Programmation Réseau WEB

Protocole HTTP – (HTML/CSS/Javascript) – API Rest

Le protocole HTTP

Le protocole **HTTP** (« **H**yper**T**ext **T**ransfert **P**rotocol ») est un des protocoles les plus utilisés dans le monde car

- il permet de faire communiquer des machines totalement différentes (processeur, architecture, …)
- il permet de faire communiquer des processus développés dans des langages de programmation différents
- il est standardisé

Principales caractéristiques

- Protocole du type **« client-serveur »** travaillant au niveau de la couche application et utilisant un protocole sous-jacent fiable → **TCP** la plupart du temps
- Protocole orienté **« lignes de texte »** → donc facilement lisible par toute machine et tout langage de programmation
- Port par défaut : **80** (443 pour HTTPS)
- Protocole **sans état** [*stateless*] → une fois la transaction terminée, il ne reste aucune trace du client sur le serveur → un serveur HTTP ne peut donc pas savoir qu'une série de requêtes proviennent toutes du même client

Utilisations principales :

- Récupérer des **ressources** (fichiers, images, vidéos, pages web, …) sur une machine → serveurs et navigateurs web classiques
- Création d'**API** (Interfaces de Programmation d'Application) du type « **services web** » pour accéder aux données d'un serveur
- Réseaux sociaux, commerce électronique, applications mobiles, cloud, streaming vidéo, IoT, …

La notion d'URL

Une « **ressource** » est un terme désignant aussi bien des **données** (dans des formats divers) que des **services**. Ces ressources sont typiquement identifiées par un **URI** (**U**niform **R**esource **I**dentifier) dont la forme la plus classique est l'**URL** (**U**niform **R**esource **L**ocator)

Par exemples,

http://www.monsite.org/index.html

est une URL qui peut désigner un ensemble de données (soit une page web)

http://www.hepl.be/login?nom=wagner

pourrait être l'URL d'un service web permettant d'accéder aux services d'un serveur de la HEPL

Analysons plus précisément une URL à partir de cet exemple :

http://www.monsite.com:8008/users/connect?nom=wagner#hello

Cette URL est structurée ainsi :

- Le **protocole** « **http** » indiquant que la communication avec le serveur se fera via HTTP
- Le **nom d'hôte** (ou **Domaine**) **www.monsite.com** qui est l'adresse du serveur où la ressource est utilisée. C'est au serveur DNS de fournir l'adresse IP sous-jacente → on aurait pu avoir http://192.168.1.122:8008/... si on avait indiqué l'adresse IP du serveur directement
- Le **port 8008** qui indique sur quel port se fera la communication → par défaut, s'il n'est pas spécifié, le navigateur utilisera le port 80 pour HTTP
- Le **chemin** « **/users/connect** » qui indique l'emplacement de la ressource à l'intérieur du serveur → cela peut correspondre au fichier connect qui se trouve dans le répertoire users → mais ce n'est pas une obligation !
- La **query** (« **requête** ») « **?nom=wagner** » représente les paramètres de la requête → ici le paramètre « nom » est défini à « wagner » → ces paramètres peuvent être utilisés pour personnaliser la manière dont la ressource est récupérée / le service web est utilisé
- Le « **fragment** » (ou « **ancre** ») « **#hello** » spécifie un ancrage à l'intérieur de la ressource → par exemple cela indique au navigateur de faire défiler la page HTML

jusqu'à la section identifiée par « hello » → les ancres sont souvent utilisées pour créer des liens vers des sections spécifiques d'une page web.

En résumé, cette URL pointe vers la ressource « connect » située sur le serveur www.monsite.com sur le port 8008. De plus, la requête inclut le paramètre « nom » avec la valeur « wagner », et le fragment « hello » indique une ancre à l'intérieur de la ressource.

Analyse du protocole HTTP proprement dit

Le fait d'insérer une URL (par exemple www.perdu.com) dans la barre de recherche d'un navigateur

provoque l'envoi d'une **requête HTTP** du browser au serveur web www.perdu.com qui répond au browser via une **réponse HTTP** contenant (ici une page HTML) dont le contenu est

```
<html>
    <head>
        <title>Vous Etes Perdu ?</title>
    </head>
    <body>
        <h1>Perdu sur l'Internet ?</h1>
        <h2>Pas de panique, on va vous aider</h2>
        <strong>
            <pre>
                * <----- vous &ecirc;tes ici </pre></strong>
    </body>
</html>
```

et dont la structure sera analysée plus tard. Ce ficher est interprété par le browser afin d'obtenir le rendu que l'on voit ci-dessus.

Sous linux (en particulier sur la VM Oracle Linux fournie), la commande **http** permet facilement de visualiser la requête HTTP envoyée ainsi que la réponse HTTP reçue :

```
$ http www.perdu.com --verbose
GET / HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: www.perdu.com
User-Agent: HTTPie/2.6.0



HTTP/1.1 200 OK
Accept-Ranges: bytes
CF-Cache-Status: DYNAMIC
CF-RAY: 7ff517e45c16fa3c-AMS
Cache-Control: max-age=600
Connection: keep-alive
Content-Encoding: gzip
Content-Length: 163
Content-Type: text/html
Date: Thu, 31 Aug 2023 11:56:59 GMT
ETag: "cc-5344555136fe9-gzip"
Expires: Thu, 31 Aug 2023 12:06:59 GMT
Last-Modified: Thu, 02 Jun 2016 06:01:08 GMT
NEL: {"success_fraction":0,"report_to":"cf-nel","max_age":604800}
Report-To:
{"endpoints":[{"url":"https:\/\/a.nel.cloudflare.com\/report\/v3?s=dw7LZOrrFt
KwqdqsqAEXq8dtqVuZsaNBJUwXISeNsKBnkrDkW3GgpEMd11CESoilDLIQMG9i79iKoPlcvd5DaX4
rWWJ2vCxucCGODQJMo%2BwiLwPTLISfpUdD2PrwzGAk"}],"group":"cf-
nel","max_age":604800}
Server: cloudflare
Vary: Accept-Encoding,User-Agent
alt-svc: h3=":443"; ma=86400

<html><head><title>Vous Etes Perdu ?</title></head><body><h1>Perdu sur
l'Internet ?</h1><h2>Pas de panique, on va vous aider</h2><strong><pre>    *
<----- vous &ecirc;tes ici</pre></strong></body></html>

$
```

Il faut tout d'abord noter qu'une fois la réponse reçue par le client (ici la commande http), la transaction est terminée (protocole sans état)

La requête

Elle correspond aux 6 premières lignes fournies par commande http.

La première ligne (la plus importante)

```
GET / HTTP/1.1
```

Comporte

- Le **verbe de la commande**, ici « **GET** » (typiquement il s'agit des méthodes GET ou POST, mais d'autres existent) :
  - **GET** : cette méthode permet de retrouver une information de nature quelconque identifiée par son URI → dans le cas où la ressource est un service, ce seront les données produites par l'exécution de ce service qui constitueront la réponse → les paramètres de la requête sont dans ce cas directement fournies dans l'URL
  - **POST** : il s'agit ici de demander au serveur d'accepter les données contenues dans le message en tant que paramètres pour la ressource-service dont l'URL est précisée. Un exemple classique d'utilisation de POST est l'envoi de données via un formulaire HTML
  - HEAD, PUT, DELETE, …
- La **ressource** demandée, ici « **/** » c'est-à-dire ce qui se trouve à la racine du serveur visé
- La **version du protocole** utilisé, ici « **HTTP/1.1** »

Les lignes suivantes correspondent à des couples « clé/valeur » dont toutes ne sont pas nécessaires. Toutes ces lignes se terminent par les caractères « **\r\n** » :

- **User-Agent** : le nom du navigateur ou du client HTTP (ici c'était la commande http de linux)
- **Host** : spécifie l'hôte à qui s'adresse la requête GET
- **Accept** : spécifie les types de fichiers qui le client accepte de recevoir → ici « */* » spécifie que le client accepte tout type de fichier (voir plus loin)
- **Accept-Encoding** : spécifie les types d'encodage qui pourront être utilisés dans la réponse
- …

Ces 6 lignes constituent ce qu'on appelle le **« header » de la requête** :

- Ce header est ensuite suivi d'une **ligne vide « \r\n »**
- Vient ensuite le **corps de la requête** qui peut être n'importe quoi → dans l'exemple ci-dessus, la requête ne comporte aucun corps

La réponse

Tout comme pour la requête, la réponse contient

- Un **header**, suivi
- D'une **ligne vide « \r\n »**, suivi du
- **Corps de la réponse**

Le corps de la réponse est tout simplement ici le fichier HTML (voir plus haut) qui se trouve à la racine « / » du serveur.

Concernant le header, la première ligne

```
HTTP/1.1 200 OK
```

correspond au statut de la réponse du serveur :

- « **HTTP/1.1** » précise le **protocole** utilisé pour la réponse par le serveur
- « **200** » correspond au **code de retour du serveur** → 200 correspond à un succès
- Le **texte explicatif** « **OK** » qui correspond au code 200 et signifie que ce qui a été demandé a été envoyé

Les codes de retour du serveur correspondent selon leur premier digit aux catégories :

- **1xx** → simple information : le traitement de la requête se poursuit
- **2xx** → succès : la requête a été comprise et acceptée
- **3xx** → redirection : une autre action doit être entreprise par le user-agent pour compléter la réponse
- **4xx** → erreur du côté client : soit dans la syntaxe, soit dans la nature même de la demande
- **5xx** → Erreur du côté serveur : la requête semblait fondée mais le serveur n'a pas pu la satisfaire

HTTP comporte des dizaines de codes. Voici une liste des codes les plus utilisés :

- **200 : OK** → la page a bien été envoyée
- **201 : Created** → la ressource a été créée

- **301 : Moved permanently** → redirection définitive
- **303 : See Other** → redirection temporaire
- **400 : Bad request** → la requête n'est pas bien formée
- **403 : Forbidden** → l'utilisateur n'a pas accès à la ressource
- **404 : Not found** → la ressource n'est pas trouvée
- **405 : Method not allowed** → la méthode (GET, POST, …) est connue mais n'est pas compatible avec la ressource visée
- **411 : Length required** → la longueur est nécessaire mais pas donnée
- **500 : Internal Server Error** → erreur interne du serveur
- **501 : Not implemented** → la méthode n'est pas implémentée

Les lignes suivantes de la réponse correspondent à nouveau à des couples « clé/valeur » dont toutes ne sont pas nécessaires. Toutes ces lignes se terminent par les caractères « **\r\n** » :

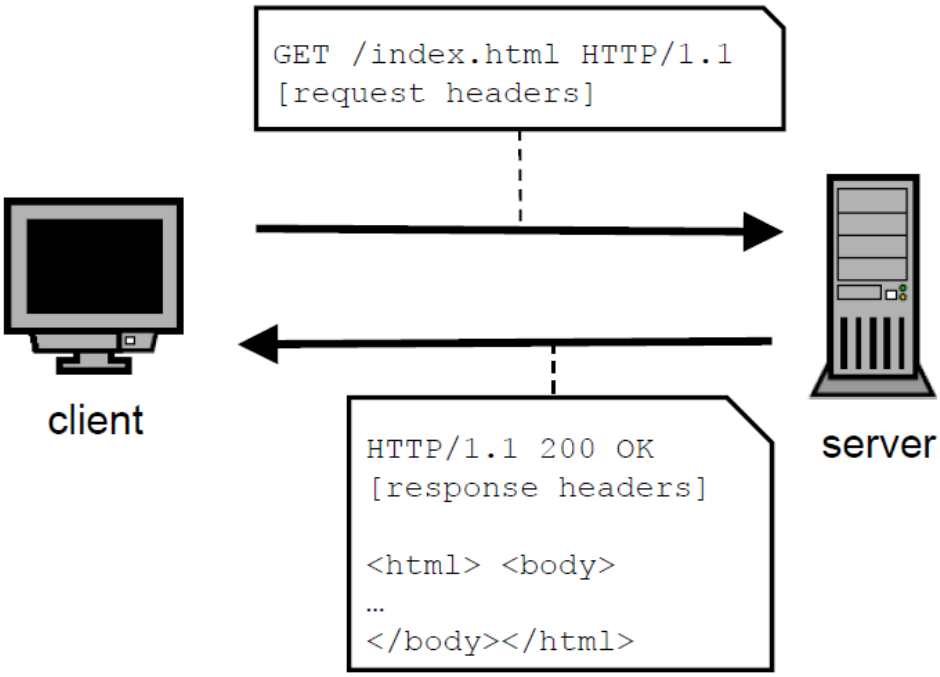
- **Content-Length** : **taille en bytes du corps de la réponse** (ce qui se trouve après le header)
- **Content-Type** : désigne le **type des données constituant le corps du message** → ici c'est « text/html » où « text » correspond au type principal tandis que « html » correspond au sous-type
- …

Voici un tableau reprenant les **types** et les **sous-types** les plus utilisés :

Type	**Définition**	**Sous-type**
text	Information textuelle	plain → « **text/plain** » texte non formaté ne nécessitant pas de logiciel particulier pour le lire
		Html → « **text/html** » Le texte comporte des tags html, c'est donc une page html
image	images	gif → « **image/gif** » jpeg → « **image/jpeg** »
application	Informations à priori binaires ou formatées, à priori destinée à être utilisée par une application particulière	octet-stream → « **application/octet-stream** » Données binaires brutes utilisé lorsque on ne connait pas le sous-type précisé dans la requête
		x-www-form-urlencoded → « **application/x-www-form-urlencoded** » utilisé lors de l'envoi d'une requête POST

		json → « **application/json** » Données au format Json
audio	Informations nécessitant un dispositif de son	mpeg → « **audio/mpeg** » pour un corps de réponse correspond à un fichier audio MP3
video	Données vidéo	MP4 → **« video/mp4** » Pour un corps de réponse correspondant à un fichier video MP4
…	…	…

Finalement et synthétiquement, une transaction HTTP est représentée par

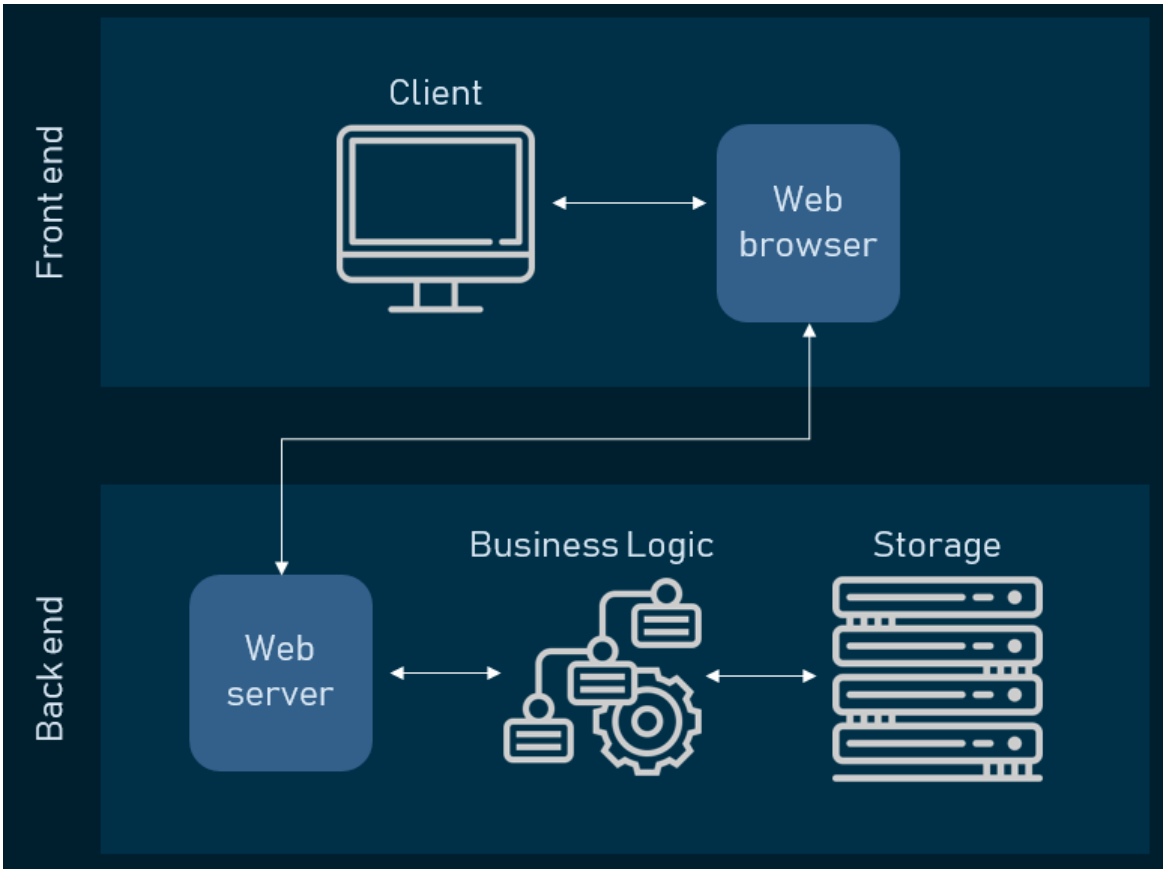


Source : Samuel Hiard

TO DO ? vue traffic reseau + cookies ?

