changer la police ou les couleurs, ou modifier la mise en page). HTML et CSS vont très bien ensemble, comme vous allez bientôt le découvrir.

Partie 2 : Le formatage du texte

1) Les titres et les paragraphes

La plupart des textes structurés comprennent des titres et des paragraphes, que ce soit dans les romans, les journaux, les livres scolaires, les magazines et ainsi de suite.

En HTML, les paragraphes doivent être contenus dans un élément <p> , comme ceci :

```
<p>Je suis un paragraphe, oh 1 oui je le suis.</p>
```

Chaque titre doit être contenu dans un élément titre :

```
1 <h1>Je suis le titre de l'histoire.</h1>
```

Il y a 6 éléments de titre — <h1> , <h2> , <h3> , <h4> , <h5> , et <h6> . Chaque élément représente un niveau de titre différent ; <h1> représente le titre principal, <h2> représente un sous-titre, <h3> représente un sous-sous-titre, et ainsi de suite jusqu'au niveau de titre le plus bas <h6> .

Prenons l'exemple suivant :

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Document sans nom</title>
</head>
<body>
<h1>L'ennui mortel</h1>

<p>par Chris Mills</p>

<h2>Chapitre I : La nuit noire</h2>
```

```
<p>Il faisait nuit noire. Quelque part une chouette ululait. La pluie tombait sur ...</p>

<h2>Chapitre II : Le silence éternel</h2>

<p>Notre protagoniste ne pouvait même pas murmurer à l'ombre de la silhouette...</p>

<h3>Le spectre parle</h3>

<p>Plusieurs heures s'étaient écoulées, quand soudain le spectre assis se releva et s'exclama : : « S'il vous plaît, ayez pitié de mon âme ! »...</p>

</body>
</html>
```

Qui donne la page web suivante :

L'ennui mortel

par Chris Mills

Chapitre I : La nuit noire

Il faisait nuit noire. Quelque part une chouette ululait. La pluie tombait sur ...

Chapitre II : Le silence éternel

Notre protagoniste ne pouvait même pas murmurer à l'ombre de la silhouette...

Le spectre parle

Plusieurs heures s'étaient écoulées, quand soudain le spectre assis se releva et s'exclama : : « S'il vous plaît, ayez pitié de mon âme ! »...

2) Les listes

Les listes sont très répandues sur le web, nous allons nous intéresser aux deux différents types de liste.

Non ordonnées

Les listes non ordonnées sont utilisées pour représenter des listes d'éléments pour lesquelles l'ordre n'a pas d'importance. Prenons par exemple une liste de courses :

lait

oeufs

pain

houmous

Les listes non ordonnées débutent par un élément <ul> (unorderd list) qui enveloppe tous les éléments de la liste:

Chaque item est contenu dans un élément <li> (list item):

```
<ul>
<li>lait</li>
<li>oeufs</li>
<li>pain</li>
<li>houmous</li>
</ul>
```

On obtient la page web suivante :

- lait
- oeufs
- pain
- houmous

ordonnées

Les listes ordonnées permettent de représenter des listes dans lesquelles l'ordre des éléments a de l'importance — prenons par exemple une série de direction à suivre:

1)Roulez jusqu'au bout de la route

2)Tournez à droite

3)Allez tout droit aux deux premiers rond-points

4)Tournez à gauche au troisième rond-point

5)Roulez sur 300 mètres, l'école est sur votre droite

Les balises suivent la même structure que pour les listes ordonnées, à cela près que la liste est contenue dans l'élément <ol> (ordered list), et non dans <ul> :

```
<ol>
<li>Roulez jusqu'au bout de la route</li>
<li>Tournez à droite</li>
<li>Allez tout droit aux deux premiers rond-points</li>
<li>Tournez à gauche au troisième rond-point</li>
<li>Roulez sur 300 mètres, l'école est sur votre droite</li>
</ol>
```

On obtient la page web suivante :

1. Roulez jusqu'au bout de la route
2. Tournez à droite
3. Allez tout droit aux deux premiers rond-points
4. Tournez à gauche au troisième rond-point
5. Roulez sur 300 mètres, l'école est sur votre droite

Le texte en italique

Dans le langage parlé, pour accentuer, nous insistons sur certains mots, modifiant subtilement le sens de ce que nous disons. De même, dans le langage écrit, nous avons tendance à mettre un certain accent sur des mots en les écrivant en italique.

Nous utilisons l'élément (emphase) pour marquer ces circonstances. Outre rendre le document plus intéressant à lire, ces marqueurs sont reconnus par les lecteurs d'écran et exprimés sur un ton de voix différent.

Je suis content que vous n'ayez pas été en retard.

Je suis *content* que vous n'ayez pas été *en retard*.

Le surgras

Pour mettre l'accent sur des mots très importants, nous les soulignons d'un ton particulier dans la langue parlée et nous les mettons en caractères gras dans la langue écrite. Par exemple :

Ce liquide est **hautement toxique**

En HTML, nous utilisons l'élément (forte importance) comme balise de telles circonstances.

Ce liquide est hautement toxique.

Les autres marqueurs

Les éléments dont nous avons discuté jusqu'à présent ont une sémantique bien définie. La situation avec <b> , <i> et <u> est un peu plus complexe. Ils sont apparus pour que les personnes puissent écrire du texte en gras, en italique ou souligné à une époque où le CSS était encore mal ou pas du tout pris en charge.

Exemple

```
<!-- noms scientifiques -->
<p>
  Le colibri à gorge rouge (<i>Archilochus colubris</i>)
  est le colibri le plus courant dans l'ouest de l'Amérique du Nord.
</p>

<!-- mots dans une langue étrangère -->
<p>
  Le menu était un océan de mots exotiques comme <i lang="uk-latn">vatrushka</i>,
  <i lang="id">nasi goreng</i> et <i lang="fr">soupe à l'oignon</i>.
</p>

<!-- une faute d'orthographe connue -->
<p>
```

```
  Un jour, j'apprendrai comment mieux <u>épeler</u>.
</p>

<!-- Mettre en évidence les mots-clés dans un ensemble d'instructions -->
<ol>
  <li>
    <b>Trancher</b> deux morceaux de pain dans la miche.
  </li>
  <li>
    <b>Mettre</b> une rondelle de tomate et une feuille de laitue
    entre les deux tranches de pain.
  </li>
</ol>
```

Le colibri à gorge rouge (*Archilochus colubris*) est le colibri le plus courant dans l'ouest de l'Amérique du Nord.

Le menu était un océan de mots exotiques comme *vatrushka*, *nasi goreng* et *soupe à l'oignon*.

Un jour, j'apprendrai comment mieux épeler.

1. **Trancher** deux morceaux de pain dans la miche.
2. **Mettre** une rondelle de tomate et une feuille de laitue entre les deux tranches de pain.

Sauts de ligne et traits horizontaux

Les éléments <br> et <hr> sont utilisés à l'occasion et il convient de les connaître :

 crée un saut de ligne dans un paragraphe ; c'est le seul moyen de forcer une structure rigide quand vous voulez obtenir une suite de lignes courtes fixes, comme dans une adresse postale ou un poème. Par exemple :

Il y avait une fois une fille nommée Nell

Qui aimait écrire du HTML

Mais ses structures et sémantiques affligeantes

rendaient de ses marquages la lecture inélégante.

qui donne :

Il y avait une fois une fille nommée Nell
Qui aimait écrire du HTML
Mais ses structures et sémantiques affligeantes
rendaient de ses marquages la lecture inélégante.

Les éléments <hr> créent un trait horizontal dans le document marquant un changement thématique dans le texte (comme un changement de sujet ou de scène). Visuellement, c'est juste un trait horizontal, comme dans cet exemple :

Ron était acculé dans un angle par des Netherbeasts en maraude. Effrayé, mais déterminé à protéger ses amis, il a levé sa baguette et s'est préparé à se battre, espérant que son appel de détresse serait entendu.

Pendant ce temps, Harry était assis à la maison, regardant sa déclaration de redevances et réfléchissant à la sortie du prochaine épisode de la fiction, quand une lettre de détresse enchantée passa par sa fenêtre et atterrit sur ses genoux. Il la lut à la hâte et, se dressant brusquement, « mieux vaut retourner au travail alors » se dit-il.

La balise DIV

La balise ** ** définit une division ou une section, appelé aussi calque, dans un document HTML.
La balise ** ** est utilisée pour regrouper des éléments sous forme de bloc et ainsi pouvoir les formater en CSS.
Remarque : Les navigateurs placent toujours un saut de ligne avant et après un élément ** **.

C'est un conteneur (à appréhender comme un cadre) qui peut inclure tous les tags html (tels que les paragraphes les tableaux, les listes, les images ..., et également d'autres divisions).
Cet élément est l'un des plus utilisé pour structurer les pages html en plusieurs grandes sections (en tête, partie principale, partie navigation, pied de page ...). En dehors de cet élément, les listes sont également très utilisées elles aussi pour structurer une page html, sous condition de leur associer des styles CSS adapté.