Frontend / Backend

Les applications web et même la plupart des applications modernes utilisent les notions de **frontend** / **backend** :



source : https://www.altexsoft.com/blog/front-end-development-technologies-concepts/

Frontend (côté client)

- Il s'agit de la **partie visible** de l'application avec laquelle les utilisateurs interagissent directement → cela inclut l'interface utilisateur, les éléments graphiques, les boutons, les formulaires, …
- Les technologies de base utilisées sont principalement
  - **HTML** (**H**yper**T**ext **M**arkup **L**anguage) : langage de balisage utilisé pour créer la structure et le contenu des pages web
  - **CSS** (**C**ascading **S**tyle **S**heets) : utilisé pour la mise en page, le style et la présentation visuelle des pages web
  - **JavaScript** : langage de programmation principal pour ajouter de l'interactivité, de la logique et des fonctionnalités interactives dynamiques aux pages web

- Etant donné le succès actuel de cette architecture, de nombreux **frameworks** et bibliothèques **JavaScript** ont vu le jour. Au jour où ces lignes sont écrites :
  - **React** : développé par Facebook, il s'agit d'une bibliothèque JavaScript très populaire pour la création d'interfaces réactives et dynamiques → couramment utilisé pour le développement de **S**ingle-**P**age **A**pplications (**SPA**)
  - **Angular** : framework JavaScript complet développé par Google
  - **Vue.js** : framework JavaScript populaire qui se concentre sur la création d'interfaces utilisateur simples et réutilisables → connu pour sa facilité d'apprentissage
  - **Svelte** : framework JavaScript qui compile le code en code JavaScript pur au moment de la construction → peut entrainer des performances améliorées
- Remarquons également l'existance de **TypeScript** fortement utilisé par ces frameworks : il s'agit d'un sur-ensemble de JavaScript qui ajoute des fonctionnalités de typage statique au langage.

Backend (côté serveur)

- Il s'agit de la **partie invisible** de l'application qui gère la logique métier et la gestion des données
- Il répond aux demandes du frontend et effectue des opérations telles que l'authentification, la validation de données, l'accès à la base de données, …
- Les technologies couramment utilisées sont principalement
  - Des langages de programmation comme Python, Java (avec **Spring Boot**), Ruby, **Node.js**, …
  - Des frameworks comme Django (Python), Ruby on Rails, Express.js, …
  - **Microservices** et **conteneurs** (**Docker**, **Kubernetes**) : l'approche des microservices, combinée à des technologies de conteneurs comme Docker et Kubernetes, permet de développer des applications backend modulaires et hautement évolutives

Frontend et **backend** communiquent la plupart du temps via le **protocole HTTP**.

Introduction à HTML

Le langage **HTML** (**H**yper**T**ext **M**arkup **L**anguage)

- est un langage de balisage standard utilisé pour créer des pages web → une **balise** est une chaine de caractères commençant par « < » et se terminant par « > ». Exemples : <b>, , , <p>, …
- permet de créer la **structure** et le **contenu** d'une page web sous la forme d'un « arbre »

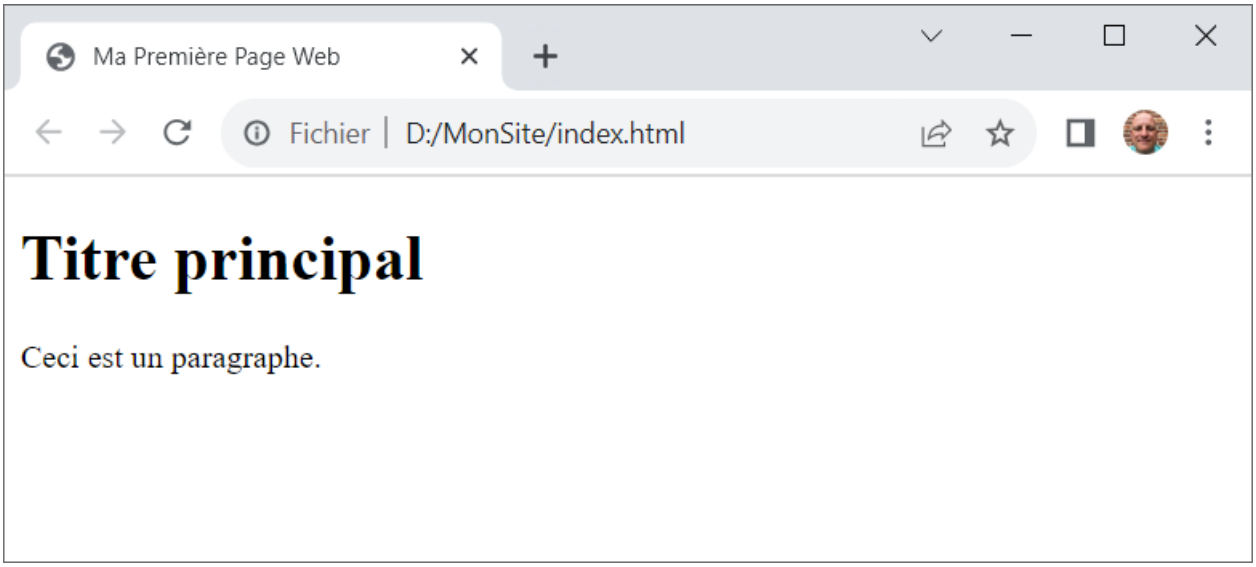
Un fichier HTML est un fichier texte dont la structure de base est la suivante :

```
<!DOCTYPE html>
<html>
    <head>
        <title>Ma Première Page Web</title>
    </head>
    <body>
        <h1>Titre principal</h1>
        <p>Ceci est un paragraphe.</p>
    </body>
</html>
```

où

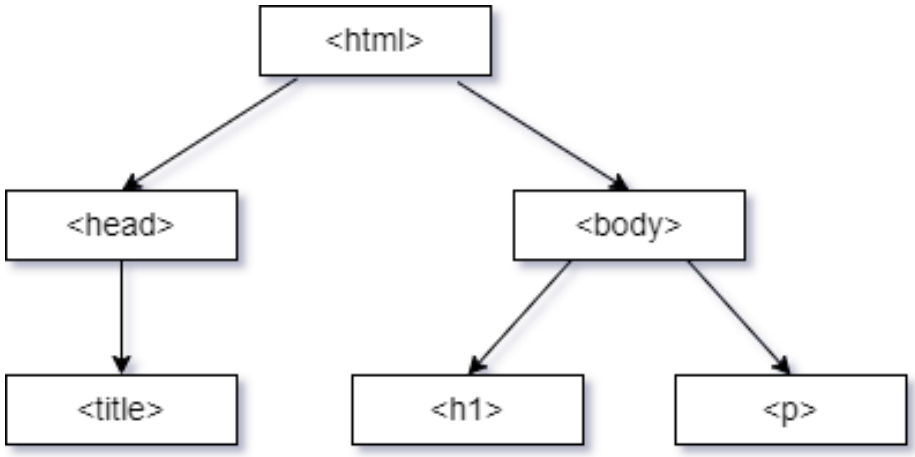
- Quand une balise est ouverte, elle devrait toujours être fermée (en ajoutant « / » en premier caractère de la balise) → la balise <html> est fermée par la balise </html>
- La balise **<html>** correspond à la racine d'un arbre décrivant la page web. Cette balise comporte 2 sous-balises (des « nœuds de l'arbre ») :
  - **<head>** : qui comporte actuellement le titre de la page mais qui pourra à terme contenir des métadonnées de la page web (charset, fichiers CSS, …)
  - **<body>** qui est le contenu principal de la page web, visible à l'écran
- La balise **<title>** correspond au titre de la page qui apparaitra dans l'onglet du browser
- La balise **<h1>** et **<p>** sont des exemples d'éléments HTML pour les titres et les paragraphes → il s'agit ici de « feuilles de l'arbre »
- La première ligne **< !DOCTYPE html>** définit la version d'HTML utilisée, ici il s'agit de HTML5

Si on nomme ce fichier **index.html**, son ouverture dans Chrome fournit



Le browser est donc capable d'interpréter le code de balisage HTML afin de fournir un rendu visuel.

Comme dit ci-dessus, cette page peut être représentée à l'aide d'un arbre :



qu'on appelle **arbre DOM** (**D**ocument **O**bject **M**odel) de la page HTML.

Le **DOM** est une interface de programmation qui permet à des scripts d'examiner et de modifier le contenu de la page web → Par le DOM, un document HTML (ou XML) est représenté sous forme d'un jeu d'objets (fenêtre, phrase, style, …) reliés selon une structure en arbre → à l'aide du DOM, un script pourra modifier le document présent dans le navigateur en ajoutant ou supprimant des nœuds/feuilles de l'arbre (voir plus loin).

Quelques balises de base

Pour les titres :

- **<h1></h1>** : titre 1er niveau
- **<h2></h2>** : titre 2ème niveau
- **<h3></h3>** : titre 3ème niveau (… et ainsi de suite jusqu'à 6)

Pour les éléments de texte :

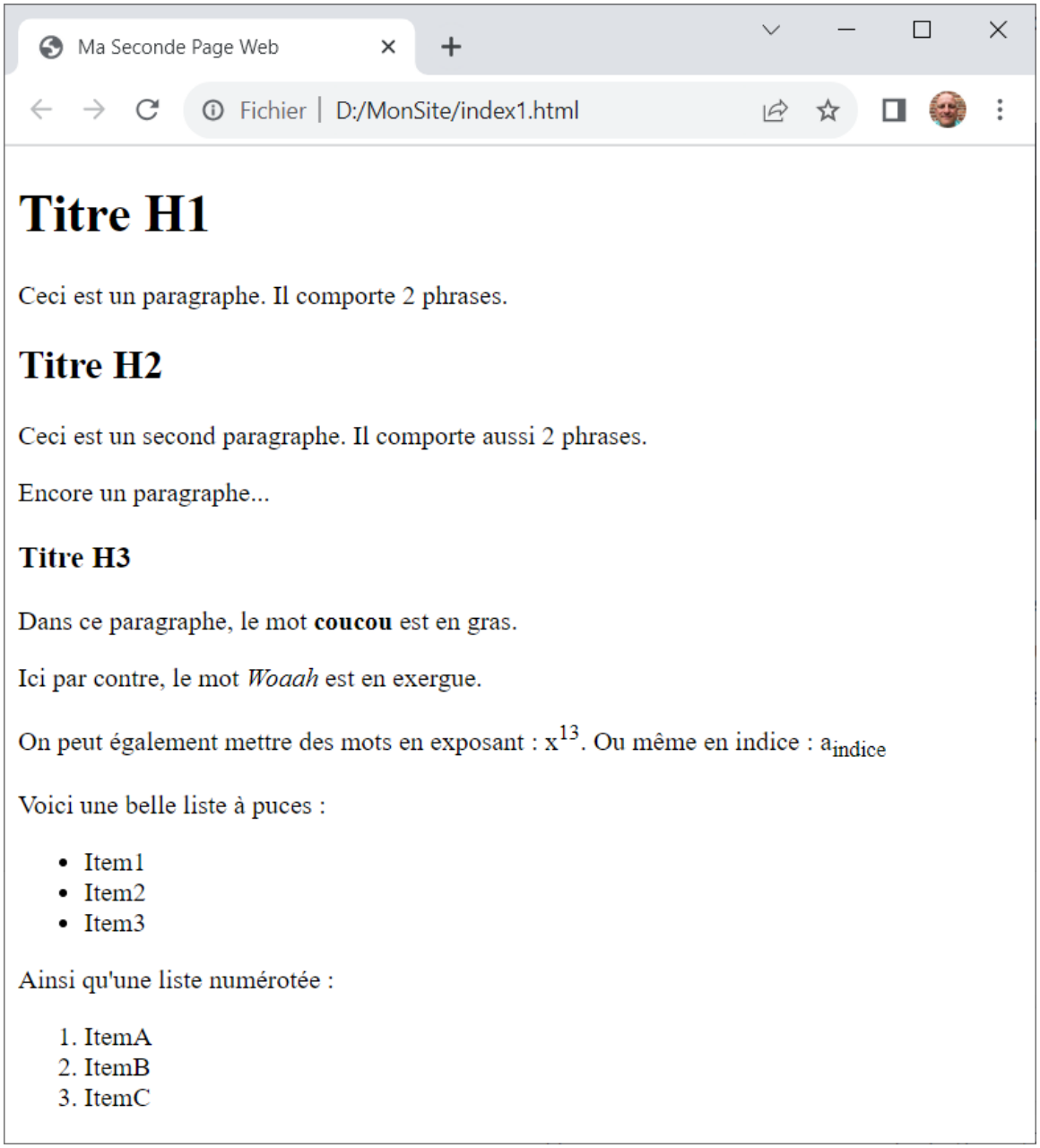
- ** ** : paragraphe
- **** : mettre en gras
- **** : mettre en exergue
- **** : mettre en exposant
- **** : mettre en indice
- **
** : liste à puces
- **
** : liste numérotée
- **
** : éléments d'une liste

Exemple : soit le fichier **index1.html** suivant

```
<!DOCTYPE html>
<html>
	<head>
		<title>Ma Seconde Page Web</title>
	</head>

	<body>
		<h1>Titre H1</h1>
		<p>Ceci est un paragraphe. Il comporte 2 phrases.</p>

		<h2>Titre H2</h2>
		<p>Ceci est un second paragraphe. Il comporte aussi 2 phrases.</p>
		<p>Encore un paragraphe...</p>

		<h3>Titre H3</h3>
		<p>Dans ce paragraphe, le mot <strong>coucou</strong> est en
gras.</p>
		<p>Ici par contre, le mot <em>Woaah</em> est en exergue.</p>
		<p>On peut également mettre des mots en exposant : x<sup>13</sup>.
Ou même en indice : a<sub>indice</sub></p>

		<p>
			Voici une belle liste à puces :
			<ul>
				<li>Item1</li>
```

```
                    <li>Item2</li>
                    <li>Item3</li>
                </ul>

                Ainsi qu'une liste numérotée :
                <ol>
                    <li>ItemA</li>
                    <li>ItemB</li>
                    <li>ItemC</li>
                </ol>
            </p>
        </body>
</html>
```

dont le rendu dans Google Chrome est :



On peut également structurer des données sous forme d'une table :

- **<table></table>** : créer un table
- **<tr></tr>** : ligne du tableau
- **<td></td>** : cellule du tableau (colonne)

Exemple : soit le fichier **index2.html** suivant

```
<!DOCTYPE html>
<html>
	<head>
		<title>Ma Page avec une table</title>
	</head>

	<body>
		<h1>Table exemple</h1>

		<table>
			<tr>
				<td>Nom</td>
				<td>Prénom</td>
				<td>Age</td>
			</tr>
			<tr>
				<td>Wagner</td>
				<td>Jean-Marc</td>
				<td>27</td>
			</tr>
			<tr>
				<td>Hiard</td>
				<td>Samuel</td>
				<td>76</td>
			</tr>
		</table>
	</body>
</html>
```

dont le rendu dans Google Chrome est

Le rendu n'est clairement pas esthétique… Pas de tracé de ligne pour délimiter les lignes ou les colonnes de la table. Néanmoins, ce n'est pas au code HTML de faire cela, c'est le fichier CSS (voir plus loin) qui s'en occupera.

Les liens

Il est possible de créer des **liens** entre une page HTML et d'autres ressources, comme une autre page HTML du même site, une page HTML d'un autre site, ou même vers une « **ancre** » de la même page HTML. Pour cela, on utilise la balise **<a>**.