```
<div style="z-index:1; position:relative; width:250px; height:120px; border:solid 1px black;
background-color:lightgray;">
</div>
<div style="z-index:0; position:relative; top:-50px; left:100px; width:250px; height:120px; border:solid
1px black; background-color:lightsteelblue;">
</div>
```

```
<div style="width:300px; height:120px; overflow:auto; border:solid 1px black;">
  <ul>
    <li>Elémént 1</li>
    <li>Elémént 2</li>
    <li>Elémént 3</li>
    <li>Elémént 4</li>
    <li>Elémént 5</li>
  </ul>
</div>
```

La balise span et l'attribut id

La balise span au contraire d'une balise div est une balise de type "inline" et se trouve obligatoirement à l'intérieur d'une balise "block". Une balise inline ne crée pas de retour à la ligne, le texte qui se trouve à l'intérieur s'écrit donc à la suite du texte précédent, sur la même ligne (c'est pour cela que l'on parle de balise "en ligne").

Une balise de type "block" sur votre page web crée automatiquement un retour à la ligne avant et après. Il suffit d'imaginer tout simplement un bloc. Votre page web sera en fait constituée d'une série de blocks à la suite les uns des autres. Mais vous verrez qu'en plus, il est possible de mettre un block à l'intérieur d'un autre, ce qui va augmenter considérablement nos possibilités pour créer le design du site !

L'attribut id fournit un identificateur unique pour un élément dans un document. L'attribut id doit commencer par une lettre :

```
<span id="me">This is me</span>
```

L'élément id est utilisé par le code javascript pour faire un changement ou additionner un comportement à un élément.

3) Les liens hypertext :

Ils nous permettent de lier nos documents à n'importe quel autre document (ou autre ressource) voulu ; nous pouvons faire des liens vers des parties précises de documents et rendre des applications disponibles à une simple adresse web (contrairement aux applications natives, qui doivent être installées et tout le travail). À peu près tout contenu web peut être converti en lien, de sorte que cliqué (ou activé autrement), il dirigera le navigateur vers une autre adresse web (URL).

Un lien élémentaire se crée en intégrant le texte (ou tout autre contenu, cf. Liens de niveau bloc) que vous voulez transformer en lien dans un élément <a> et en lui affectant un attribut href (également connu comme étant une **Hypertext Reference**, ou **target**) contenant l'adresse web vers laquelle vous voulez que le lien pointe.

```
<p>Je suis en train de créer un lien à
<a href="https://www.mozilla.org/fr/">la page d'accueil Mozilla</a>.
</p>
```

L'autre attribut qu'il est possible d'ajouter à un lien est title ; il est destiné à contenir des informations utiles supplémentaires à propos du lien, comme le type d'informations contenues dans la page ou ce qu'il faut savoir. Par exemple :

```
<p>Je suis en train de créer un lien à
<a href="https://www.mozilla.org/fr/"
```

```
   title="Le meilleur endroit pour trouver plus d'informations sur la
  mission de Mozilla et comment contribuer">la page d'accueil Mozilla</a>.
</p>
```

Liens de niveau bloc

Comme mentionné précédemment, vous pouvez transformer à peu près tout contenu en un lien, même des éléments bloc . Si vous avez une image que vous voulez transformer en lien, vous avez juste à mettre l'image entre les balises <a>.

```
<a href="https://www.mozilla.org/fr/">
  <img src="mozilla-image.png" alt="logo mozilla pointant sur la page d'accueil mozilla">
</a>
```

Liaison vers des ressources non-HTML : signalez-les clairement

Quand faites un lien vers une ressource à télécharger (comme un PDF ou un document Word) ou lue en flux (comme une video ou de l'audio) ou qui a un autre effet potentiellement inattendu (qui ouvre d'une fenêtre pop-up ou qui charge une vidéo Flash), vous devez le signaler clairement pour réduire toute confusion. Cela peut être parfaitement ennuyeux, par exemple :

- si vous êtes sur une connexion à faible bande passante, cliquer sur un lien et initier un téléchargement de plusieurs mégaoctets de façon inattendue.
- si vous n'avez pas Flash player installé, cliquer sur un lien et être soudain redirigé vers une page qui nécessite Flash.

Voici quelques exemples suggérant les genres de texte pouvant être employé :

```
<p><a href="http://www.exemple.com/rapport-volumineux.pdf">
  Télécharger le rapport des ventes (PDF, 10Mo)
</a></p>

<p><a href="http://www.exemple.com/flux-video/" target="_blank">
  Regarder la vidéo (le flux s'ouvre dans un nouvel onglet, qualité HD)
</a></p>

<p><a href="http://www.exemple.com/jeu-de-voiture">
  Jouer au jeu de voiture (nécessite Flash)
</a></p>
```

Lien de téléchargement

Utilisez l'attribut **download** pour faire un lien vers un téléchargement

Quand vous faites un lien avec une ressource qui doit être téléchargée plutôt qu'ouverte dans le navigateur, vous pouvez utiliser l'attribut download pour fournir un nom d'enregistrement par défaut. Voici un exemple avec un lien de téléchargement vers la version Windows la plus récente de Firefox :

```
<a href="https://download.mozilla.org/?product=firefox-latest-ssl&os=win64&lang=fr-FR"
   download="firefox-latest-64bit-installer.exe">
  Télécharger la version de Firefox pour Windows la plus récente (64-bit)(français, France)
</a>
```

Liens de courriel

Il est possible de créer des liens ou des boutons qui, lorsqu'ils sont cliqués, ouvrent un nouveau message de courriel sortant plutôt que de faire un lien vers une ressource ou une page. C'est fait en utilisant l'élément <a> et le schéma d'URL mailto:.

Sous sa forme la plus basique et la plus communément utilisée, un lien mailto: indique simplement l'adresse du destinataire voulu. Par exemple :

```
<a href="mailto:nullepart@mozilla.org">Envoyer un courriel à nullepart</a>
```

Les liens CSS et Javascript

À peu près tous les sites web que vous rencontrerez actuellement utiliseront des CSS pour agrémenter leur aspect, et JavaScript pour assurer les fonctionnalités interactives, telles que lecteurs vidéo, cartes géographiques, jeux et plus encore. Ceux-ci sont le plus souvent appliqués à une page web en utilisant respectivement les éléments <link> et <script> .

L'élément <link> se situe toujours dans l'en-tête du document. Il comporte deux attributs, rel="stylesheet" indiquant qu'il s'agt de la feuille de style du document, et href indiquant le chemin d'accès au fichier la contenant :

```
<link rel="stylesheet" href="mon_1 fichier_css.css">
```

L'élément <script> ne doit pas aller dans l'en-tête ; en fait, il est souvent préférable de le placer au bas du corps du document (juste avant la balise de clôture </body>), pour s'assurer que tout le contenu HTML a été lu par le

navigateur avant de lui appliquer le JavaScript (si JavaScript essaie d'accéder à un élément qui n'existe pas encore, le navigateur déclenchera une erreur).

```
<script src="mon-fichier-js.js"></script>
```

Au lieu de pointer vers un fichier de script externe, vous pouvez également choisir de mettre le code du script dans le HTML à l'intérieur d'un élément <script> .

Les Applets JAVA

Il est nécessaire de rajouter une page HTML qui s'occupera d'intégrer cette applet au milieu des autres éléments classiques que comporte une page Web, savoir ; les textes, les images, les listes, etc. L'intégration de l'applet se fait comme pour les autres éléments, par l'intermédiaire d'une balise adaptée et prédéfinie qui s'appelle, tout simplement, <applet>. Cette balise dispose également de paramètres qui permettent, entre autres, de spécifier l'applet à gérer ainsi que ses dimensions. Ces trois paramètres sont un minimum et sont donc obligatoires.

Exemple :