On peut d'abord placer une « ancre » dans la page HTML même :

```
<a id="id unique"></a>
```

Ensuite, on peut créer des liens vers ces ancres ou vers d'autres pages :

```
<a href="#id unique"> Cliquez ici pour aller sur l'ancre ci-dessus</a>

<a href="URL"> Cliquez ici pour aller sur la page …</a>
```

id="id unique" et **href="URL"** s'appellent des **attributs de balises**

Exemple : soit le fichier **index3.html** suivant

```
<!DOCTYPE html>
<html>
	<head>
		<title>Ma Page avec des liens</title>
	</head>

	<body>
		<h1>Quelques liens à cliquer :</h1>

		<p> <a href="#retour"> Cliquez ici pour aller sur l'ancre ci-dessous</a> </p>
		<p> <a href="index.html"> Cliquez ici pour retourner sur la page index.html</a> </p>
		<p> <a href="http://www.google.be"> Cliquez ici pour aller sur la page de Google</a> </p>

		<a id="retour">Vous arriver ici...</a>

	</body>
</html>
```

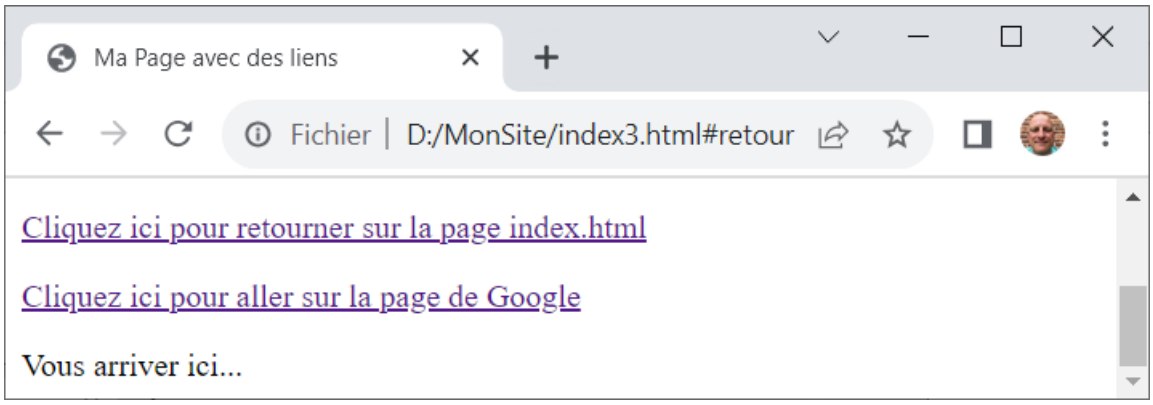
dont le rendu dans Google Chrome fournit

Le clic sur le 2ème lien fait apparaitre la page index.html (si celle-ci se trouve dans le répertoire de index3.html actuellement, vu que l'on travaille en local ici).

Le clic sur le 3ème lien nous redirige vers le site de Google.

Pour visualiser l'effet du clic sur le 1er lien, commençons par redimensionner la fenêtre :

Le clic sur le 1er lien fournit alors



Nous avons été redirigés vers la balise « **ancre** » mise en place dans la même page HTML → ceci se remarque également dans l'URL apparaissant dans Google Chrome → « **#retour** »

Les images

Pour inclure une image dans une page HTML, on utilise la balise **<img>**, dont la syntaxe est

```
<img att="valeur">
```

où **att** est un attribut de la balise pouvant prendre les valeurs :

- **src** : attribut obligatoire précisant où se trouve l'image qui peut être
  - un fichier local
  - une URL d'une image d'un autre site
  - le contenu d'une image en base64
- **alt** : attribut non obligatoire précisant le texte de remplacement au cas où l'image ne serait pas trouvée
- **width** : attribut non obligatoire permettant de fixer la largeur (en pixels) de l'image

Exemple : soit le fichier **index4.html** (source : S. Hiard que je remercie au passage) suivant

```
<!DOCTYPE html>
<html>
	<head>
		<title>Ma Page avec des images</title>
	</head>

	<body>
		<h1>Quelques images :</h1>

		Une image en base64 : <br/>
			<img
src="
BMVEX///8QX3X1+fnY6+oGWG/G3t8ATWYxo5P8/f0jnYzo8vKdy8mx2dUydYhYtKduvrNEq52Gwb5
4pbFWjp6S6lfVAAAD+0lEQVRYw62Y2YKDIAxFEQQSQAX9/3+dBPcW12leplPt4cINESPE/wPETwNA
SPghjCLE8AuVIwxAdYhRAvxEnx+EjGgtpjsSTwclZW3wxvTCE9CiujXpEySIDq2q6gSyRbT+2hcQS
p0g6XL0EDR90NipG7kD0I43wlHqLRfoLnlJBBHQoi0jYSaC1tMn5c+JAM5yHCAhNQ44mjr/Famq9S
mRbkE7I/UeCaAbtpe/CoEvgeiNGc6AvNJ2DsQ9kuZXV5XRQtLKidTx1EXTO7hwxNojJMk39aBcNQg
XEVkkyL1PXwL9lveJpN8PgZwdEkCiK6eskefsVyC2evs7GCmyDfRJdzJ/qU92wTkSNhuJ54Oe/+uN
v3TkExnG/BOO0gmkbhSDAtcbrhWNhmtHvpDAlcFiK8lt0+SsSZ0EuO9IAUkVoaU86eskYbFfDapAn
ffIBVJ5Vibd7A4lTl9Rcsu7jnwgYxgpuiEk1xyQjakrEx448sGMgW0IjaJHQMv7uTdVVVfuC5hL+h
0iRp/nFzBS0tAPm9pUvXvmyKdKL/MSRS3IIW8GVajD9klgJF+Vl65tuTh+b8A7juyJeSPzjuYC+10
```

```
3bzqyJeZnvIydKmQ2eR/t04hSJi/cmz1yIDHJuimXr6eOzKGU064M7PAFj6pEagEOHpzvAktPvFeO
zBrLAtNrgaXj4VtHcuaUBbY/FvjaEVs6vv7DEYu+KDC9F1hI6v84Ujxenz04L6Mo8L0jJYF0DHjtC
JWGEvC9I9PR679legMsCfyHI2WB4R8CC3WLjjLvBRbrlvPtWyQevdaqRGfvXwkcz2MydBbxVwLHJz
5N/SEyHr+Cwvj6px9NHf3Jewm/FD2depTHM5ZN0/MBnpnK35N5LJB4g6lNPTFJpr4l0x0A+UXL0GG
2WpjsULhKzsO+Bc0y82pCbplwlZzHbQvm1RlZ73WeOVQqC9POo/XL6qq6oFMdJuehQN0PDSHMgs0y
V2Y5OY8Fjiq1X7HVpHZlFhzC4w7DcjSGAnZhfjiE7UVXZT1wf2IX5s4h1NdtGu+DcgVsPc19WzmP6
tb+OUDDxzZ5raQoqDUjc5w6XvULlyYDZmyXgnYLVo7YemaGDqO8FNjtuyoTdl2DEdtwDZHcZbsS6H
EMu+fSGnTbNRBSBe+1vHQE6LaUuq5t2xhnjWvkpZ3WgF/db/Rw126MlM4ppbQO0yA0Styodfc6wjD
GUbnMo+QxQtDqSfsWQGt603LlWT1vBOcuGe86srMZKPo+Jcp4zcPQONNIcJ+npj3MYbZB39MIPZsD
TybcmAXESlkmiWSJd93YC+xNlee6MHKba53j6Bw8cETNOw7WVuEcb9v9OUukXLPoIekPRXxs+n14N
QQAAAAASUVORK5CYII=" alt="logo">

                <br/>Une image depuis le serveur local : <br/>
                <img src="petitchat.jpg" width=300 alt="un petit chat
mignon">

                <br/>Une image depuis internet : <br/>
                <img src="https://static.wikia.nocookie.net/half-
life/images/7/77/YoungAlyxVance.jpg/revision/latest?cb=20200323221922&path-
prefix=en" width=200 alt="Alyx Vance">

                <br/>Une image qui ne se charge pas bien : <br/>
                <img src="non" alt="Image introuvable">
    </body>
</html>
```

Quelques remarques :

- on remarque au passage l'utilisation de la balise **
** qui permet d'aller à la ligne
- la **base64** est un système de codage de données binaire en ASCII. Elle permet de représenter des données binaires (comme des fichiers ou des images) sous forme de texte → très utile lorsqu'il est nécessaire de stocker ou transmettre des données binaires dans des systèmes qui ne prennent en charge que du texte (comme par exemple les fichiers JSON, les URL, les fichiers HTML, …)

Le rendu dans Google Chrome est

où la dernière image (« non ») est introuvable et a donc été remplacée par le texte de remplacement (attribut **alt** de la balise).

Introduction à CSS

CSS (**C**ascading **S**tyle **S**heets ou **feuilles de style**) est

- un langage de programmation utilisé pour contrôler la présentation et le style des pages web
- utilisé en conjonction avec HTML pour définir la manière dont le contenu d'une page web (rôle de l'HTML) doit être affiché (rôle du CSS) dans le navigateur

Caractéristiques :

- CSS permet de séparer la structure (HTML) et la présentation (CSS) d'une page web. Cela signifie que l'on peut **modifier l'apparence** d'une page web **sans en modifier le contenu**
- CSS utilise des **sélecteurs** pour cibler les éléments HTML (tels que les balises <p> ou <h1> → il est ensuite possible de définir des propriétés telles que la couleur, la taille de police, les marges, …
- Le terme « **cascading** » signifie que les styles peuvent être appliqués en cascade, avec des règles plus spécifiques ayant la priorité sur des règles plus générales
- Les **styles CSS** peuvent être inclus dans une page web de différentes manières :
  - soit les inclure directement dans le fichier HTML (**style interne**),
  - soit les placer dans un **fichier CSS** externe lié à la page HTML (**style externe**),
  - soit les appliquer directement en utilisant des attributs HTML (**style en ligne**)

  → il est conseillé d'utiliser le style externe (fichier CSS séparé) afin de bien séparer contenu et présentation

Partons d'un exemple de page HTML (fichier **index.html**) vierge de tout style :

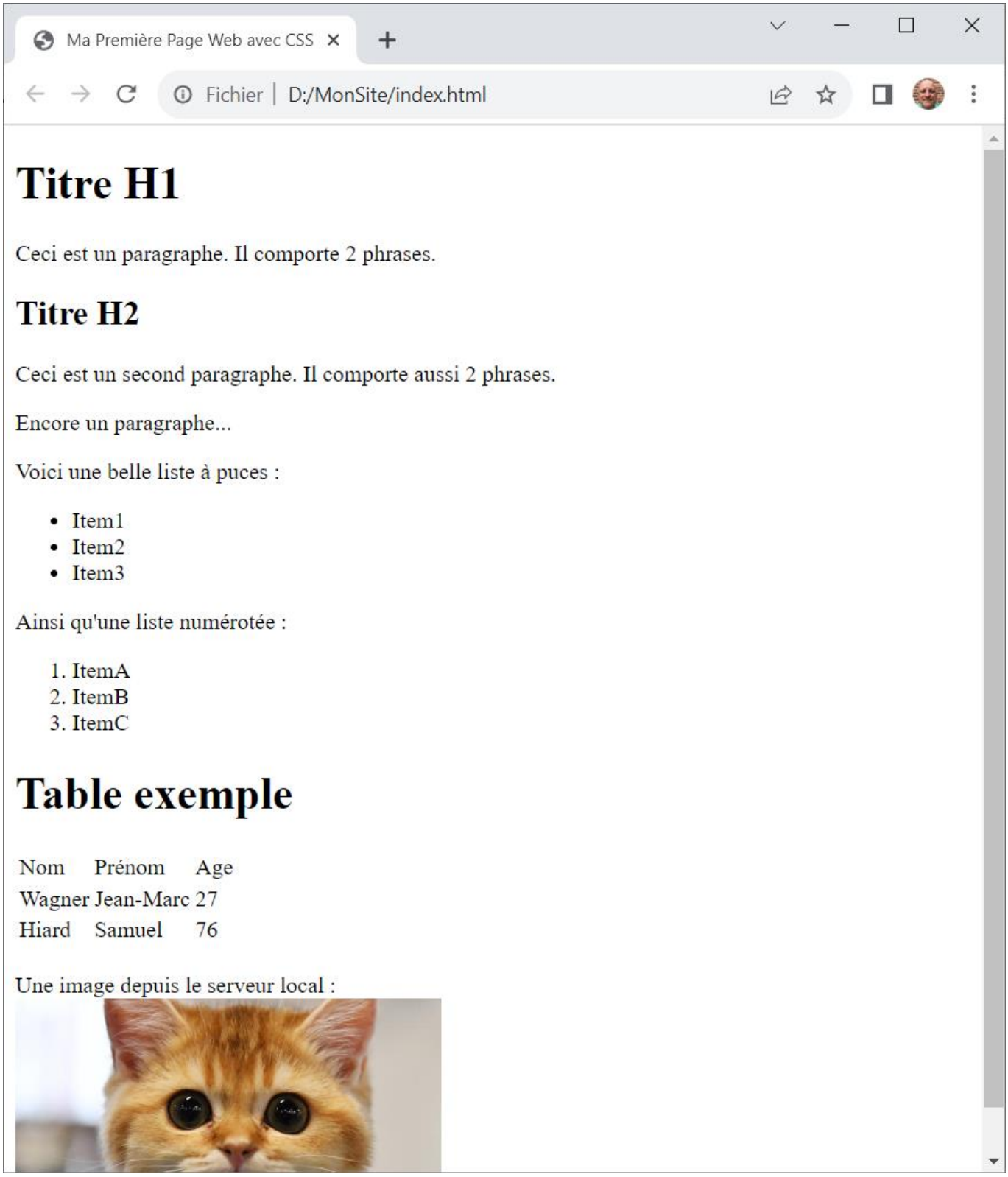
```
<!DOCTYPE html>
<html>
	<head>
		<title>Ma Première Page Web avec CSS</title>
	</head>

	<body>
		<div id="conteneur">

			<div id="bloc1">
			<h1>Titre H1</h1>
			<p>Ceci est un paragraphe. Il comporte 2 phrases.</p>
```

```
                <h2>Titre H2</h2>
                <p>Ceci est un second paragraphe. Il comporte aussi 2 phrases.</p>
                <p>Encore un paragraphe...</p>

                <p>
                      Voici une belle liste à puces :
                      <ul>
                                  <li>Item1</li>
                            <li>Item2</li>
                            <li>Item3</li>
                            </ul>

                      Ainsi qu'une liste numérotée :
                            <ol>
                            <li>ItemA</li>
                            <li>ItemB</li>
                            <li>ItemC</li>
                            </ol>
                </p>
                </div>

                <div id="bloc2">
                <h1>Table exemple</h1>

                <table>
                      <tr>
                                  <td>Nom</td>
                                  <td>Prénom</td>
                                  <td>Age</td>
                      </tr>
                      <tr>
                                  <td>Wagner</td>
                                  <td>Jean-Marc</td>
                                  <td>27</td>
                      </tr>
                      <tr>
                                  <td>Hiard</td>
                                  <td>Samuel</td>
                                  <td>76</td>
                      </tr>
                </table>
                </div>

                <div id="bloc3">
                <br/>Une image depuis le serveur local : <br/>
                <img src="images/petitchat.jpg" width=300 alt="un petit chat
mignon">
                </div>

            </div>
      </body>
</html>
```

On remarque :

- la présence des différentes balises déjà abordée précédemment, ainsi que la présence d'une table et d'une image
- la présence d'une nouvelle balise ** ** (pour **« division »**) :
    - c'est une balise de conteneur générique utilisée pour regrouper et structurer des éléments HTML → à priori elle n'apporte rien au contenu de la page à part que plusieurs éléments HTML sont regroupés afin d'être traités simultanément (voir plus loin)
    - elle est essentiellement utilisée à des fins de mise en page → cela va permettre de créer des mises en page complexes en divisant la page web en sections ou colonnes (un peu comme les **layout en Java**)
- dans notre exemple, on remarque une balise principale ** ** qui englobe 3 « sous-balises » ** **
- A chacune de ces balises ** ,** nous avons ajouté un **attribut « id=… »** → cela permet d'identifier un bloc (ou tout autre élément HTML) par un identifiant dont nous pouvons choisir le nom → ces identifiants seront utilisés par CSS (et par Javascript → voir plus tard)

Voici le rendu de cette page HTML dans Google Chrome :



On remarque que les 3 « blocs » sont positionnés les uns à la suite des autres sans aucune mise en forme de couleur ou de style particulier.

Afin de limiter la taille de cette introduction à CSS, nous allons ici nous limiter au « **style externe** » et construire un fichier CSS séparé. Voici un exemple de fichier CSS (fichier **style.css** que nous compléterons par la suite) :

```
#bloc1 {
      background-color: lightblue;
      color: black;
      font-size: 12px;
}

#bloc2 {
      background-color: lightseagreen;
      color: yellow;
}

#bloc3 {
      background-color: lightsalmon;
      color: blue;
      border: 2px solid black;
}
```

Dans ce fichier :

- **#bloc1**, **#bloc2**, **#bloc3** sont appelés des **sélecteurs d'identifiant** → ils ciblent tous les éléments HTML ayant bloc1, bloc2 ou bloc3 comme identifiant → l'effet cascade va se faire sentir ici → tous les éléments HTML situés dans les balises correspondantes vont « hériter » des propriétés de style définies ici
- A l'intérieur de ces « sélecteurs », on trouve les **propriétés CSS** choisies comme la couleur de fond, la couleur du texte, la taille de police, des bords, etc… → il existe des centaines d'autres propriétés CSS ! (à vous de voir 😉…)

Pour utiliser ce fichier de style CSS dans notre fichier HTML (**index.html**), nous devons simplement modifier la balise **<head>** de notre page HTML :

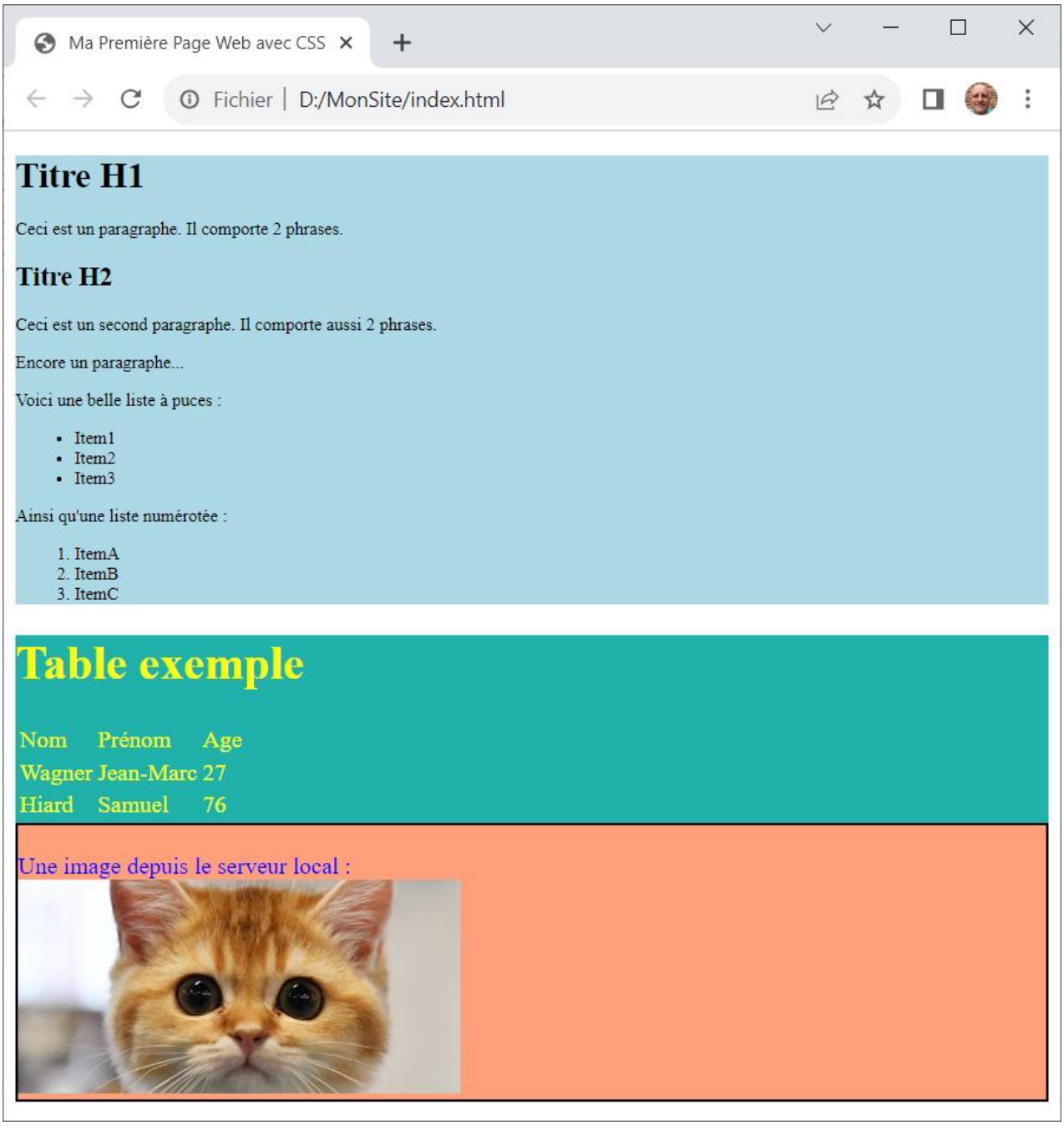
```
…
      <head>
            <title>Ma Première Page Web avec CSS</title>
            <link href="css/style.css" type="text/css" rel="stylesheet">
      </head>
…
```

Dans la balise **<link>** ainsi introduite :

- L'attribut **href** spécifie l'URL ou le chemin du fichier CSS (ici on travaille en local, le fichier **style.css** se trouve dans le répertoire css situé à partir de l'endroit où se situe le fichier **index.html**

- L'attribut **type** précise le type de média associé au fichier lié → ici il s'agit d'un fichier de style CSS → « **text/css** » (cela servira lorsqu'on enverra réellement une requête HTTP pour obtenir le fichier CSS)
- L'attribut **rel** spécifie la relation entre le document HTML actuel et le fichier lié → ici « **rel=stylesheet** » indique que le fichier est une feuille de style

Une fois cette simple ligne ajoutée, le rendu de cette page HTML dans Google Chrome est :



On voit clairement apparaître les 3 « blocs » qui ont des propriétés CSS différentes et ceux-ci sont disposés verticalement.

La mise en page avec Flexbox

Le principe de **Flexbox** est de définir un « **conteneur** » (une balise ** **) qui va contenir des « blocs » c'est-à-dire des éléments HTML ou encore d'autres balises ** **. Ces « blocs » vont pouvoir être positionnés en ligne ou en colonne, et cela avec des marges ou non.

Pour cela, on peut définir un **sélecteur d'identité** pour le conteneur principal de notre page HTML (balise ** ** dont l'id est « **conteneur** »). Pour cela, on peut ajouter ces lignes à notre fichier CSS :

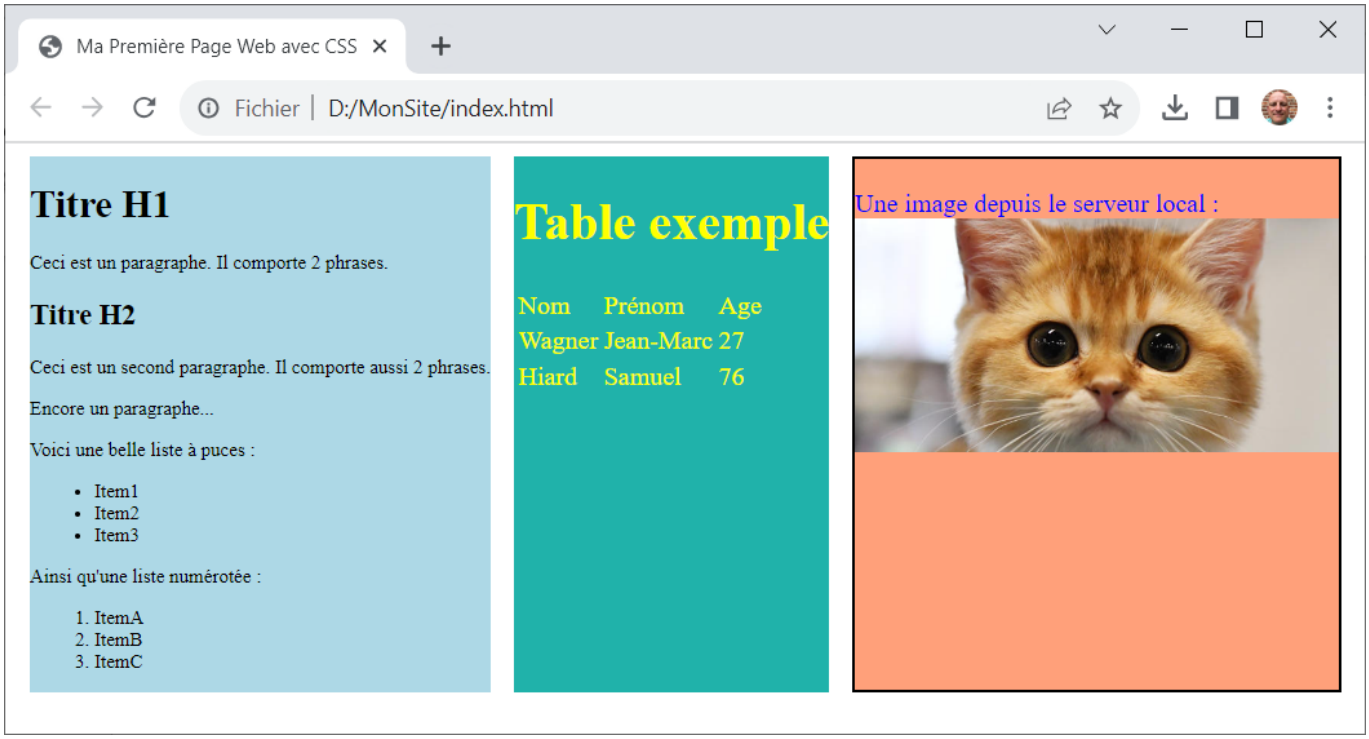
```
…
#conteneur {
      display: flex;
      flex-direction: row;
      justify-content: space-around;
}
```

où

- La propriété **display** met en place **Flexbox** → par défaut les blocs situés dans le « conteneur » se disposent en colonnes
- La propriété **flex-direction** permet de préciser comment les blocs vont se disposer dans le conteneur :
  - **row** : organisés sur une ligne (par défaut)
  - **column** : organisés sur une colonne
  - **row-reverse** : organisés sur une ligne mais dans l'ordre inverse
  - **column-reverse** : organisés sur une colonne mais dans l'ordre inverse
- La propriété **justify-content** permet de spécifier l'alignement des blocs :
  - **flex-start** : alignés au début (par défaut)
  - **flex-end** : alignés à la fin
  - **center** : alignés au centre
  - **space-between** : étirés sur tout l'axe (il y a de l'espace entre eux)
  - **space-around** : étirés sur tout l'axe (il y a de l'espace entre eux mais également sur les extrémités)

Le plus simple est de les tester. Il existe encore d'autres propriétés à **Flexbox** mais elles sont laissées à la curiosité du lecteur.

Une fois que l'on ajouté les quelques lignes ci-dessus à notre fichier **style.css**, le rendu de notre page web dans Google Chrome est



On remarque que les 3 blocs sont alignés horizontalement, étirés sur tout l'axe horizontal mais ont des espaces entre eux, et à gauche et à droite.

Les sélecteurs d'éléments

Il est possible d'attribuer des propriétés CSS à des éléments HTML, comme les paragraphes (**p**), les titres (**h1**), le corps (**body**), la page html elle-même (**html**), …

Par exemple, nous allons donner du style à la page html et aux titres. Pour cela, nous ajoutons les lignes suivantes dans le fichier **style.css** :

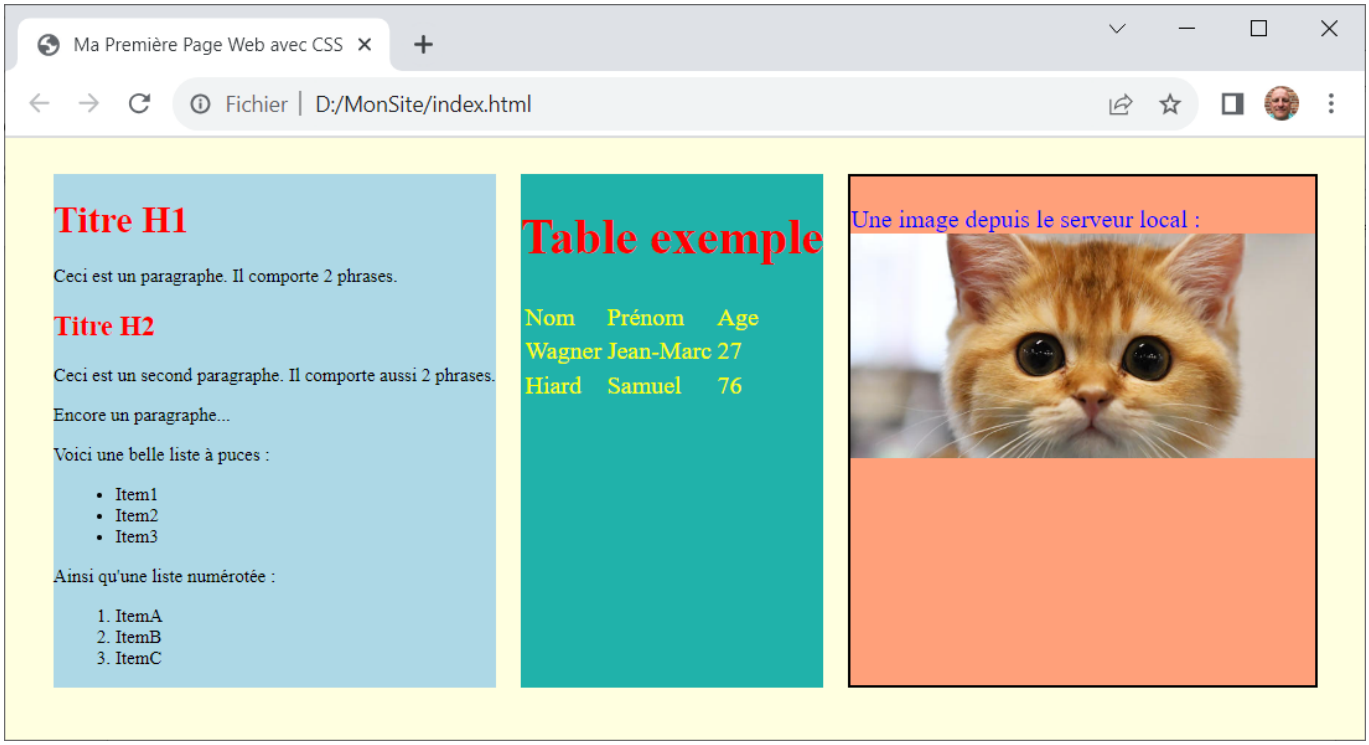
```
…
html {
  background: lightyellow;
  padding: 15px;
}

h1, h2 {
      color: red;
}
```

où

- la page HTML elle-même aura un fond jaune clair et un « padding » de 15 pixels tout autour

- les titres h1 et h2 seront écrits en rouge

Le rendu de notre page web dans Google Chrome est alors



Les sélecteurs de classe

On remarque que dans le dernier exemple, tous les titres **h1** ont été traités de la même manière. Comment puis-je alors faire pour donner du style à un titre bien précis mais pas aux autres ? En définissant une **classe CSS**

Dans l'exemple qui suit, nous allons simplement donner du style au titre « Table Exemple ». Pour cela, on peut définir une classe CSS en ajoutant ces quelques lignes dans le fichier style.css :

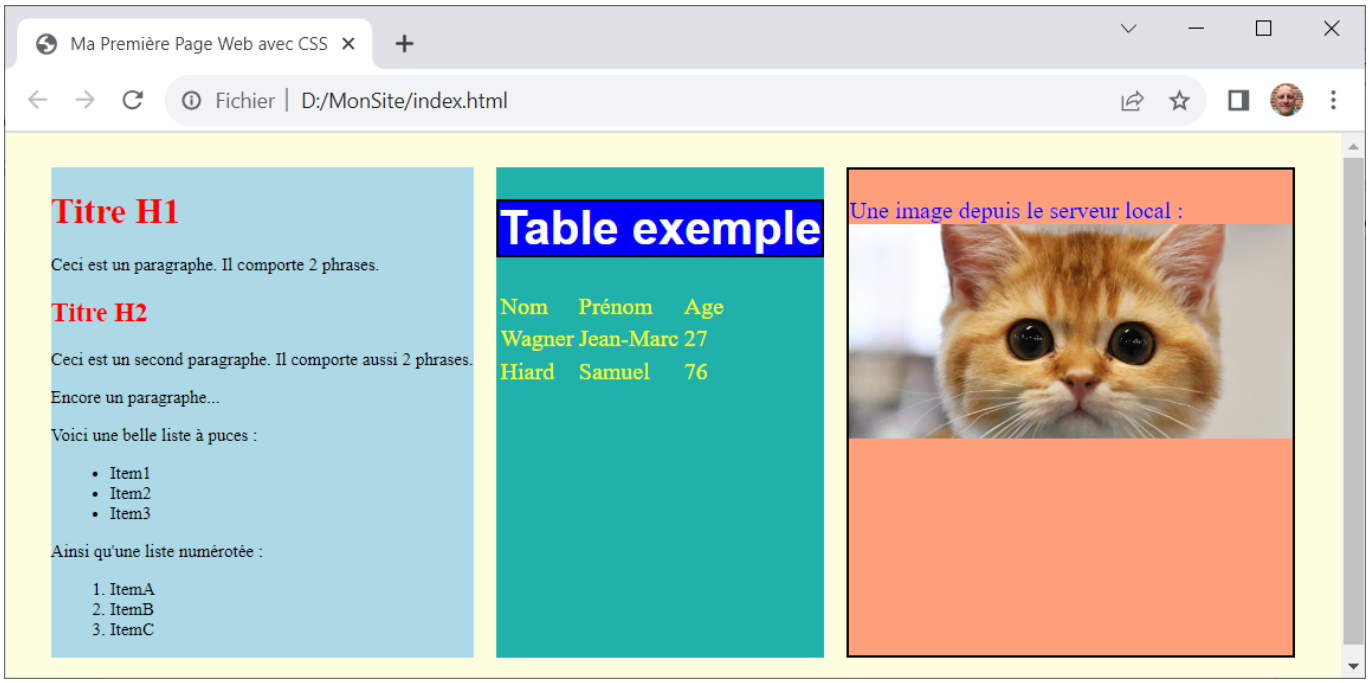
```
.monTitre {
  border: 2px solid black;
  background-color: blue;
  text-align: center;
  font-family: Arial, sans-serif;
  color: white;
}
```

où le nom de la classe CSS est « **monTitre** » et où les propriétés CSS ne demandent pas vraiment d'explication.