```
<head>
<title>
testapplet
</title>
</head>
<body>
<img src="aster.jpg" width="413" height="100" align="right" />
<img src="retour.jpg" width="128" height="65">
<H1 align="center">BIT OFF-line AWMLS <BR></H1>
<table width="100%" height=100%">
<tr align="center"><td vertical-align="middle">
<applet
archive="applettest.jar"
code="Applettest.class"
width=800 height=600>
</applet>
</td></tr></table>
<P>retour vers page précédente
<A href="applettest2.htm">lien</A>.</P>
</body>
</html>
```

La création de l'applet peut s'effectuer sur un outil de développement rapide (WINDOWBUILDER dans notre cas de figure)

Partie 3 : Structure de Site Web et de document

Principales parties d'un document

Les pages web peuvent sembler assez différentes les unes des autres, mais elles ont toutes tendance à partager des composantes standard similaires :

En-tête (header)

Généralement une grande bande placée en travers au haut de la page avec un titre ou un logo. C'est là où les principales informations du site restent d'une page à l'autre.

Barre de navigation

Elle fait le lien vers les principales parties du site ; d'habitude, elle est présentée sous forme de boutons de menu, de liens ou d'onglets. Comme l'en-tête, la barre de navigation reste souvent cohérente d'une page à l'autre.

Beaucoup de créateurs de site considèrent la barre de navigation partie de l'en-tête et non comme un composant individuel, mais ce n'est pas une exigence. En fait certains soutiennent également que le fait d'avoir les deux séparés est préférable pour l'accessibilité, car les lecteurs d'écran lisent mieux les deux éléments s'ils sont séparés.

Contenu principal

Une grande zone au centre contenant la majeure partie du contenu unique de la dite page web, par ex. la vidéo à regarder, ou le corps de l'article à parcourir, ou la carte à lire, ou les dernières nouvelles etc. C'est la partie du site variable de page en page.

Pied de page

Une bande au bas de la page qui contient généralement, en petits caractères, des avis de droit d'auteur ou des coordonnées de contact. C'est un endroit pour mettre de l'information commune (comme l'en-tête), mais il s'agit dans ce cas d'informations non-critiques, voire secondaires par rapport au site Web lui-même.

Le pied de page est aussi parfois utilisé à des fins de SEO, en fournissant des liens pour un accès rapide à des contenus prisés. Un « site web typique » pourrait ressembler à quelque chose comme ceci :



Certains sites Web ont plus de colonnes, d'autres sont beaucoup plus complexes, mais vous voyez l'idée.

Avec le bon CSS, vous pouvez utiliser à peu près n'importe quel élément pour envelopper les différentes sections et obtenir l'apparence que vous voulez, mais comme nous l'avons déjà dit, nous devons respecter la sémantique et utiliser le bon élément pour le bon travail.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My page title</title>
    <link
href="https://fonts.googleapis.com/css?family=Open+Sans+Condensed:300|Sonsie+One"
rel="stylesheet" type="text/css">
    <link rel="stylesheet" href="style.css">

    <!-- the below three lines are a fix to get HTML5 semantic elements working in old
versions of Internet Explorer-->
    <!--[if lt IE 9]>
```

```
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/html5shiv/3.7.3/html5shiv.js"></script>
    <![endif]-->
  </head>

  <body>
    <!-- Here is our main header that is used across all the pages of our website -->

    <header>
      <h1>Header</h1>
    </header>

    <nav>
      <ul>
        <li><a href="#">Home</a></li>
        <li><a href="#">Our team</a></li>
        <li><a href="#">Projects</a></li>
        <li><a href="#">Contact</a></li>
      </ul>

      <!-- A Search form is another commmon non-linear way to navigate through a website.
-->

      <form>
        <input type="search" name="q" placeholder="Search query">
        <input type="submit" value="Go!">
      </form>
    </nav>

    <!-- Here is our page's main content -->
    <main>

      <!-- It contains an article -->
      <article>
        <h2>Article heading</h2>

        <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Donec a diam lectus.
Set sit amet ipsum mauris. Maecenas congue ligula as quam viverra nec consectetur ant
```

```
hendrerit. Donec et mollis dolor. Praesent et diam eget libero egestas mattis sit amet
vitae augue. Nam tincidunt congue enim, ut porta lorem lacinia consectetur.</p>


        <h3>subsection</h3>


        <p>Donec ut librero sed accu vehicula ultricies a non tortor. Lorem ipsum dolor
sit amet, consectetur adipisicing elit. Aenean ut gravida lorem. Ut turpis felis,
pulvinar a semper sed, adipiscing id dolor.</p>


        <p>Pelientesque auctor nisi id magna consequat sagittis. Curabitur dapibus, enim
sit amet elit pharetra tincidunt feugiat nist imperdiet. Ut convallis libero in urna
ultrices accumsan. Donec sed odio eros.</p>


        <h3>Another subsection</h3>


        <p>Donec viverra mi quis quam pulvinar at malesuada arcu rhoncus. Cum soclis
natoque penatibus et manis dis parturient montes, nascetur ridiculus mus. In rutrum
accumsan ultricies. Mauris vitae nisi at sem facilisis semper ac in est.</p>


        <p>Vivamus fermentum semper porta. Nunc diam velit, adipscing ut tristique vitae
sagittis vel odio. Maecenas convallis ullamcorper ultricied. Curabitur ornare, ligula
semper consectetur sagittis, nisi diam iaculis velit, is fringille sem nunc vet mi.</p>
      </article>


      <!-- the aside content can also be nested within the main content -->
      <aside>
        <h2>Related</h2>


        <ul>
          <li><a href="#">Oh I do like to be beside the seaside</a></li>
          <li><a href="#">Oh I do like to be beside the sea</a></li>
          <li><a href="#">Although in the North of England</a></li>
          <li><a href="#">It never stops raining</a></li>
          <li><a href="#">Oh well...</a></li>
        </ul>
      </aside>


    </main>


    <!-- And here is our main footer that is used across all the pages of our website -->
```

```
    <footer>
      <p>©Copyright 2050 by nobody. All rights reversed.</p>
    </footer>

  </body>
</html>
```

Pour mettre en oeuvre le marquage sémantique, HTML fournit des balises dédiées que vous pourrez utiliser pour représenter ces parties, par exemple :

• header : <header> .

• barre de navigation : <nav> .

• contenu principal : <main> , avec diverses sous-sections de contenu représentées à l'aide de des éléments <article> , <section> et .

• barre latérale : <aside> ; souvent mise à l'intérieur de l'élément <main> .

• pied de page : <footer> .

Partie4 : Les formulaires

Pour construire notre formulaire, nous aurons besoin des éléments HTML suivants : <form> , <label> , <input> , <textarea> et <button> .

L'élément <form>

Tous les formulaires HTML débutent par un élément <form> comme celui-ci :

```
<form action="/my-handling-form-page" method="post">
</form>
```

Cet élément définit un formulaire. C'est un élément conteneur au même titre que les éléments ou <p> , mais il accepte aussi quelques attributs spécifiques afin de contrôler la manière dont il se comporte. Tous ses attributs sont optionnels mais définir au moins les attributs action et method est considéré comme de bonne pratique.

• L'attribut action définit l'emplacement (une URL) où doivent être envoyées les données collectées par le formulaire.

• L'attribut method définit la méthode HTTP utilisée pour envoyer les données (cela peut être « get » ou « post »).

Pour le moment, ajoutez l'élément <form> ci dessus dans le corps de votre HTML.

Les éléments <label>, <input> et <textarea>

Notre formulaire de contact est très simple et ne contient que trois champs de texte, chacun ayant une étiquette. Le champ d'entrée pour le nom est un champ de texte sur une seule ligne, le champ pour l'adresse électronique est un champ de texte sur une ligne qui n'accepte que des adresses électroniques et enfin le champ pour le message est un champ de texte sur plusieurs lignes.

En termes de code HTML, nous avons besoin de quelque chose qui ressemble à ceci pour mettre en œuvre nos widgets de formulaire.

```
<form action="/ma-page-de-traitement" method="post">
        <div>
```

```
            <label for="name">Nom :</label>
            <input type="text" id="name" name="user_name">
        </div>
        <div>
            <label for="mail">e-mail :</label>
            <input type="email" id="mail" name="user_mail">
        </div>
        <div>
            <label for="msg">Message :</label>
            <textarea id="msg" name="user_message"></textarea>
        </div>
</form>
```

Les éléments sont ici pour structurer notre code et rendre la mise en page plus facile (voir ci-dessous). Veuillez noter l'utilisation de l'attribut for sur tous les éléments <label> . C'est une manière formelle de lier un libellé à un élément du formulaire. Cet attribut fait référence à l' id de l'élément correspondant. Il y a plusieurs avantages à faire ainsi. Le plus évident de permettre à l'utilisateur de cliquer sur l'étiquette pour activer le bloc correspondant. Si vous souhaitez mieux comprendre les bénéfices de cet attribut.

Concernant l'élément <input> , l'attribut le plus important est l'attribut type . Ce dernier est extrêmement important puisqu'il définit le comportement de l'élément <input> . Il peut radicalement changer le sens de l'élément, faites-y attention. Si vous voulez en savoir plus à ce propos, vous pouvez lire l'article au sujet des widgets natifs pour formulaire.

• Dans notre exemple nous n'utilisons que la valeur text — qui est la valeur par défaut de cet attribut et représente un champ de texte basique sur une seule ligne acceptant n'importe quel type de texte.

• Pour la deuxième entrée, nous utilisons la valeur email qui définit un champ de texte sur une seule ligne n'acceptant que des adresses électroniques valides.

Cette dernière valeur transforme un champ basique en une sorte de champ « intelligent » qui réalise des vérifications sur les données fournies par l'utilisateur.

Last but not least, remarquez la syntaxe de <input> vs <textarea></textarea> . C'est une des bizarreries du HTML. La balise <input> est un élément vide, ce qui signifie qu'il n'a pas besoin de

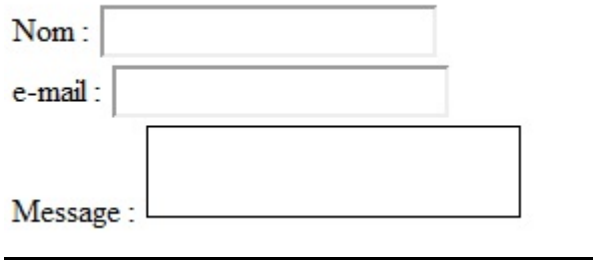
balise fermante. Au contraire, <textarea> n'est pas un élément vide, il faut donc le fermer avec la balise fermante appropriée. Cela a un effet sur une caractéristique spécifique des formulaires HTML : la manière dont vous définissez la valeur par défaut. Pour définir une valeur par défaut d'un élément <input> vous devez utiliser l'attribut value de la manière suivante :

<input type="text" value="par défaut cet élément sera renseigné avec ce texte">

A contrario, si vous souhaitez définir la valeur par défaut d'un élément <textarea>, il suffit simplement de mettre la valeur par défaut entre les balises ouvrantes et fermantes de l'élément <textarea> de la manière suivante :

```
<textarea>par défaut cet élément sera renseigné avec ce texte</textarea>
```

L'interface WEB est la suivante :



L'élément <button>

Notre formulaire est presque terminé. Il nous suffit seulement d'ajouter un bouton pour permettre à l'utilisateur de nous envoyer les données renseignées dans le formulaire. Ceci se fait simplement en ajoutant d'un élément <button> ; ajoutez-le juste avant la balise fermante </form> :

```
<div class="button">
<button type="submit">Envoyer le message</button>
</div>
```

Comme vous le voyez l'élément <button> accepte aussi un attribut de type — il peut prendre une des trois valeurs : submit , reset ou button .

Un clic sur un bouton submit (valeur par défaut) envoie les données du formulaire vers la page définie par l'attribut action de l'élément <form> .

• Un clic sur un bouton reset réinitialise tous les widgets du formulaire à leurs valeurs par défaut immédiatement. Du point de vue de l'expérience utilisateur, utiliser un tel bouton est une mauvaise pratique.

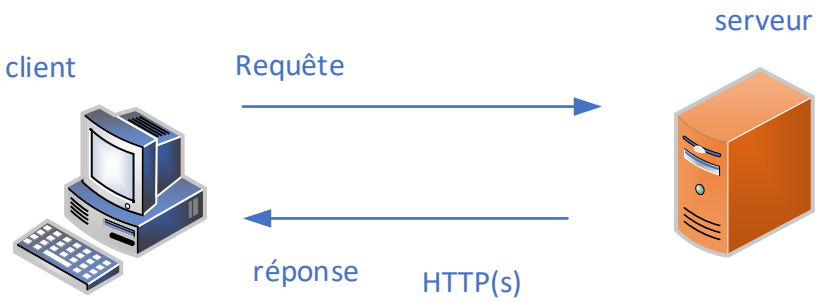
• Un clic sur un bouton button ne fait... rien ! Cela peut paraître stupide mais c'est en réalité très pratique pour concevoir des boutons personnalisés avec JavaScript.