Pour pouvoir utiliser cette **classe CSS**, il faut <u>préciser la balise où on veut l'appliquer</u>. Dans notre cas, il s'agit de la balise **<h1>** (située dans le bloc2) que l'on remodifie ainsi dans le fichier HTML :

```
…
        <div id="bloc2">
                    <h1 class="monTitre">Table exemple</h1>
…
```

Le rendu de notre page HTML dans Google Chrome devient alors



<u>En résumé</u>, pour modifier le rendu des éléments HTML, nous disposons de

- **<u>sélecteur d'élément</u>** → il cible tous les éléments d'un type particulier comme p, h1, html, …
- **<u>sélecteur de classe</u>** → il cible tous les éléments qui ont une classe spécifique
- **<u>sélecteur d'identifiant</u>** → il cible un élément spécifique avec un attribut « id » → l'identifiant est unique dans une page HTML

Donner du style à une table HTML

Il est possible de combiner des sélecteurs différents. En effet, on pourrait par exemple vouloir donner des styles différents pour des tables différentes.

Dans notre exemple, nous allons donner un identifiant (« **maTable** ») à la table et donner du style aux éléments **<td>** et **<tr>** de cette table bien précise. Pour cela, nous pouvons ajouter les lignes suivantes au fichier **style.css** :

```
#maTable {
  border-collapse: collapse;    /* Fusionner les bordures adjacentes */
  border: 3px solid black;      /* Bordure du tableau */
}

#maTable td {
  border: 1px solid black;      /* Bordure des cellules */
  padding: 8px;      /* Espacement interne autour du contenu des cellules */
}

/* Appliquer un style différent à la première ligne */
#maTable tr:first-child {
  background-color: #f2f2f2;    /* Couleur de fond de la première ligne */
  font-weight: bold;            /* Texte en gras dans la première ligne */
  color: black;                 /* Couleur du texte */
  font-family: Arial;           /* Police du texte */
}
```

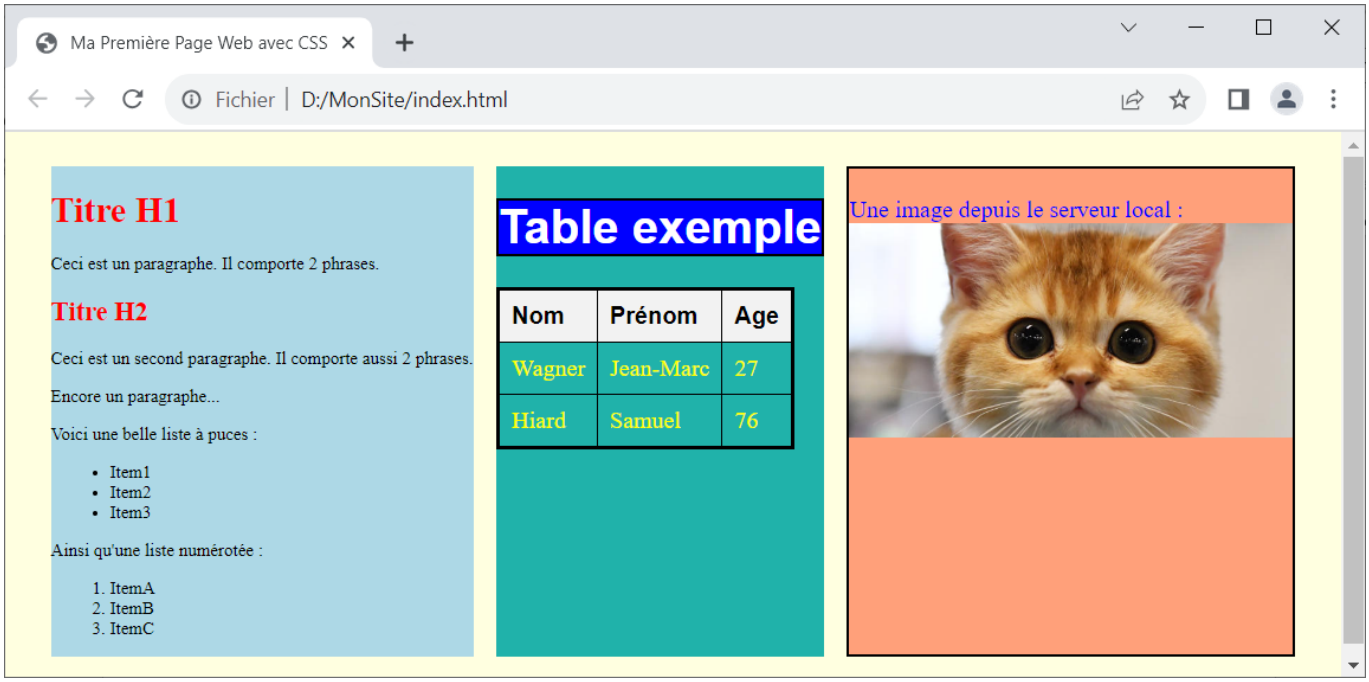
On remarque que

- avec « **#maTable td** », on a combiné un sélecteur d'identifiant (**#maTable**) avec un sélecteur d'élément (**td**) → cela signifie que cette règle CSS ne concerne que les éléments **td** de la table d'identifiant **maTable**
- avec « **#maTable tr:first-child** » c'est la même chose pour les éléments **tr** de la table d'identifiant **maTable** mais uniquement pour le 1er « enfant » de l'arborescence, c'est-à-dire la première ligne de la table ou encore les titres des colonnes

Pour appliquer ces styles CSS, il faut encore donner un identifiant à notre table en modifiant le fichier HTML :

```
…
      <table id="maTable">
…
```

Ainsi, le rendu de notre page HTML dans Google Chrome devient



Un backend élémentaire en Java

Le but ici est donc de créer un **serveur web** qui sera capable de fournir le contenu d'un site à un client HTTP, comme un browser par exemple.

Ce qui suit n'est pas ce que l'on pourrait qualifier de « professionnel ». En réalité, on utiliserait plutôt des serveurs comme **Apache Tomcat**, **Glassfish** ou même des framework comme **Spring Boot**, ou encore **Node.js**

Le but ici est de comprendre ce qu'il se passe derrière ces « machines à gaz »

Un backend simple, nous dirons ici un « **serveur web** », a pour rôle de

- créer une socket d'écoute sur un port particulier (par exemple 80)
- attendre une connexion provenant d'un client HTTP
- lire et « parser » la requête HTTP du client afin de répondre sa demande
- créer une réponse HTTP qui sera envoyée au client
- fermer la connexion et se remettre en attente d'une nouvelle connexion

Remarques :

- Si on désire que notre serveur web puissent accepter plusieurs clients simultanément, il va être nécessaire de le « **threader** » → pool de threads ou autre
- Le **parsing** et la création de messages utilisant le protocole HTTP n'est pas très compliqué (quoi que…) mais peut prendre un temps certain et la plupart des langages de programmation proposent des outils de parsing de ces messages.

Les classes **Java** étudiées ci-dessous permettent de faire cela de manière assez simple

La classe HttpServer

La classe abstraite **HttpServer** du package **com.sun.net.httpserver** permet de créer un serveur HTTP simple en Java → elle va permettre au processus qui l'utilise de

- Écouter sur un port particulier et gérer les requêtes HTTP
- Lorsqu'un requête HTTP est reçue, elle est redirigée vers un « **gestionnaire** » (**handler**) qui traitera cette demande et enverra une réponse HTTP appropriée

- Mettre en place le pool de threads permettant de gérer plusieurs connexions simultanément

Cette classe présente les méthodes principales suivantes :

- **public static HttpServer create(InetSocketAddress addr, int backlog)** → crée une instance de HttpServer qui sera "bindé" à l'adresse addr (adresse IP + port). Le paramètre backlog représente le nombre maximum de connexions TCP entrantes que le système placera en file d'attente → une valeur **0** signifie qu'il n'y a aucune limite à cette file d'attente
- **public abstract HttpContext createContext(String path, HttpHandler handler)** → permet d'associer un "gestionnaire" à un chemin d'accès → elle fait le mapping entre un chemin d'URI (le « **Context path** ») et un **handler** (voir ci-dessous) → le premier caractère de **path** doit être **« / »**
- **public abstract void setExecutor(Executor executor)** → permet de spécifier le pool de threads du serveur représenté par l'objet **executor** → si cette méthode n'est pas appelée ou avec le paramètre null, une implémentation par défaut est utilisée
- **public abstract void start()** → permet de démarrer le serveur, c'est-à-dire créer le pool de threads, et de mettre le serveur en écoute
- **public abstract void stop(int delay)** → permet de stopper le serveur et fermer les sockets. La méthode est bloquante tant que les échanges en cours ne sont pas terminés ou que le « delay » (en secondes) n'est pas écoulé.

Dans un premier temps, il est nécessaire de créer une instance de la classe **HttpServer**. Pour cela, on commence par créer une socket d'écoute en utilisant la classe **InetSocketAddress** qui possède les constructeurs :

- **public InetSocketAddress(InetAddress addr, int port)** → crée une socket à partir d'une adresse IP et d'un port
- **public InetSocketAddress(String hostname, int port)** → crée une socket à partir du hostname de la machine serveur et d'un port
- **public InetSocketAddress(int port)** → crée une socket à partir d'un port → l'adresse IP est alors une adresse « wildcard » (adresse de diffusion ou générique → pour pouvoir écouter sur toutes les interfaces réseau → en IPv4, l'adresse wildcard est généralement représentée par « **0.0.0.0** »)

Nous pouvons à présent créer une instance de la classe **HttpServer** :

```
HttpServer server = HttpServer.create(new InetSocketAddress(8080), 0);
```

où on a choisi le port 8080, une adresse IP wildcard et précisé qu'il n'y a pas de limite à la file d'attente des connexions entrantes.

Même si cela n'est pas nécessaire on peut fixer la taille du pool de threads (par exemple ici 5) :

```
Executor executor = Executors.newFixedThreadPool(5);
server.setExecutor(executor);
```

La classe HttpContext et l'interface HttpHandler

La classe abstraite **HttpContext** modélise le mapping entre le chemin d'une URI (« **Context path** ») et la classe instanciant le gestionnaire associé (**HttpHandler**).

Prenons l'exemple suivant où 3 objets **HttpContext** ont été créés et ajoutés au serveur HTTP par la méthode **createContext** :

HttpContext	Context path
ctx1	"/"
ctx2	"/apps"
ctx3	"/apps/foo"

Les chemins d'URI sont mappés littéralement (de manière « case sensitive ») au « **Context path** » définis dans le serveur. La table suivante présente des URI dont 3 sont matchées avec un des 3 contextes définis ci-dessus et 1 qui provoquera l'envoi d'une réponse HTTP avec le code 404 (« Not found ») :

URI de la requête	HttpContext qui « match »
"http://foo.com/apps/foo/bar"	ctx3
"http://foo.com/apps/Foo/bar"	Pas de correspondance → code 404
"http://foo.com/apps/app1"	ctx2
"http://foo.com/foo"	ctx1

Un « gestionnaire » d'URI doit implémenter l'interface **HttpHandler** qui ne possède qu'une seule méthode

- **void handle(HttpExchange exchange)**

Cette méthode reçoit en paramètre un objet **HttpExchange** qui modélise la connexion établie entre le client et le serveur HTTP. C'est dans la méthode **handle** que

- on récupèrera les paramètres de la requête HTTP
- on réalisera l'action souhaitée par le client
- on enverra la réponse HTTP au client

Dans l'exemple ci-dessus, pour mettre en place les 3 contextes cités, cela donnera au niveau code

```
server.createContext("/", new HandlerCtx1());
server.createContext("/apps", new HandlerCtx2());
server.createContext("/apps/foo", new HandlerCtx3());
```

où les classes **HandlerCtx1**, **HandlerCtx2**, **HandlerCtx3** (à créer par le programmeur applicatif) implémentent l'interface **HttpHandler**

La classe HttpExchange

La classe **HttpExchange** modélise la connexion établie entre le client et le serveur HTTP → elle contient le requête http reçue mais également tout ce qui est nécessaire pour répondre au client. Elle dispose de méthodes pour examiner le contenu de la requête mais également pour construire la réponse à envoyer au client.

Le cycle de vie typique d'un objet **HttpEchange** est le suivant :

1. Appel à la méthode **String getRequestMethod()** afin de déterminer la commande (GET, POST, …)
2. Appel à la méthode **Headers getRequestHeaders()** afin d'examiner le header de la requête (si nécessaire)
3. Appel à la méthode **InputStream getRequestBody()** → retourne une instance de la classe **InputStream** → ce flux va permettre de lire le corps de la requête → après lecture, le flux est fermé
4. Appel à la méthode **Headers getResponseHeaders()** afin de préciser le header de la réponse, à l'exception du **Content-Length** qui se fait avec la méthode suivante
5. Appel à la méthode **void sendResponseHeaders(int code, long responseLength)** afin d'envoyer le **code** de retour de la réponse (200, …) ainsi que le **Content-Length** de la réponse → cette méthode doit être appelée obligatoirement avant la suivante

6. Appel à la méthode **OutputStream getResponseBody()** → retourne une instance de la classe **OutputStream** → ce flux va permettre d'écrire/envoyer le corps de la réponse → après écriture, le flux doit être fermé (appel manuel de **close()**) pour terminer l'échange

Remarques :

- Les échanges entre le client et le serveur sont terminés lorsque les **InputStream** et **OutputStream** sont tous deux fermés
- Fermer le flux de sortie **OutputStream** ferme automatiquement le flux d'entrée **InputStream**
- Terminer l'échange avant d'avoir lu complètement la requête peut placer la connexion TCP dans un état incertain pour les échanges suivants → à éviter
- La classe **HttpExchange** dispose également de la méthode **URI getRequestURI()** permettant de récupérer l'URI de la requête

La classe Headers

La classe **Headers** représente le header d'une requête ou d'une réponse HTTP. Cette classe implémente l'interface **Map<String,List<String>>** et contient donc l'ensemble des couples clés/valeurs des headers d'une message HTTP. Elle dispose notamment des méthodes

- **String getFirst(String key)** → retourne la première valeur associée à la clé
- **void add(String key, String value)** → ajoute la value à la key correspondante → une même clé peut avoir plusieurs valeurs
- **void set(String key, String value)** → associe value à key en supprimant toutes les valeurs précédentes associées à key

Premier exemple de serveur avec un seul HttpContext

Dans l'exemple qui suit, nous mettons en place un serveur HTTP ayant un seul **HttpContext** liée au chemin d'URI **« / »**.