L'interface WEB est la suivante :



Envoyer et extraire les données d'un formulaire :

Les technologies IP dont fait partie le web se fonde sur une architecture client/serveur élémentaire ; en résumé : un client (généralement un navigateur Web) envoie une requête à un serveur (le plus souvent un serveur web comme Apache, Nginx, IIS, Tomcat...), en utilisant le protocole HTTP. Le serveur répond à la requête en utilisant le même protocole.



Côté client, un formulaire HTML n'est rien d'autre qu'un moyen commode et convivial de configurer une requête HTTP pour envoyer des données à un serveur. L'utilisateur peut ainsi adresser des informations à joindre à la requête HTTP.

L'élément <form> définit la méthode d'envoi des données. Tous ses attributs sont conçus pour vous permettre de configurer la requête à envoyer quand un utilisateur presse le bouton d'envoi. Les deux attributs les plus importants sont action et method .

L'attribut action

Cet attribut définit où les données sont envoyées. Sa valeur doit être une URL valide.

S'il n'est pas fourni, les données seront envoyées à l'URL de la page contenant le formulaire

Dans cet exemple, les données sont envoyées à une URL précise — http://foo.com :

```
<form action="1 http://foo.com">
```

Ici, nous utilisons une URL relative — les données sont envoyées à une URL différente sur le serveur :

```
<form action="/somewhere_else">
```

Sans attribut, comme ci-dessous, les données de <form> sont envoyées à la même page que celle du formulaire :

```
<form>
```

L'attribut method

Cet attribut définit comment les données sont envoyées. Le Protocole HTTP fournit plusieurs façons d'envoyer une requête. Les données des formulaires HTML peuvent être envoyées via au moins deux méthodes : la méthode GET et la méthode POST.

Pour bien comprendre la différence entre ces deux méthodes, il convient d'examiner comment le protocole HTTP marche. Chaque fois qu'on veut atteindre une ressource sur Internet, le navigateur envoie une requête à une URL. Une requête HTTP consiste en deux parties : un en-tête (header) qui contient les métadonnées sur les capacités du navigateur, et un corps (body) qui contient les informations nécessaires au serveur pour effectuer la requête.

La méthode GET

La méthode GET est utilisée par le navigateur pour demander au serveur de renvoyer une certaine ressource. "Hé le serveur, je veux cette ressource." Dans ce cas, le navigateur envoie un corps vide. Du coup, si un formulaire est envoyé avec cette méthode, les données envoyées au serveur sont ajoutées à l'URL.

L'ensemble des données du formulaire est rajouté à l'URI spécifié par l'attribut action (avec comme séparateur un caractère point d'interrogation ?) et ce nouvel URI est envoyé à l'agent de traitement.

Considérons le formulaire suivant :

```
<form action="http://foo.com" method="get">
        <div>
                <label for="say">Quel salut voulez-vous adresser ?</label>
                <input name="say" id="say" value="Hi">
        </div>
        <div>
                <label for="to">À qui voulez-vous l'adresser ?</label>
                <input name="to" value="Mom">
        </div>
        <div>
                <button>Send my greetings</button>
        </div>
</form>
```

Qui donne :

Quel salut voulez-vous adresser ? Hi

À qui voulez-vous l'adresser ? Mom

Send my greetings

Les données sont ajoutées à l'URL sous forme d'une suite de paires nom/valeur. À la fin de l'URL de l'adresse Web, il y a un point d'interrogation (?) suivi par les paires nom/valeur séparés par une esperluette (&). Dans ce cas, nous passons deux éléments de données au serveur :

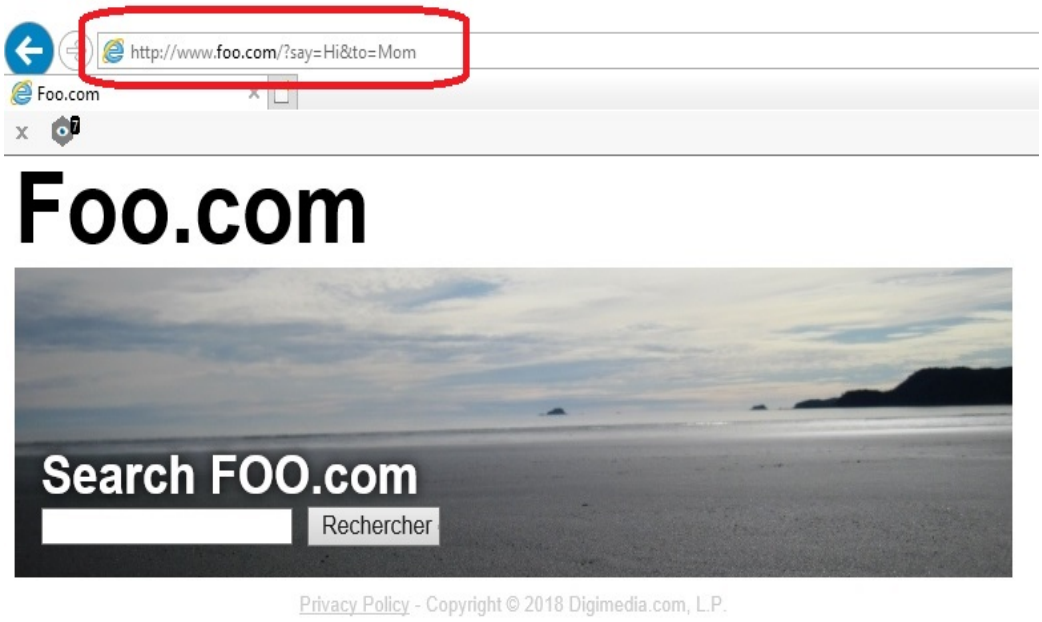
• say , dont la valeur est Hi

• to , qui a la valeur Mom

La requête HTTP ressemble à quelque chose comme :

GET /?say=Hi&to=Mom HTTP/1.1

Host: foo.com



La méthode POST

La méthode POST est un peu différente. C'est la méthode que le navigateur utilise pour demander au serveur une réponse prenant en compte les données contenues dans le corps de la requête HTTP : « Hé serveur ! vois ces données et renvoie-moi le résultat approprié ». Si un formulaire est envoyé avec cette méthode, les données sont ajoutées au corps de la requête HTTP.

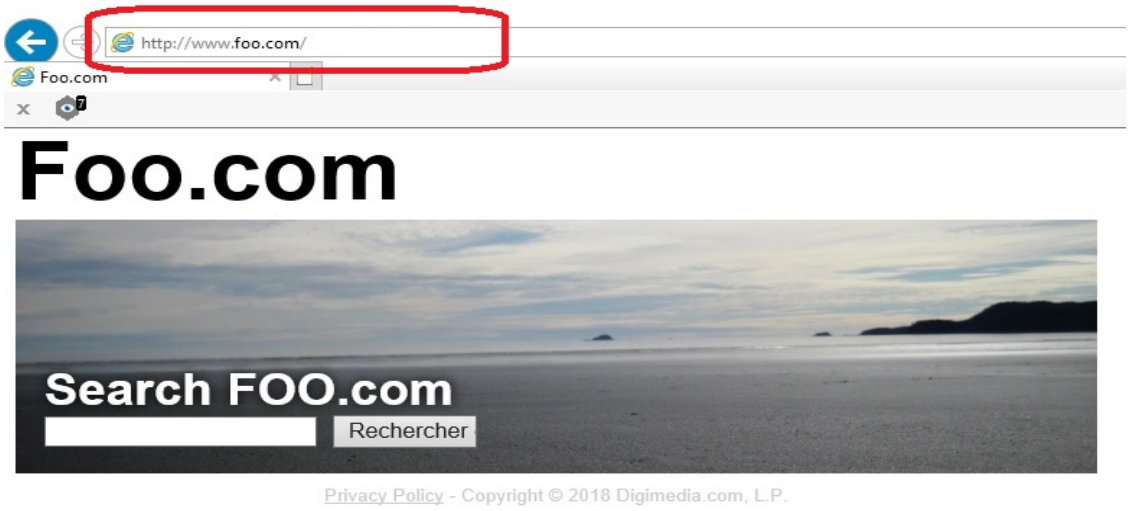
Voyons un exemple — c'est le même formulaire que celui que nous avons vu pour GET ci-dessus, mais avec post comme valeur de l'attribut method .

```
<form action="http://foo.com" method="post">
        <div>
                <label for="say">Quel salut voulez-vous adresser ?</label>
                <input name="say" id="say" value="Hi">
        </div>
        <div>
                <label for="to">À qui voulez-vous l'adresser ?</label>
                <input name="to" value="Mom">
        </div>
        <div>
                <button>Send my greetings</button>
        </div>
</form>
```

Quand le formulaire est soumis avec la méthode POST, aucune donnée n'est ajoutée à l'URL et la requête HTTP ressemble à ceci, les données incorporées au corps de requête :

```
POST / HTTP/1.1
Host: foo.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 13
say=Hi&to=Mom
```

L'en-tête Content-Length indique la taille du corps, et l'en-tête Content-Type indique le type de ressources envoyées au serveur.



Partie 5 : Mise en place d'un serveur WEB sur un IOT ou ESP32 ou ESP8266

Mise en place d'une page statique réactualisé sur le module ESP8266

La librairie ESP8266WiFi.h pour l'ESP 8266 doit être inclus pour mettre en place le serveur.

L'appel du constructeur wifiServer permet de mettre en place un serveur WEB WIFI sur le port 80

L'appel du constructeur apIP permet de mettre en place un objet point d'accès WIFi sur l'@ 192.168.4.1.