Voici le code du **serveur** proprement dit (fichier **Main.java**) :

```
package ServeurWeb;

import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
```

```
import java.net.InetSocketAddress;

public class Main
{
    public static void main(String[] args)
    {
        HttpServer serveur = null;
        try
        {
            serveur = HttpServer.create(new InetSocketAddress(8080),0);
            serveur.createContext("/",new HandlerRoot());
            System.out.println("Demarrage du serveur HTTP...");
            serveur.start();
        }
        catch (IOException e)
        {
            System.out.println("Erreur: " + e.getMessage());
        }
    }
}
```

On remarque que :

- le serveur HTTP va attendre sur le port **8080** et il n'y a pas de limite à la file d'attente des connexions
- Le seul **HttpContext** défini associe le chemin d'URI **« / »** au handler **HandlerRoot()**

Le code de la classe **HandlerRoot** (fichier **HandlerRoot.java**) est donné par

```
package ServeurWeb;

import com.sun.net.httpserver.*;
import java.io.*;
import java.net.InetSocketAddress;

public class HandlerRoot implements HttpHandler
{
    @Override
    public void handle(HttpExchange exchange) throws IOException
    {
        System.out.println("--- Requete recue ---");

        // Lecture de la requete
        String requestPath = exchange.getRequestURI().getPath();
        System.out.println("requestPath = " + requestPath);

        String requestMethod = exchange.getRequestMethod();
        System.out.println("requestMethod = " + requestMethod);

        Headers requestHeaders = exchange.getRequestHeaders();
```

```
        System.out.println("Header : " );
        for (String key : requestHeaders.keySet())
            System.out.println(key + ": " + requestHeaders.get(key));

        InputStream is = exchange.getRequestBody();
        BufferedReader br = new BufferedReader(new InputStreamReader(is));
        String ligne;
        System.out.println("Request Body :");
        while ((ligne = br.readLine()) != null)
            System.out.println(ligne);

        // Ecriture de la reponse
        String reponse = "ceci est le corps de la reponse";
        exchange.getResponseHeaders().set("Content-Type","text/plain");
        exchange.sendResponseHeaders(200,reponse.length());
        OutputStream os = exchange.getResponseBody();
        os.write(reponse.getBytes());
        os.close();

        System.out.println("--- Reponse envoyee ---\n");
    }
}
```

Une fois lancé sous linux, on peut tester ce petit serveur HTTP avec la commande **http** :

```
$ http http://localhost:8080/nom=charlet --verbose
GET /nom=charlet HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8080
User-Agent: HTTPie/2.6.0



HTTP/1.1 200 OK
Content-length: 31
Content-type: text/plain
Date: Mon, 04 Sep 2023 12:33:23 GMT

ceci est le corps de la reponse


$ http --form POST http://localhost:8080 'nom'='wagner' 'prenom'='jean-marc'
Content-Type:'application/x-www-from-urlencoded' --verbose
POST / HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

```
Content-Length: 27
Content-Type: application/x-www-from-urlencoded
Host: localhost:8080
User-Agent: HTTPie/2.6.0

nom=wagner&prenom=jean-marc

HTTP/1.1 200 OK
Content-length: 31
Content-type: text/plain
Date: Mon, 04 Sep 2023 12:33:28 GMT

ceci est le corps de la reponse


$
```

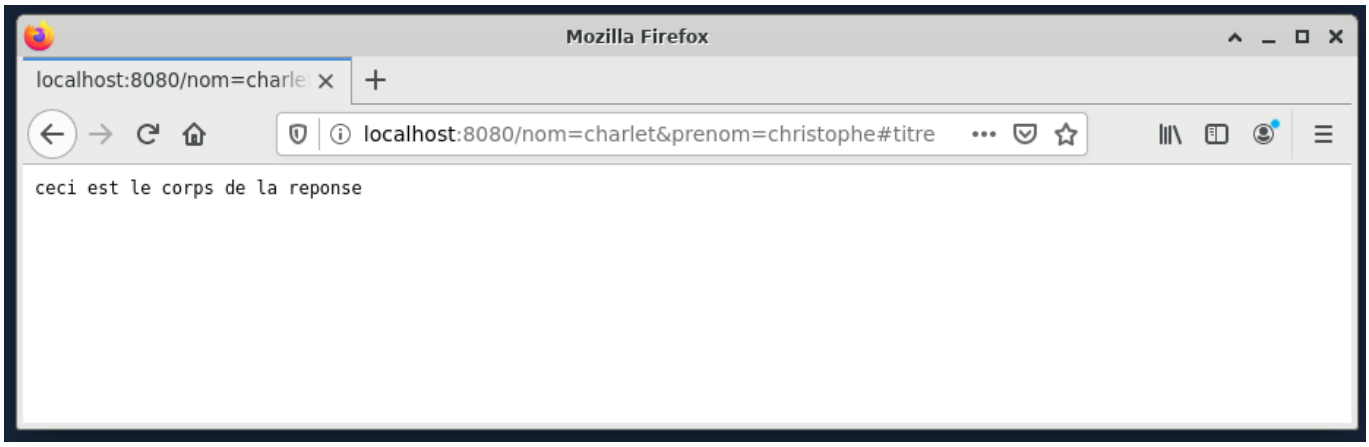
Tandis que sur la console du **serveur**, on observe :

```
$ java ServeurWeb.Main
Demarrage du serveur HTTP...
--- Requete recue ---
requestPath = /nom=charlet
requestMethod = GET
Header :
Accept-encoding: [gzip, deflate]
Accept: [*/*]
Connection: [keep-alive]
Host: [localhost:8080]
User-agent: [HTTPie/2.6.0]
Request Body :
--- Reponse envoyee ---

--- Requete recue ---
requestPath = /
requestMethod = POST
Header :
Accept-encoding: [gzip, deflate]
Accept: [*/*]
Connection: [keep-alive]
Host: [localhost:8080]
User-agent: [HTTPie/2.6.0]
Content-type: [application/x-www-from-urlencoded]
Content-length: [27]
Request Body :
nom=wagner&prenom=jean-marc
--- Reponse envoyee ---
```

On remarque que :

- La 1ère requête est de type **GET** → les paramètres (**« nom=charlet »**) passent directement dans l'URL et non dans le corps de la requête (resté vide ici)
- La 2ème requête est de type **POST** → les paramètres (**« nom=wagner&prenom=jean-marc »**) passent dans le corps de la requête → il s'agit typiquement d'une requête envoyée par un **formulaire HTML** (voir plus loin)

Il est également possible de tester notre serveur web à partir d'un **browser**, ici Firefox :



On remarque que le corps de la réponse (dont le type est **text/plain**) est interprété par le navigateur comme un simple texte sans mise en forme.

Au niveau de la console du **serveur**, nous observons

```
...
--- Requete recue ---
requestPath = /nom=charlet&prenom=christophe
requestMethod = GET
Header :
Accept-encoding: [gzip, deflate]
Accept:
[text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8]
Connection: [keep-alive]
Host: [localhost:8080]
User-agent: [Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101
Firefox/78.0]
Accept-language: [fr,fr-FR;q=0.8,en-US;q=0.5,en;q=0.3]
Upgrade-insecure-requests: [1]
Request Body :
--- Reponse envoyee ---
```

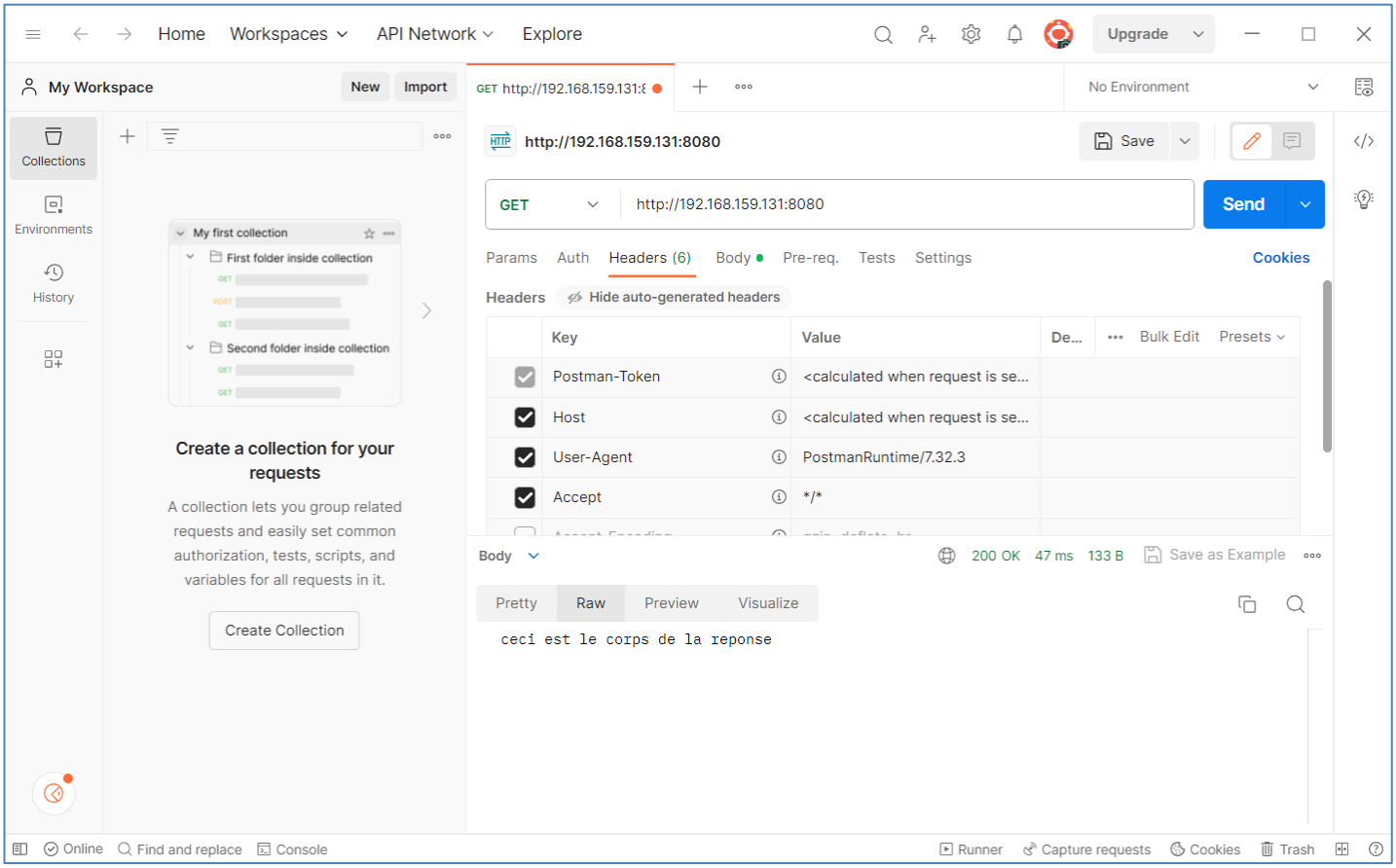
Il s'agit d'une simple requête **GET** dont le corps est vide.

Une manière simple et efficace de tester un serveur web (ou même une API web que l'on développe → voir plus loin) est d'utiliser le logiciel **Postman** (https://www.postman.com/) :

Ce soft permet de

- Créer simplement et « graphiquement » des requêtes HTTP et de les envoyer à un serveur web
- Récupérer et analyser la réponse fournie par le serveur

Voici un premier exemple d'utilisation de logiciel avec notre petit serveur web :



On voit qu'il est possible

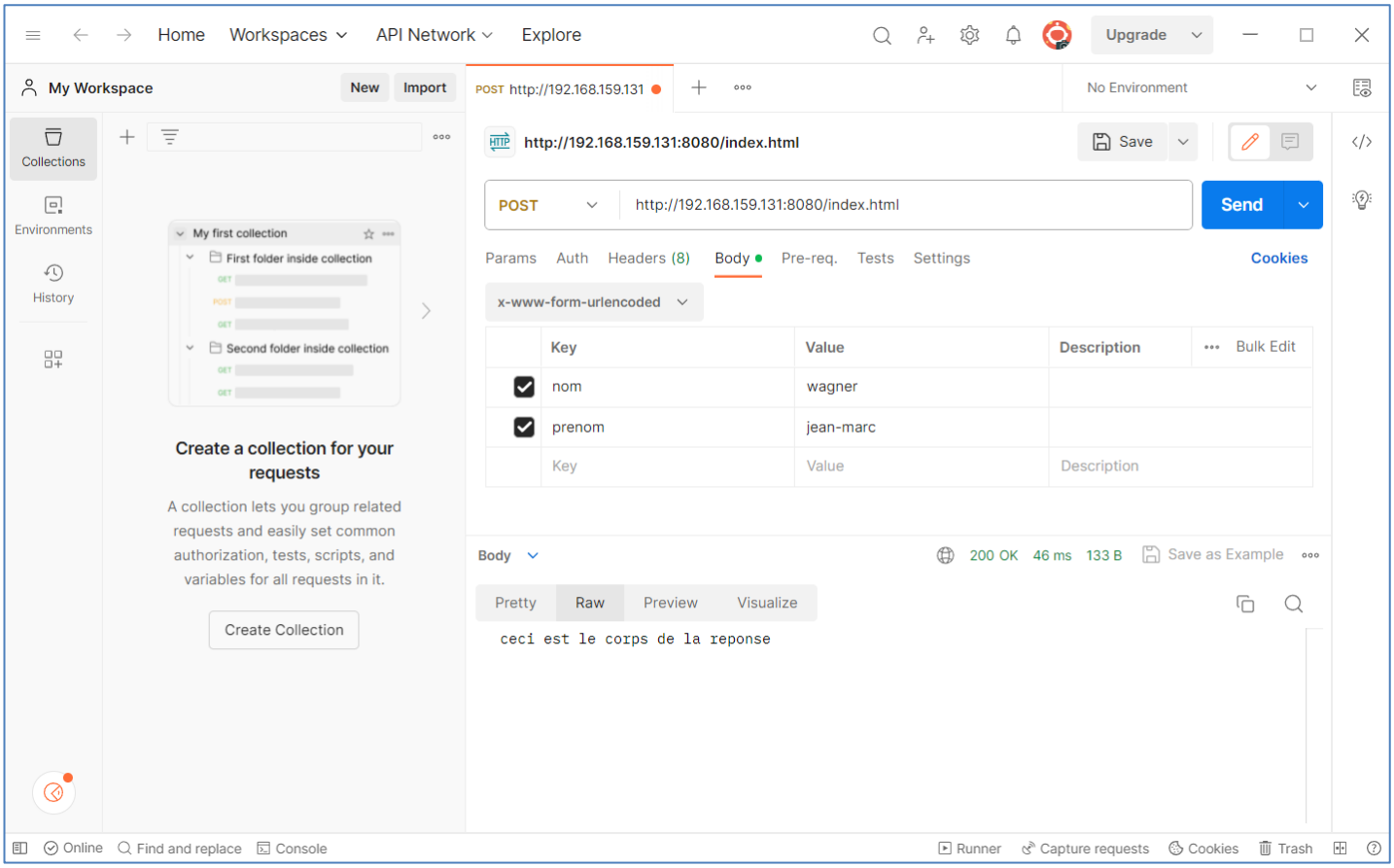
- De préciser le nom de la commande (POST, GET, …) ainsi que l'URL → ici **GET**
- Préciser les entêtes de la requête ainsi que son corps

- Et bien d'autre choses encore…
- La lecture de la réponse est affichée dans le bas et peut-être interprétée de différentes manières selon la nature de la réponse

Pour la requête générée ci-dessus dans **Postman**, on observe ceci dans la console du **serveur** (tournant sur une machine d'adresse IP **192.168.159.131**) :

```
--- Requete recue ---
requestPath = /
requestMethod = GET
Header :
Accept: [*/*]
Host: [192.168.159.131:8080]
User-agent: [PostmanRuntime/7.32.3]
Postman-token: [a269aab2-bfec-4662-876d-448fc783da81]
Request Body :
--- Reponse envoyee ---
```

Nous pouvons également faire le test pour une requête **POST** ayant un corps :



Tandis que nous observons ceci dans la console du **serveur** :