Les deux méthodes WiFi.softAPConfig et WiFi.softAP permettent de configurer le point d'accès avec le SSID, l'adresse IP, le masque ….

Le serveur est démarré du moment que la méthode wifiServer.begin() est exécutée

La méthode LOOP récupère un objet client lors de la connexion de celui-ci WiFiClient client = wifiServer.available();

Les deux méthodes connected() et available() permettent de vérifier la connexion et la disponibilité du client. La lecture de la requête faite par le client est assurée par la méthode read(). Le contenu de la page Web est renvoyé au client par une longue série de println(). Les grandeurs physiques Pabs, temp2 … sont mises à jour lors de l'appel de la fonction acquisition().

La balise meta permet de rafraichir la page toutes les 5 secondes

<meta http-equiv=\"refresh\" content=\"5\">

```
#include <ESP8266WiFi.h>
IPAddress    apIP(192, 168, 4, 1);  // Defining a static IP address: local & gateway
 // Default IP in AP mode is 192.168.4.1
/* This are the WiFi access point settings. Update them to your likin */
const char *ssid = "ESP8266";
const char *password = "ESP8266Test";
WiFiServer server(80);  //Démarage du Wifi
void setup()
{
WiFi.softAPConfig(apIP, apIP, IPAddress(255, 255, 255, 0));   // subnet FF FF FF 00
WiFi.softAP(ssid);
  IPAddress myIP = WiFi.softAPIP();
// demarrage serveur web
  server.begin();
```

```
   delay(1000);
  // affichage adresse ip locale
  //Serial.println(WiFi.localIP());
}



void loop() {
WiFiClient client = server.available();
  if (client) {
   Serial.println("detection dune nouvelle demande client web .....");
   // bolean to locate when the http request ends
   boolean blank_line = true;
   while (client.connected()) {                    //Gestion des demandes de connexion au serveur
    if (client.available()) {
     char c = client.read();
     if (c == '\n' && blank_line) {
                                                  // envoi des données au client web
       client.println("HTTP/1.1 200 OK");
       client.println("Content-Type: text/html");
       client.println("Connection: close");
       client.println();
       // your actual web page that displays temperature and humidity
       client.println("<!DOCTYPE HTML>");
       client.println("<html><head><meta http-equiv=\"refresh\" content=\"5\"><meta charset=\"utf-
8\"><meta http-equiv=\"X-UA-Compatible\" content=\"IE=edge\"><title>Alarme E&R</title>");
       client.println("</head><body><div style=\"background-color:bisque\"><div style=\"background-color:
grey; color:white; text-align: center\"><p><font face = \"Calibri\">Surveillance</font></p></div><h1
style=\"text-align: center\"><font face = \"Calibri\" size = 10>SURVEILLANCE PAC</font></h1><h3><font face
= \"Calibri\">temperature soufflage : ");
       client.println(temp2);
       client.println("°c</font></h3><h3><font face = \"Calibri\">Puissance electrique: ");
       client.println(Pabs);
```

```
        client.println(" W</font></h3><h3><font face = \"Calibri\">temperature reprise : ");
        client.println(temp1);
        client.println("°C</font></h3><h3><font face = \"Calibri\">puissance calorifique : ");
        client.println(Pcalorifique);
         client.println("w</font></h3><h3><font face = \"Calibri\">Rendement : ");
        client.println(rendement);
        client.println(" </font></h3>");
        client.println("<div style=\"background-color: gray; color:white; text-align: center\"><p><font face =
\"Calibri\">Site web ESP8266 by BESSEL ©</font></p></div></div></body></html>");
         break;
       }

      if (c == '\n') {
                                                // premiere ligne a blanc
        blank_line = true;
      }
      else if (c != '\r') {
                                               // lignes suivantes avec du contenu
        blank_line = false;
      }
    }
  }
```

Mise en place d'une page web statique non réactualisé sur le module ESP32

La librairie WiFi.h pour l'ESP 32 doit être inclus pour mettre en place le serveur.

L'appel du constructeur wifiServer permet de mettre en place un serveur WEB WIFI sur le port 80

L'appel du constructeur apIP permet de mettre en place un objet point d'accès Wifi sur l'@ 192.168.4.1.

Les deux méthodes WiFi.softAPConfig et WiFi.softAP permettent de configurer le point d'accès avec le SSID, l'adresse IP, le masque ….

Le serveur est démarré du moment que la méthode wifiServer.begin(); est exécutée

La méthode LOOP récupère un objet client lors de la connexion de celui-ci WiFiClient client = wifiServer.available();

Les deux méthodes connected() et available() permettent de vérifier la connexion et la disponibilité du client. La lecture de la requête faite par le client est assurée par la méthode read(). Le contenu de la page Web est renvoyé au client par une longue série de println(). Les grandeurs physiques Pabs, temp2 … sont mises à jour lors de l'appel de la fonction acquisition().

```
#include <WiFi.h>
IPAddress    apIP(192, 168, 4, 1);
const char *ssid = "MyESP32AP";
const char *password = "testpassword";
iFiServer wifiServer(80);

void setup() {

  delay(1000);

  WiFi.softAPConfig(apIP, apIP, IPAddress(255, 255, 255, 0));   // subnet FF FF FF 00

 // You can remove the password parameter if you want the AP to be open.
 // WiFi.softAP(ssid, password);
 WiFi.softAP(ssid);
  IPAddress myIP = WiFi.softAPIP();
  Serial.print("\n AP IP address: ");
```

```
  Serial.println(myIP);
  Serial.println('\n');
   wifiServer.begin();
}




void loop() {

  WiFiClient client = wifiServer.available();

  if (client) {
  boolean blank_line = true;
    while (client.connected()) {

      if (client.available()) {
        char c = client.read();

       if (c == '\n' && blank_line) {
acquisition();

                                                  // envoi des données au client web
         client.println("HTTP/1.1 200 OK");
         client.println("Content-Type: text/html");
         client.println("Connection: close");
         client.println();
         // your actual web page that displays temperature and humidity
         client.println("<!DOCTYPE HTML>");
         client.println("<html><head><meta http-equiv=\"refresh\" content=\"5\"><meta charset=\"utf-
8\"><meta http-equiv=\"X-UA-Compatible\" content=\"IE=edge\"><title>Alarme E&R</title>");
```

```
client.println("</head><body><div style=\"background-color:bisque\"><div style=\"background-color:
grey; color:white; text-align: center\"><p><font face = \"Calibri\">Alarme Courant E&R</font></p></div><h1
style=\"text-align: center\"><font face = \"Calibri\" size = 10>SURVEILLANCE PAC</font></h1><h3><font face
= \"Calibri\">temperature soufflage : ");
client.println(temp2);
client.println("°c</font></h3><h3><font face = \"Calibri\">Puissance electrique: ");
client.println(Pabs);
client.println(" W</font></h3><h3><font face = \"Calibri\">temperature reprise : ");
client.println(temp1);
client.println("°C</font></h3><h3><font face = \"Calibri\">puissance calorifique : ");
client.println(Pcalorifique);
client.println("°C</font></h3><h3><font face = \"Calibri\">Rendement : ");
client.println(rendement);
client.println(" </font></h3>");
client.println("<div style=\"background-color: gray; color:white; text-align: center\"><p><font face =
\"Calibri\">Site web ESP32 by BESSEL ©</font></p></div></div></body></html>");
break;
}

if (c == '\n') {
                                        // premiere ligne a blanc
blank_line = true;
}
else if (c != '\r') {
                                        // lignes suivantes avec du contenu
blank_line = false;
}
}
}
// fermeture connexion client
delay(1);
client.stop();
Serial.println("client web deconnecte _____");}}
```

Mise en place d'une page WEB dynamique sur le module ESP32

La librairie AsyncTCP.h, pour l'ESP32 et ESPAsyncWebServer.h doit être inclus pour mettre en place le serveur.

L'appel du constructeur Server(80) permet de mettre en place un serveur WEB WIFI sur le port 80

L'appel du constructeur apIP permet de mettre en place un objet point d'accès Wifi sur l'@ 192.168.4.1.

Les deux méthodes WiFi.softAPConfig et WiFi.softAP permettent de configurer le point d'accès avec le SSID, l'adresse IP, le masque ….

Le serveur est démarré du moment que la méthode wifiServer.begin() est exécutée

La méthode LOOP récupère un objet client lors de la connexion de celui-ci WiFiClient client = wifiServer.available();

Les trois fonctions suivantes permettent de répondre à la méthode web get , la méthode web post et aucune méthode :

handleRoot(AsyncWebServerRequest *request) sensible à n'importe quelle requête get

handleLogin(AsyncWebServerRequest *request) sensible à la requête post

handleNotFound(AsyncWebServerRequest *request)