```
--- Requete recue ---
requestPath = /index.html
requestMethod = POST
Header :
Accept: [*/*]
Host: [192.168.159.131:8080]
User-agent: [PostmanRuntime/7.32.3]
Content-type: [application/x-www-form-urlencoded]
Postman-token: [f09249ec-8d69-4a6a-b50b-5c721559438d]
Content-length: [27]
Request Body :
nom=wagner&prenom=jean-marc
--- Reponse envoyee ---
```

Exemple de serveur web avec plusieurs HttpContext

Dans l'exemple qui suit, nous mettons en place un **serveur web** permettant de fournir du contenu au format **HTML**, **CSS** mais également des **images** et des **fichiers à télécharger**.

Dans cet exemple, le serveur va tourner sur une machine linux dont l'adresse IP est **192.168.159.131** et sera en attente sur le port **8080**. Le contenu du site web se trouve dans le répertoire **/home/student/MonSite**. Ce répertoire contient les sous-répertoires

- **html** : contenant les fichiers index.html (point d'entrée du site) et download.html proposant un lien de téléchargement d'un fichier PDF
- **css** : contenant le fichier style.css utilisé par les deux fichiers HTML
- **images** : contenant le fichier petitchat.jpg affichée par la page index.html
- **pdfs** : contenant le fichier PDF chat.pdf qui pourra être téléchargé sur la page download.html

Voici le contenu du fichier **index.html** :

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="UTF-8">
		<title>Ma Première Page Web avec CSS</title>
		<link href="css/style.css" type="text/css" rel="stylesheet">
	</head>

	<body>
```

```
        <div id="conteneur">

            <div id="bloc1">
            <h1>Titre H1</h1>
            <p>Ceci est un paragraphe. Il comporte 2 phrases.</p>

            <h2>Titre H2</h2>
            <p>Ceci est un second paragraphe. Il comporte aussi 2
phrases.</p>
            <p>Encore un paragraphe...</p>

            <p>
                Voici une belle liste à puces :
                <ul>
                    <li>Item1</li>
                    <li>Item2</li>
                    <li>Item3</li>
                </ul>

                Ainsi qu'une liste numérotée :
                <ol>
                    <li>ItemA</li>
                    <li>ItemB</li>
                    <li>ItemC</li>
                </ol>
            </p>
            </div>

            <div id="bloc2">
            <h1 class="monTitre">Table exemple</h1>

            <table id="maTable">
                <tr>
                        <td>Nom</td>
                        <td>Prénom</td>
                        <td>Age</td>
                </tr>
                <tr>
                        <td>Wagner</td>
                        <td>Jean-Marc</td>
                        <td>27</td>
                </tr>
                <tr>
                        <td>Hiard</td>
                        <td>Samuel</td>
                        <td>76</td>
                </tr>
            </table>
            </div>
```

```
                <div id="bloc3">
                <br/>Une image depuis le serveur local : <br/>
                <img src="images/petitchat.jpg" width=300 alt="un petit chat
mignon">
                <p><a href="download.html">Cliquez ici pour accéder à la page
de téléchargement.</a></p>
                </div>

            </div>
      </body>
</html>
```

Ce fichier correspond au fichier HTML utilisé plus haut comme exemple, à la différence qu'un lien vers la page **download.html** a été ajouté dans le bloc3.

Voici le contenu du fichier **download.html** :

```
<!DOCTYPE html>
<html>
      <head>
            <meta charset="UTF-8">
            <title>Page de téléchargement</title>
            <link href="css/style.css" type="text/css" rel="stylesheet">
      </head>

      <body>
            <div id="bloc1">
            <h1 class="monTitre">Page de téléchargement</h1>
            <p><a href="pdfs/chat.pdf">Cliquez ici pour télécharger le fichier
chat.pdf</a></p>
            </div>
      </body>
</html>
```

Cette page utilise la même feuille de style que index.html et on y voit apparaître un lien de téléchargement du fichier PDF **chat.pdf**

Le fichier **style.css** complet est donné par :

```
#bloc1 {
      background-color: lightblue;
      color: black;
      font-size: 12px;
}

#bloc2 {
      background-color: lightseagreen;
      color: yellow;
}

#bloc3 {
```

```
      background-color: lightsalmon;
      color: blue;
      border: 2px solid black;
}

#conteneur {
      display: flex;
      flex-direction: row;
      justify-content: space-around;
}

html {
  background: lightyellow;
  padding: 15px;
}

h1, h2 {
      color: red;
}

.monTitre {
  border: 2px solid black;
  background-color: blue;
  text-align: center;
  font-family: Arial, sans-serif;
  color: white;
}

#maTable {
  border-collapse: collapse;   /* Fusionner les bordures adjacentes */
  border: 3px solid black;     /* Bordure du tableau */
}

#maTable td {
  border: 1px solid black;     /* Bordure des cellules */
  padding: 8px;                  /* Espacement interne autour du contenu des
cellules */
}

/* Appliquer un style différent à la première ligne */
#maTable tr:first-child {
  background-color: #f2f2f2;    /* Couleur de fond de la première ligne */
  font-weight: bold;            /* Texte en gras dans la première ligne */
  color: black;                 /* Couleur du texte */
  font-family: Arial;           /* Police du texte */
}
```

et correspond exactement à celui utilisé plus haut dans l'exemple.

Nous sommes en présence de 4 ressources de types différents et notre **serveur HTTP** va gérer ces ressources à l'aide de **4 HttpHandler** différents.

Le code du serveur (fichier Main.cpp) est

```
package ServeurWebHtmlCss;

import com.sun.net.httpserver.HttpServer;
import java.io.IOException;
import java.net.InetSocketAddress;

public class Main
{
    public static void main(String[] args)
    {
        HttpServer serveur = null;
        try
        {
            serveur = HttpServer.create(new InetSocketAddress(8080),0);
            serveur.createContext("/",new HandlerHtml());
            serveur.createContext("/css",new HandlerCss());
            serveur.createContext("/images",new HandlerImages());
            serveur.createContext("/pdfs",new HandlerPdfs());
            System.out.println("Demarrage du serveur HTTP...");
            serveur.start();
        }
        catch (IOException e)
        {
            System.out.println("Erreur: " + e.getMessage());
        }
    }
}
```

On observe que 4 « **Context Path** » ont été définis et un handler spécifique leur a été assigné.

Le code de la classe **HandlerHtml** (fichier **HandlerHtml.java**) est donné par

```
package ServeurWebHtmlCss;

import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpHandler;

import java.io.*;
import java.nio.file.Files;

public class HandlerHtml implements HttpHandler
{
    @Override
    public void handle(HttpExchange exchange) throws IOException
    {
        // Lecture de la requete
        String requestPath = exchange.getRequestURI().getPath();
        String requestMethod = exchange.getRequestMethod();
```

```
        System.out.print("HandlerHtml (methode " + requestMethod + ")    = " +
requestPath + " --> ");

        // Ecriture de la reponse
        if (requestPath.endsWith(".html"))
        {
            String fichier = "/home/student/MonSite/html/" + requestPath;
            File file = new File(fichier);
            if (file.exists())
            {
                exchange.sendResponseHeaders(200, file.length());
                exchange.getResponseHeaders().set("Content-Type",
"text/html");
                OutputStream os = exchange.getResponseBody();
                Files.copy(file.toPath(), os);
                os.close();
                System.out.println("OK");
            }
            else Erreur404(exchange);
        }
        else Erreur404(exchange);
    }

    private void Erreur404(HttpExchange exchange) throws IOException
    {
        String reponse = "Fichier HTML introuvable !!!";
        exchange.sendResponseHeaders(404, reponse.length());
        exchange.getResponseHeaders().set("Content-Type", "text/plain");
        OutputStream os = exchange.getResponseBody();
        os.write(reponse.getBytes());
        os.close();
        System.out.println("KO");
    }
}
```

On observe que

- le « **request path** » est utilisé pour vérifier que la ressource demandée se termine bien par « **.html** » → dans le cas contraire, une erreur 404 est envoyée au client
- si la ressource est trouvée, le code **200** est utilisé et le « **Content-Type** » prend la valeur « **text/html** » pour préciser que client que la réponse reçue est bien un fichier HTML
- la méthode **copy** de la classe **Files** permet de copier le contenu entier d'un fichier sur un flux de sortie, ce qui facilite les choses

Le code de la classe **HandlerCss** (fichier **HandlerCss.java**) est fourni par

```
package ServeurWebHtmlCss;

import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpHandler;

import java.io.File;
import java.io.IOException;
import java.io.OutputStream;
import java.nio.file.Files;

public class HandlerCss implements HttpHandler
{
    @Override
    public void handle(HttpExchange exchange) throws IOException
    {
        // Lecture de la requete
        String requestPath = exchange.getRequestURI().getPath();
        String requestMethod = exchange.getRequestMethod();
        System.out.print("HandlerCss (methode " + requestMethod + ")     = " +
requestPath + " --> ");

        // Ecriture de la reponse
        if (requestPath.endsWith(".css"))
        {
            String fichier = "/home/student/MonSite/" + requestPath;
            File file = new File(fichier);
            if (file.exists())
            {
                exchange.sendResponseHeaders(200, file.length());
                exchange.getResponseHeaders().set("Content-Type",
"text/css");
                OutputStream os = exchange.getResponseBody();
                Files.copy(file.toPath(), os);
                os.close();
                System.out.println("OK");
            }
            else Erreur404(exchange);
        }
        else Erreur404(exchange);
    }

    private void Erreur404(HttpExchange exchange) throws IOException
    {
        String reponse = "Fichier CSS introuvable !!!";
        exchange.sendResponseHeaders(404, reponse.length());
        exchange.getResponseHeaders().set("Content-Type", "text/plain");
        OutputStream os = exchange.getResponseBody();
        os.write(reponse.getBytes());
```

```
            os.close();
            System.out.println("KO");
        }
}
```

Il est tout à fait similaire au précédent mis à part pour

- le nom du fichier à aller lire sur disque (côté serveur) ne doit pas contenir le répertoire « css » → en effet, celui-ci est compris dans le « **request path** » → ce n'est pas une obligation mais c'est plus simple ainsi
- le « **Content-Type** » de la réponse est à présent « **text/css** »

Le code de la classe **HandlerImages** (fichier **HandlerImages.hava**) est

```
package ServeurWebHtmlCss;

import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpHandler;

import java.io.File;
import java.io.IOException;
import java.io.OutputStream;
import java.nio.file.Files;

public class HandlerImages implements HttpHandler
{
    @Override
    public void handle(HttpExchange exchange) throws IOException
    {
        // Lecture de la requete
        String requestPath = exchange.getRequestURI().getPath();
        String requestMethod = exchange.getRequestMethod();
        System.out.print("HandlerImages (methode " + requestMethod + ") = " +
requestPath + " --> ");

        // Ecriture de la reponse
        if (requestPath.endsWith(".jpg"))
        {
            String fichier = "/home/student/MonSite/" + requestPath;
            File file = new File(fichier);
            if (file.exists())
            {
                exchange.sendResponseHeaders(200, file.length());
                exchange.getResponseHeaders().set("Content-Type",
"image/jpeg");
                OutputStream os = exchange.getResponseBody();
                Files.copy(file.toPath(), os);
                os.close();
                System.out.println("OK");
            }
```

```
            else Erreur404(exchange);
        }
        else Erreur404(exchange);
    }

    private void Erreur404(HttpExchange exchange) throws IOException
    {
        String reponse = "Image introuvable !!!";
        exchange.sendResponseHeaders(404, reponse.length());
        exchange.getResponseHeaders().set("Content-Type", "text/plain");
        OutputStream os = exchange.getResponseBody();
        os.write(reponse.getBytes());
        os.close();
        System.out.println("KO");
    }
}
```

De nouveau très similaire mis à part que le « **Content-Type** » de la réponse est « **image/jpeg** » → remarquez que ce handler ne sera capable que de traiter des images jpeg.

Le code de la classe **HandlerPdfs** (fichier **HandlerPdfs.java**) est

```
package ServeurWebHtmlCss;

import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpHandler;

import java.io.File;
import java.io.IOException;
import java.io.OutputStream;
import java.nio.file.Files;

public class HandlerPdfs implements HttpHandler
{
    @Override
    public void handle(HttpExchange exchange) throws IOException
    {
        // Lecture de la requete
        String requestPath = exchange.getRequestURI().getPath();
        String requestMethod = exchange.getRequestMethod();
        System.out.print("HandlerPdfs (methode " + requestMethod + ")    = " +
requestPath + " --> ");

        // Ecriture de la reponse
        if (requestPath.endsWith(".pdf"))
        {
            String fichier = "/home/student/MonSite/" + requestPath;
            File file = new File(fichier);
            if (file.exists())
```

```
            {
                exchange.sendResponseHeaders(200, file.length());
                exchange.getResponseHeaders().set("Content-Type",
"application/pdf");
                OutputStream os = exchange.getResponseBody();
                Files.copy(file.toPath(), os);
                os.close();
                System.out.println("OK");
            }
            else Erreur404(exchange);
        }
        else Erreur404(exchange);
    }

    private void Erreur404(HttpExchange exchange) throws IOException
    {
        String reponse = "Fichier PDF introuvable !!!";
        exchange.sendResponseHeaders(404, reponse.length());
        exchange.getResponseHeaders().set("Content-Type", "text/plain");
        OutputStream os = exchange.getResponseBody();
        os.write(reponse.getBytes());
        os.close();
        System.out.println("KO");
    }
}
```

De nouveau très similaire sauf pour le « **Content-Type** » de la réponse qui est ici « **application/pdf** » dans le cas d'un fichier PDF contenu dans la réponse HTTP.

Une fois le serveur lancé, nous pouvons tester son API sous linux avec la commande **http** :

```
$ http http://localhost:8080/download.html --verbose
GET /download.html HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8080
User-Agent: HTTPie/2.6.0



HTTP/1.1 200 OK
Content-length: 401
Date: Fri, 15 Sep 2023 05:19:07 GMT

<!DOCTYPE html>
<html>
     <head>
                <meta charset="UTF-8">
```

```
            <title>Page de téléchargement</title>
            <link href="css/style.css" type="text/css" rel="stylesheet">
      </head>

      <body>
                  <div id="bloc1">
                  <h1 class="monTitre">Page de téléchargement</h1>
                  <p><a  href="pdfs/chat.pdf">Cliquez  ici  pour  télécharger  le
fichier chat.pdf</p>
                  </div>
      </body>
</html>


$
```

pour une des 2 pages html. On peut également le faire le fichier CSS :

```
$ http http://localhost:8080/css/style.css --verbose
GET /css/style.css HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8080
User-Agent: HTTPie/2.6.0



HTTP/1.1 200 OK
Content-length: 1233
Date: Fri, 15 Sep 2023 05:24:11 GMT

#bloc1 {
      background-color: lightblue;
      color: black;
      font-size: 12px;
}

#bloc2 {
      background-color: lightseagreen;
      color: yellow;
}

#bloc3 {
      background-color: lightsalmon;
      color: blue;
      border: 2px solid black;
}

#conteneur {
      display: flex;
```

```
      flex-direction: row;
      justify-content: space-around;
}

html {
  background: lightyellow;
  padding: 15px;
}

h1, h2 {
      color: red;
}

.monTitre {
  border: 2px solid black;
  background-color: blue;
  text-align: center;
  font-family: Arial, sans-serif;
  color: white;
}

#maTable {
  border-collapse: collapse;   /* Fusionner les bordures adjacentes */
  border: 3px solid black;     /* Bordure du tableau */
}

#maTable td {
  border: 1px solid black;     /* Bordure des cellules */
  padding: 8px;                  /* Espacement interne autour du contenu des
cellules */
}

/* Appliquer un style différent à la première ligne */
#maTable tr:first-child {
  background-color: #f2f2f2;    /* Couleur de fond de la première ligne */
  font-weight: bold;            /* Texte en gras dans la première ligne */
  color: black;                 /* Couleur du texte */
  font-family: Arial;           /* Police du texte */
}


$
```

Et même le fichier image :