La méthode on() de l'objet server permettent d'inscrire l'appel de ces méthodes lorsque le client ou le navigateur demande une ressource par un get ou un post.

```
#include <WiFi.h>
#include <AsyncTCP.h>
#include <ESPAsyncWebServer.h>

//ESP8266WiFiMulti wifiMulti;     // Create an instance of the ESP8266WiFiMulti class, called 'wifiMulti'
IPAddress    apIP(192, 168, 4, 1);  // Defining a static IP address: local & gateway
                                    // Default IP in AP mode is 192.168.4.1

/* This are the WiFi access point settings. Update them to your likin */
const char *ssid = "MyESP32AP";
const char *password = "testpassword";

AsyncWebServer server(80);    // Create a webserver object that listens for HTTP request on port 80

void handleRoot();            // function prototypes for HTTP handlers
```

```
void handleLogin();
void handleNotFound();

void setup(void){
 Serial.begin(115200);        // Start the Serial communication to send messages to the computer
 Serial.println('\n');

 //set-up the custom IP address
 WiFi.softAPConfig(apIP, apIP, IPAddress(255, 255, 255, 0));   // subnet FF FF FF 00

// You can remove the password parameter if you want the AP to be open.
// WiFi.softAP(ssid, password);
WiFi.softAP(ssid);
 IPAddress myIP = WiFi.softAPIP();

server.on("/", HTTP_GET,[](AsyncWebServerRequest *request){ handleRoot(request); });
// Call the 'handleRoot' function when a client requests URI "/"

 server.on("/login", HTTP_POST,[](AsyncWebServerRequest *request){ handleLogin(request);
 }); // Call the 'handleLogin' function when a POST request is made to URI "/login"

 server.onNotFound([](AsyncWebServerRequest *request){handleNotFound(request);
 });// When a client requests an unknown URI (i.e. something other than "/"), call function "handleNotFound"

server.begin();                    // Actually start the server

void loop(void){
 // server.handleClient();              // Listen for HTTP requests from clients
}
```

```
void handleRoot(AsyncWebServerRequest *request) {
 request->send(200, "text/html", "<form action=\"/login\" method=\"POST\"><input type=\"text\" name=\"username\" placeholder=\"Username\"></br><input type=\"password\" name=\"password\" placeholder=\"Password\"></br><input type=\"submit\" value=\"Login\"></form><p>Try 'John Doe' and 'password123' ...</p>");
}
```

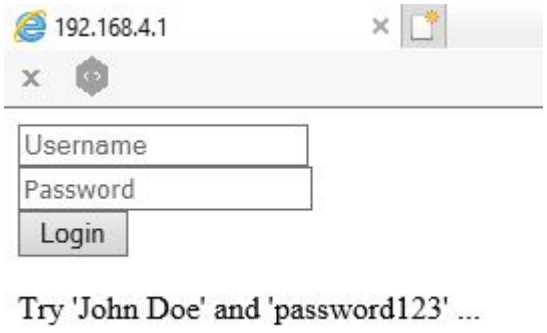
```
void handleLogin(AsyncWebServerRequest *request) {                    // If a POST request is made to URI /login
 if( ! request->hasArg("username") || ! request->hasArg("password")
    || request->arg("username") == NULL || request->arg("password") == NULL) { // If the POST request doesn't have username and password data
   request->send(400, "text/plain", "400: Invalid Request");        // The request is invalid, so send HTTP status 400
   return;
 }
 if(request->arg("username") == "John Doe" && request->arg("password") == "password123") { // If both the username and the password are correct
   request->send(200, "text/html", "<h1>Welcome, " + request->arg("username") + "!</h1><p>Login successful</p>");
 } else {                                                              // Username and password don't match
   request->send(401, "text/plain", "401: Unauthorized");
 }
}
```

```
void handleNotFound(AsyncWebServerRequest *request){
 request->send(404, "text/plain", "404: Not found"); // Send HTTP status 404 (Not Found) when there's no handler for the URI in the request
}
```

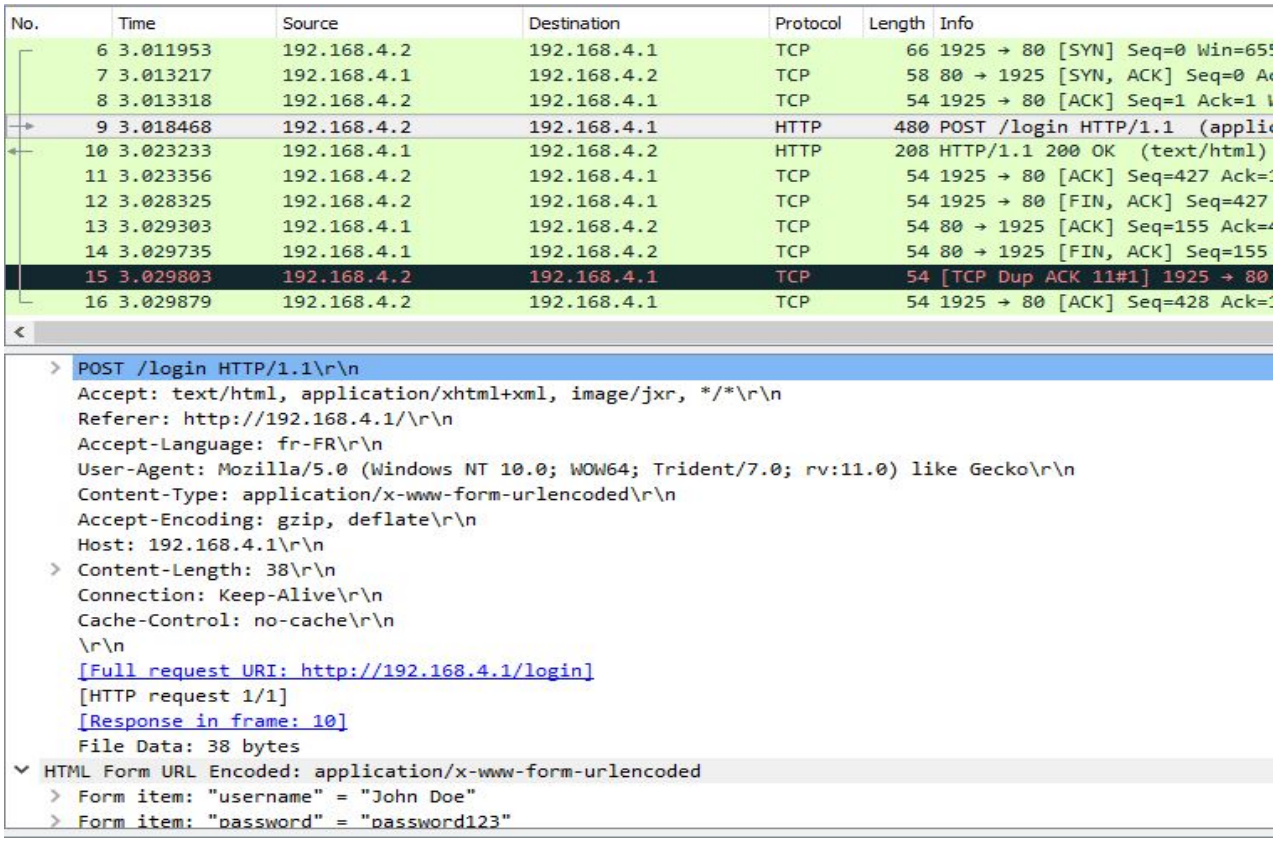
Lors de la demande de la page web par le navigateur la méthode get est envoyée pour la requête

18	7.959241	192.168.4.2	192.168.4.1	HTTP	312 GET / HTTP/1.1
19	7.963115	192.168.4.1	192.168.4.2	HTTP	410 HTTP/1.1 200 OK (text/
20	7.963180	192.168.4.2	192.168.4.1	TCP	54 1874 → 80 [ACK] Seq=259
21	7.968781	192.168.4.2	192.168.4.1	TCP	54 1874 → 80 [FIN, ACK] Se
22	7.969785	192.168.4.1	192.168.4.2	TCP	54 80 → 1874 [ACK] Seq=357
23	7.970053	192.168.4.1	192.168.4.2	TCP	54 80 → 1874 [FIN, ACK] Se
24	7.970129	192.168.4.2	192.168.4.1	TCP	54 [TCP Dup ACK 20#1] 1874
25	7.970187	192.168.4.2	192.168.4.1	TCP	54 1874 → 80 [ACK] Seq=260
58	26.470926	192.168.4.2	192.168.4.1	DNS	90 Standard query 0xad81 A
59	26.472100	192.168.4.1	192.168.4.2	ICMP	70 Destination unreachable
60	26.472464	192.168.4.2	192.168.4.1	DNS	90 Standard query 0xad81 A
61	26.473671	192.168.4.1	192.168.4.2	ICMP	70 Destination unreachable
62	26.474026	192.168.4.2	192.168.4.1	DNS	90 Standard query 0xad81 A
63	26.475241	192.168.4.1	192.168.4.2	ICMP	70 Destination unreachable

```
> Frame 18: 312 bytes on wire (2496 bits), 312 bytes captured (2496 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: 1874, Dst Port: 80, Seq: 1, Ack: 1, Len: 258
v 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
```



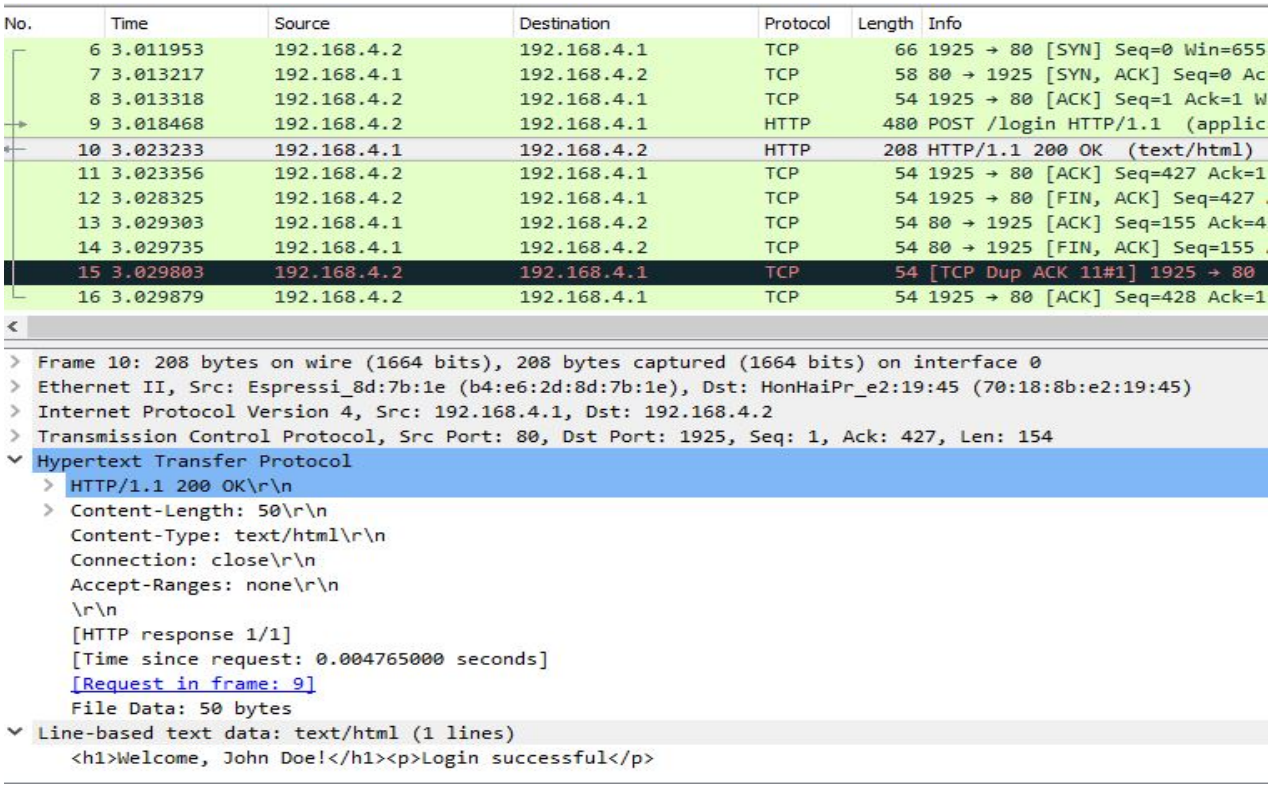
Quand on appuie sur le bouton la méthode post /login est envoyée pour la requête



No.	Time	Source	Destination	Protocol	Length	Info
6	3.011953	192.168.4.2	192.168.4.1	TCP	66	1925 → 80 [SYN] Seq=0 Win=65
7	3.013217	192.168.4.1	192.168.4.2	TCP	58	80 → 1925 [SYN, ACK] Seq=0 A
8	3.013318	192.168.4.2	192.168.4.1	TCP	54	1925 → 80 [ACK] Seq=1 Ack=1
9	3.018468	192.168.4.2	192.168.4.1	HTTP	480	POST /login HTTP/1.1 (appli
10	3.023233	192.168.4.1	192.168.4.2	HTTP	208	HTTP/1.1 200 OK (text/html)
11	3.023356	192.168.4.2	192.168.4.1	TCP	54	1925 → 80 [ACK] Seq=427 Ack=
12	3.028325	192.168.4.2	192.168.4.1	TCP	54	1925 → 80 [FIN, ACK] Seq=427
13	3.029303	192.168.4.1	192.168.4.2	TCP	54	80 → 1925 [ACK] Seq=155 Ack=
14	3.029735	192.168.4.1	192.168.4.2	TCP	54	80 → 1925 [FIN, ACK] Seq=155
15	3.029803	192.168.4.2	192.168.4.1	TCP	54	[TCP Dup ACK 11#1] 1925 → 80
16	3.029879	192.168.4.2	192.168.4.1	TCP	54	1925 → 80 [ACK] Seq=428 Ack=

```
> POST /login 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: 38\r\n
  Connection: Keep-Alive\r\n
  Cache-Control: no-cache\r\n
  \r\n
  [Full request URI: http://192.168.4.1/login]
  [HTTP request 1/1]
  [Response in frame: 10]
  File Data: 38 bytes
v HTML Form URL Encoded: application/x-www-form-urlencoded
  > Form item: "username" = "John Doe"
  > Form item: "password" = "password123"
```

La réponse du serveur est la suivante :



No.	Time	Source	Destination	Protocol	Length	Info
6	3.011953	192.168.4.2	192.168.4.1	TCP	66	1925 → 80 [SYN] Seq=0 Win=655
7	3.013217	192.168.4.1	192.168.4.2	TCP	58	80 → 1925 [SYN, ACK] Seq=0 Ac
8	3.013318	192.168.4.2	192.168.4.1	TCP	54	1925 → 80 [ACK] Seq=1 Ack=1 W
9	3.018468	192.168.4.2	192.168.4.1	HTTP	480	POST /login HTTP/1.1 (applic
10	3.023233	192.168.4.1	192.168.4.2	HTTP	208	HTTP/1.1 200 OK (text/html)
11	3.023356	192.168.4.2	192.168.4.1	TCP	54	1925 → 80 [ACK] Seq=427 Ack=1
12	3.028325	192.168.4.2	192.168.4.1	TCP	54	1925 → 80 [FIN, ACK] Seq=427
13	3.029303	192.168.4.1	192.168.4.2	TCP	54	80 → 1925 [ACK] Seq=155 Ack=4
14	3.029735	192.168.4.1	192.168.4.2	TCP	54	80 → 1925 [FIN, ACK] Seq=155
15	3.029803	192.168.4.2	192.168.4.1	TCP	54	[TCP Dup ACK 11#1] 1925 → 80
16	3.029879	192.168.4.2	192.168.4.1	TCP	54	1925 → 80 [ACK] Seq=428 Ack=1

```
> Frame 10: 208 bytes on wire (1664 bits), 208 bytes captured (1664 bits) on interface 0
> Ethernet II, Src: Espressi_8d:7b:1e (b4:e6:2d:8d:7b:1e), Dst: HonHaiPr_e2:19:45 (70:18:8b:e2:19:45)
> Internet Protocol Version 4, Src: 192.168.4.1, Dst: 192.168.4.2
> Transmission Control Protocol, Src Port: 80, Dst Port: 1925, Seq: 1, Ack: 427, Len: 154
v Hypertext Transfer Protocol
  > HTTP/1.1 200 OK\r\n
  > Content-Length: 50\r\n
    Content-Type: text/html\r\n
    Connection: close\r\n
    Accept-Ranges: none\r\n
    \r\n
    [HTTP response 1/1]
    [Time since request: 0.004765000 seconds]
    [Request in frame: 9]
    File Data: 50 bytes
v Line-based text data: text/html (1 lines)
    <h1>Welcome, John Doe!</h1><p>Login successful</p>
```



Welcome, John Doe!

Login successful

Mise en place d'une page dynamique sur le module ESP8266

```
#include <ESP8266WiFi.h>
#include <WiFiClient.h>
#include <ESP8266WiFiMulti.h>
#include <ESP8266mDNS.h>
#include <ESP8266WebServer.h>

ESP8266WiFiMulti wifiMulti;     // Create an instance of the ESP8266WiFiMulti class, called 'wifiMulti'
IPAddress    apIP(192, 168, 4, 1);  // Defining a static IP address: local & gateway
                        // Default IP in AP mode is 192.168.4.1

/* This are the WiFi access point settings. Update them to your likin */
const char *ssid = "ESP8266";
const char *password = "ESP8266Test";

ESP8266WebServer server(80);    // Create a webserver object that listens for HTTP request on port 80

void handleRoot();              // function prototypes for HTTP handlers
void handleLogin();
void handleNotFound();

void setup(void){
  Serial.begin(9600);         // Start the Serial communication to send messages to the computer
  delay(10);
  Serial.println('\n');

  //set-up the custom IP address
   WiFi.mode(WIFI_AP);
  WiFi.softAPConfig(apIP, apIP, IPAddress(255, 255, 255, 0));   // subnet FF FF FF 00
```

```
// You can remove the password parameter if you want the AP to be open.
// WiFi.softAP(ssid, password);
WiFi.softAP(ssid);
 IPAddress myIP = WiFi.softAPIP();
 Serial.print("AP IP address: ");
 Serial.println(myIP);
 Serial.println('\n');
 Serial.print("Connected to ");
 Serial.println(WiFi.SSID());              // Tell us what network we're connected to
 Serial.print("IP address:\t");
 Serial.println(WiFi.localIP());           // Send the IP address of the ESP8266 to the computer


 server.on("/", HTTP_GET, handleRoot);       // Call the 'handleRoot' function when a client requests URI "/"
 server.on("/login", HTTP_POST, handleLogin); // Call the 'handleLogin' function when a POST request is made to URI "/login"
 server.onNotFound(handleNotFound);          // When a client requests an unknown URI (i.e. something other than "/"), call function "handleNotFound"

 server.begin();                           // Actually start the server
 Serial.println("HTTP server started");
}

void loop(void){
 server.handleClient();                    // Listen for HTTP requests from clients
}

void handleRoot() {                        // When URI / is requested, send a web page with a button to toggle the LED
 server.send(200, "text/html", "<form action=\"/login\" method=\"POST\"><input type=\"text\" name=\"username\" placeholder=\"Username\"></br><input type=\"password\" name=\"password\" placeholder=\"Password\"></br><input type=\"submit\" value=\"Login\"></form><p>Try 'John Doe' and 'password123' ...</p>");
}
```

```
void handleLogin() {                    // If a POST request is made to URI /login
  if( ! server.hasArg("username") || ! server.hasArg("password")
     || server.arg("username") == NULL || server.arg("password") == NULL) { // If the POST request doesn't have username and password data
    server.send(400, "text/plain", "400: Invalid Request");        // The request is invalid, so send HTTP status 400
    return;
  }
  if(server.arg("username") == "John Doe" && server.arg("password") == "password123") { // If both the username and the password are correct
    server.send(200, "text/html", "<h1>Welcome, " + server.arg("username") + "!</h1><p>Login successful</p>");
  } else {                                                  // Username and password don't match
    server.send(401, "text/plain", "401: Unauthorized");
  }
}
```

```
void handleNotFound(){
  server.send(404, "text/plain", "404: Not found"); // Send HTTP status 404 (Not Found) when there's no handler for the URI in the request
}
```

Bibliographie :

https://formation.upyupy.fr/html-xhtml/

https://developer.mozilla.org/fr/docs/Apprendre/HTML

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 : <span id="var1"></span> temperature <br>
```

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);
```

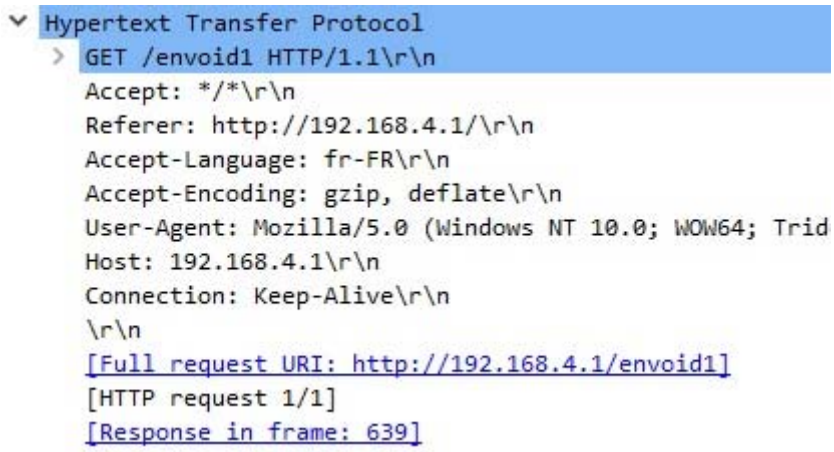
```
xhttp.send();
}
```

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

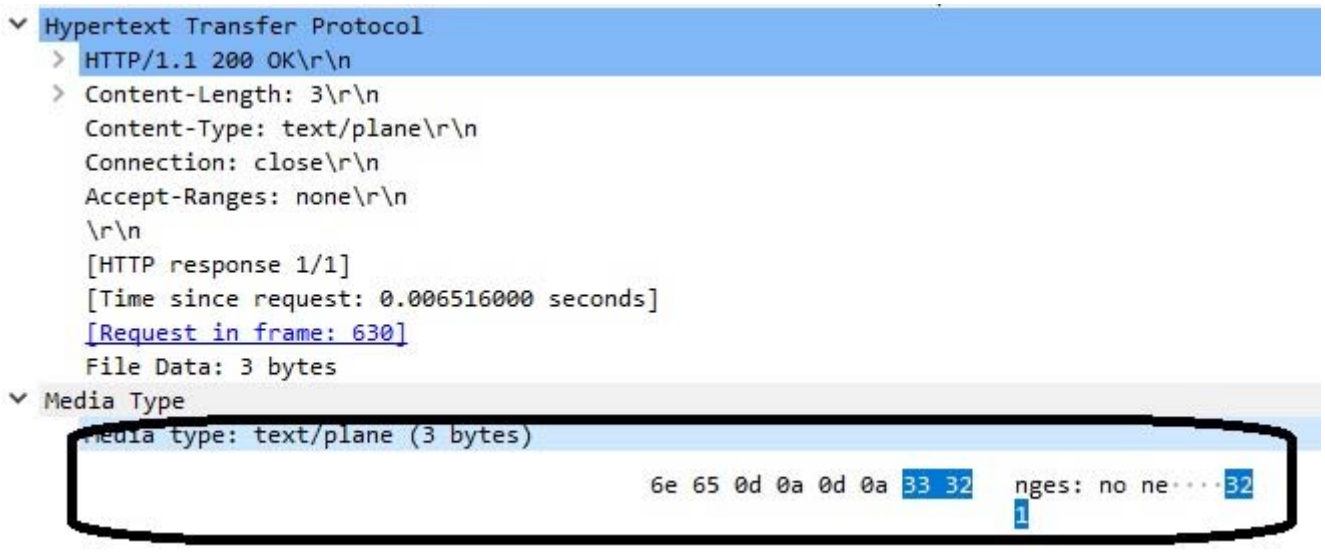
```
server.on("/envoiD1", HTTP_GET, [](AsyncWebServerRequest *request){
   request->send(200, "text/plane", eco2());
 });
```

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;
}
```

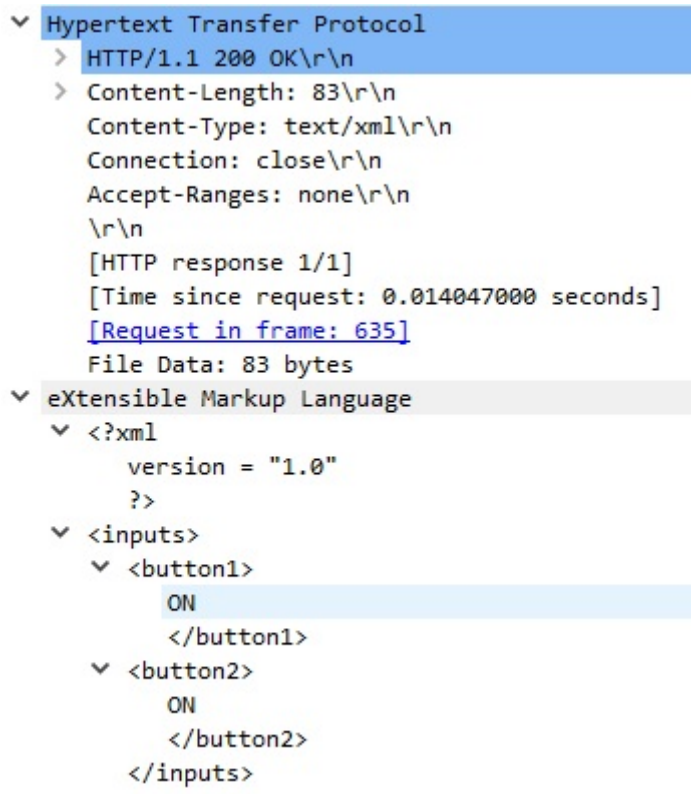
La fonction chaine retournera le contenu du texte formaté en xml avec deux variables button1 et button2.

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

Le client envoie la requête http_GET



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



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

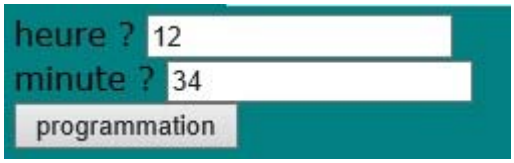
Ce qui après mise à jour nous donne les deux champs à ON et ON



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>
```

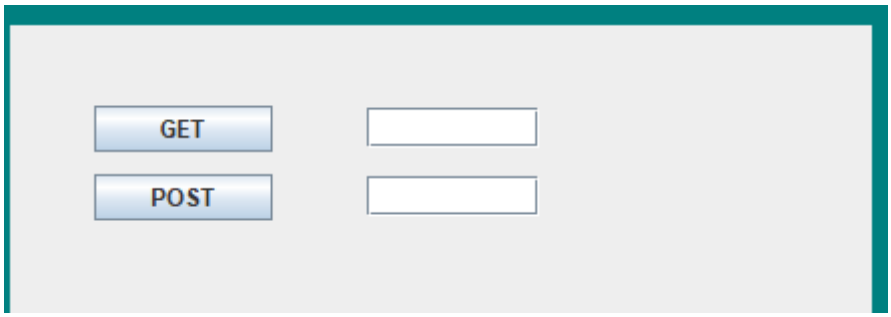
Du coté serveur web :

```
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

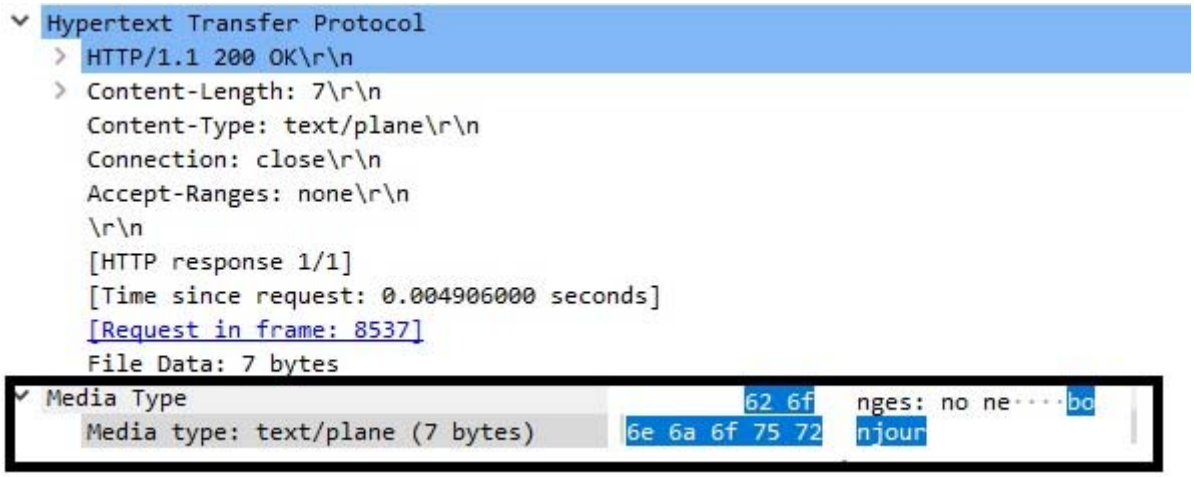


Lors de l'appuie sur le bouton POST, l'applet du client web lance une requête du type http_POST vers le serveur

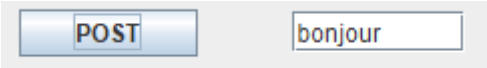
```
Hypertext Transfer Protocol
  POST /data1 HTTP/1.1\r\n
  User-Agent: Java client\r\n
  Content-Type: application/x-www-form-urlencoded\r\n
  Host: 192.168.4.1\r\n
  Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2\r\n
  Connection: keep-alive\r\n
  Content-Length: 31\r\n
  \r\n
  [Full 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



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