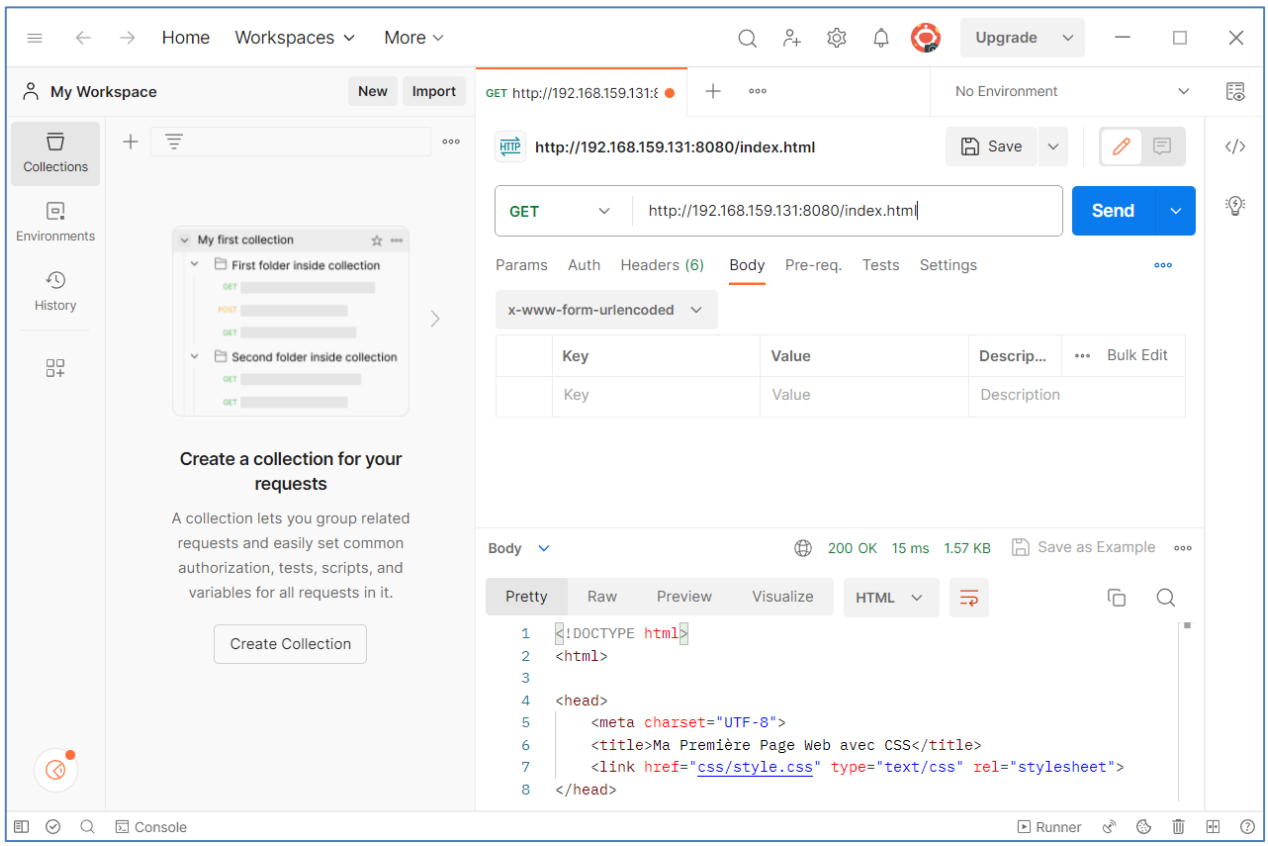
```
$ http http://localhost:8080/images/chat.jpg --verbose
GET /images/chat.jpg HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8080
```

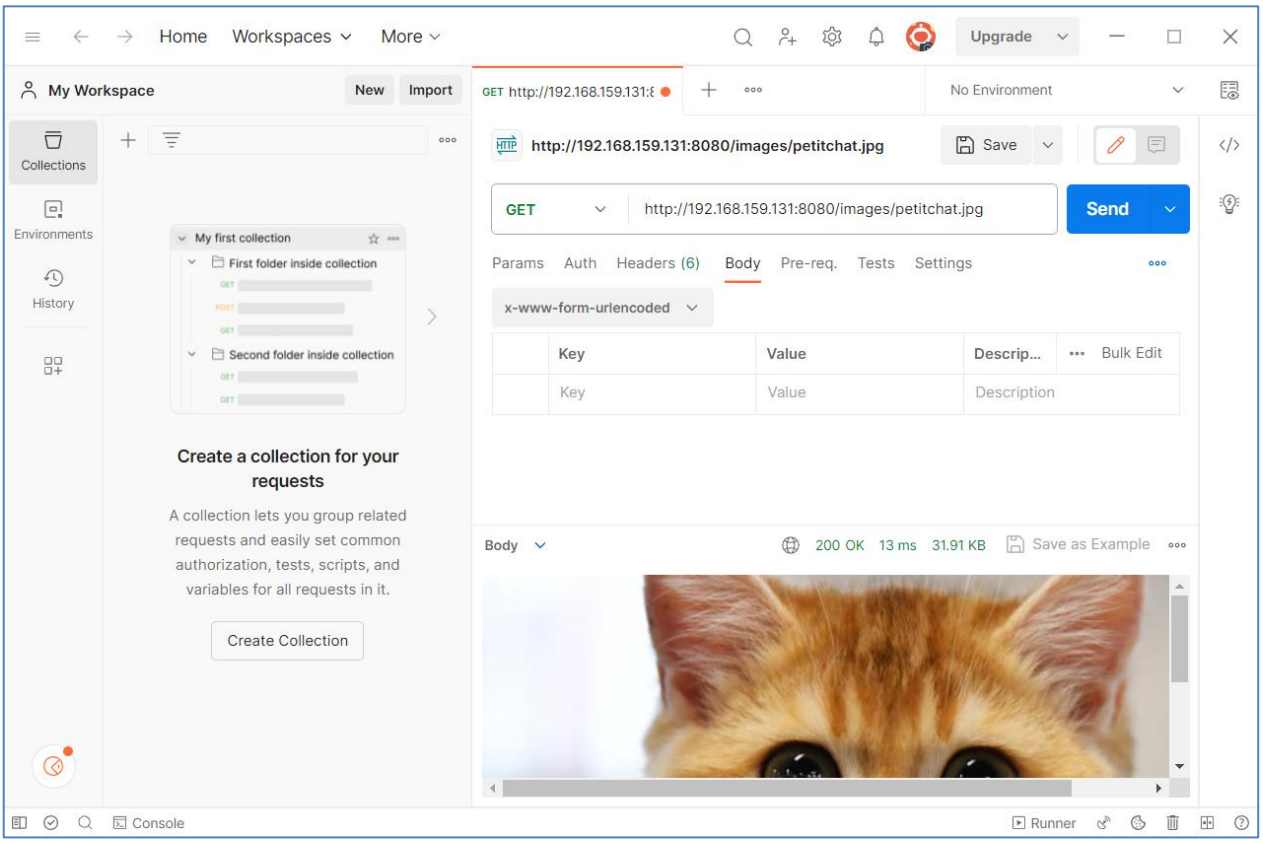
```
User-Agent: HTTPie/2.6.0



HTTP/1.1 404 Not Found
Content-length: 21
Date: Fri, 15 Sep 2023 05:26:21 GMT

Image introuvable !!!

$ http http://localhost:8080/images/petitchat.jpg --verbose
GET /images/petitchat.jpg HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8080
User-Agent: HTTPie/2.6.0



HTTP/1.1 200 OK
Content-length: 32598
Date: Fri, 15 Sep 2023 05:26:26 GMT



+-----------------------------------------+
| NOTE: binary data not shown in terminal |
+-----------------------------------------+

$
```

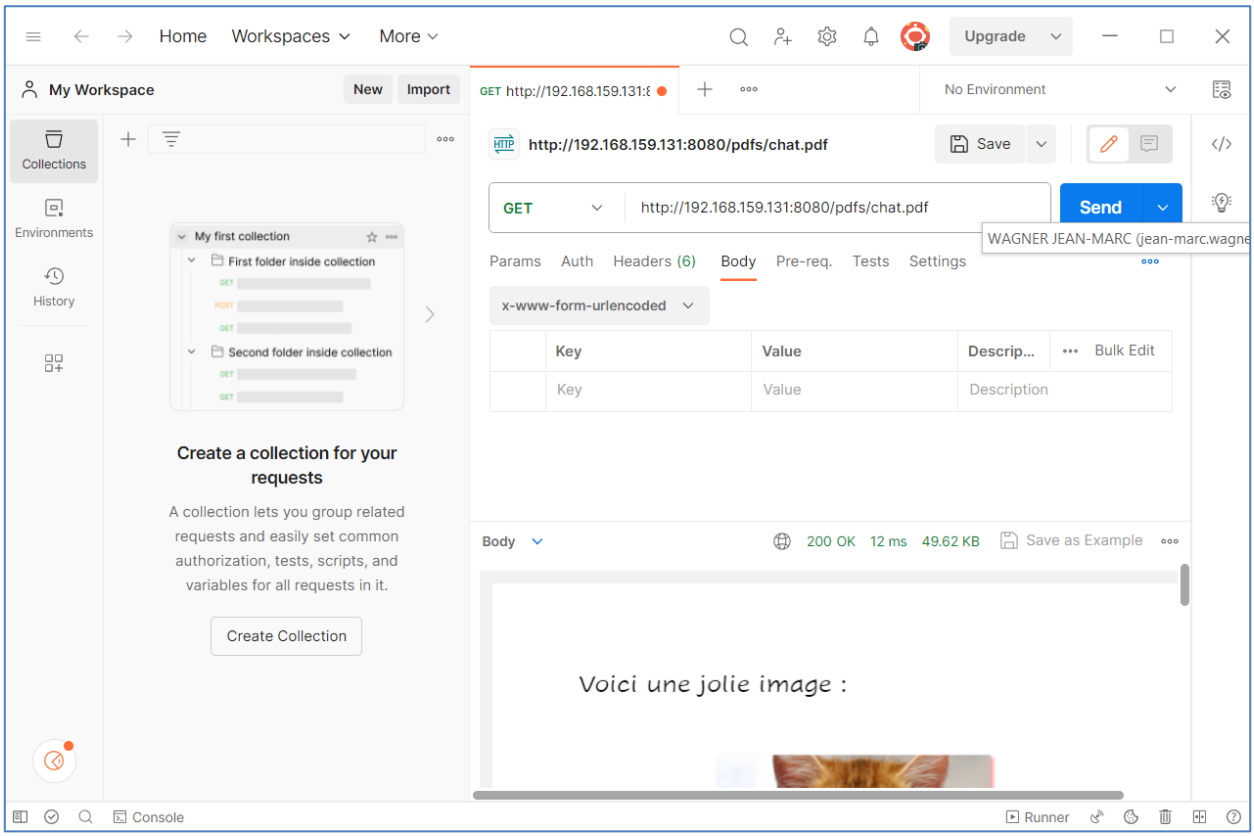
où on observe que

- lors du 1er appel de la commande **http**, nous avons délibérément spécifié une URL qui n'existe pas → cela à conduit à une erreur du serveur **code 404**
- le 2ème appel de la commande a fourni une réponse contenant l'image jpg, dont la taille est 32598 bytes mais dont le contenu n'est pas affiché

On peut encore tester notre API en utilisant **Postman**. Voici quelques exemples :

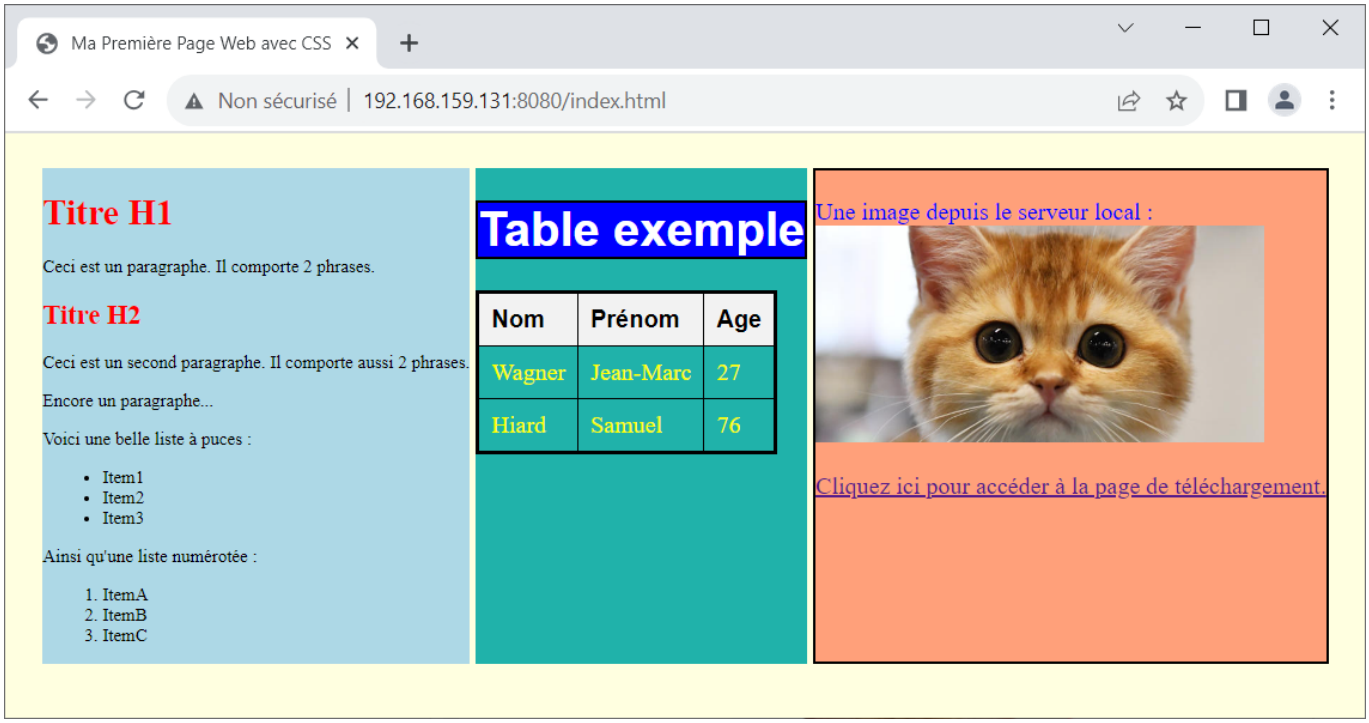




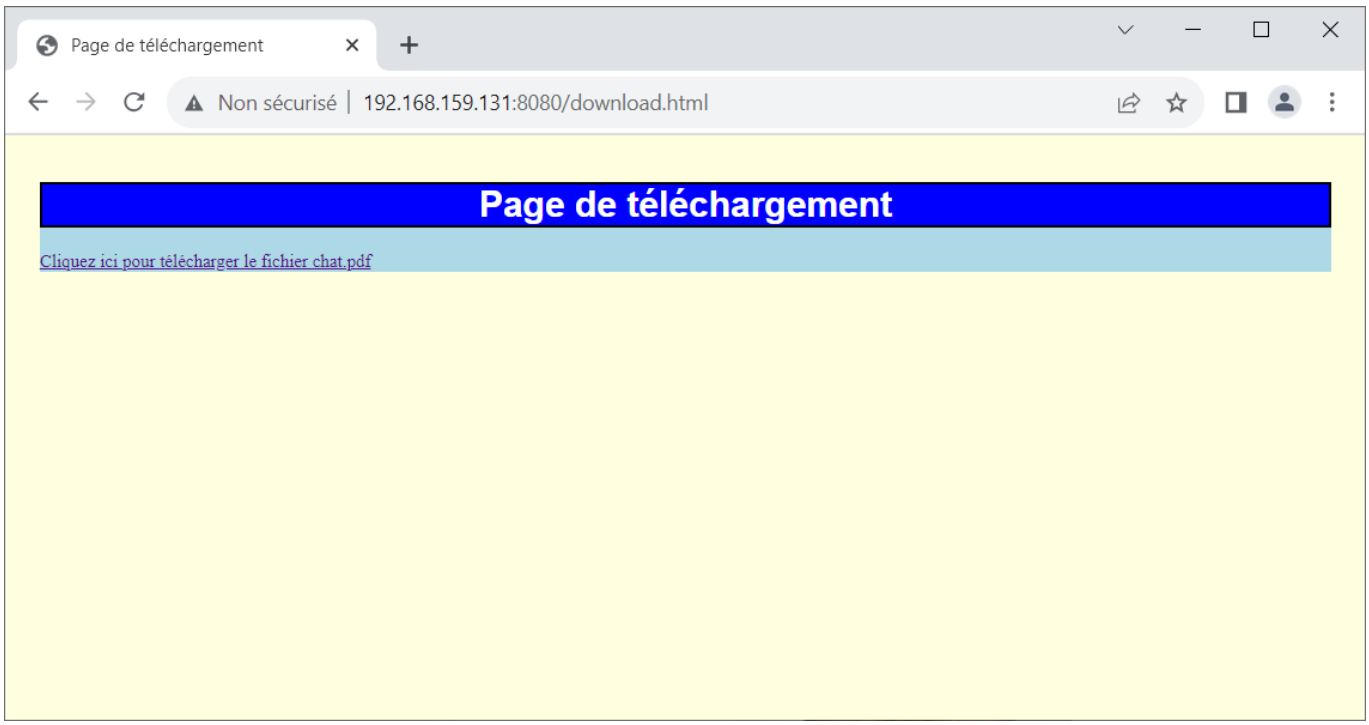


On peut ainsi tester chaque requête individuellement.

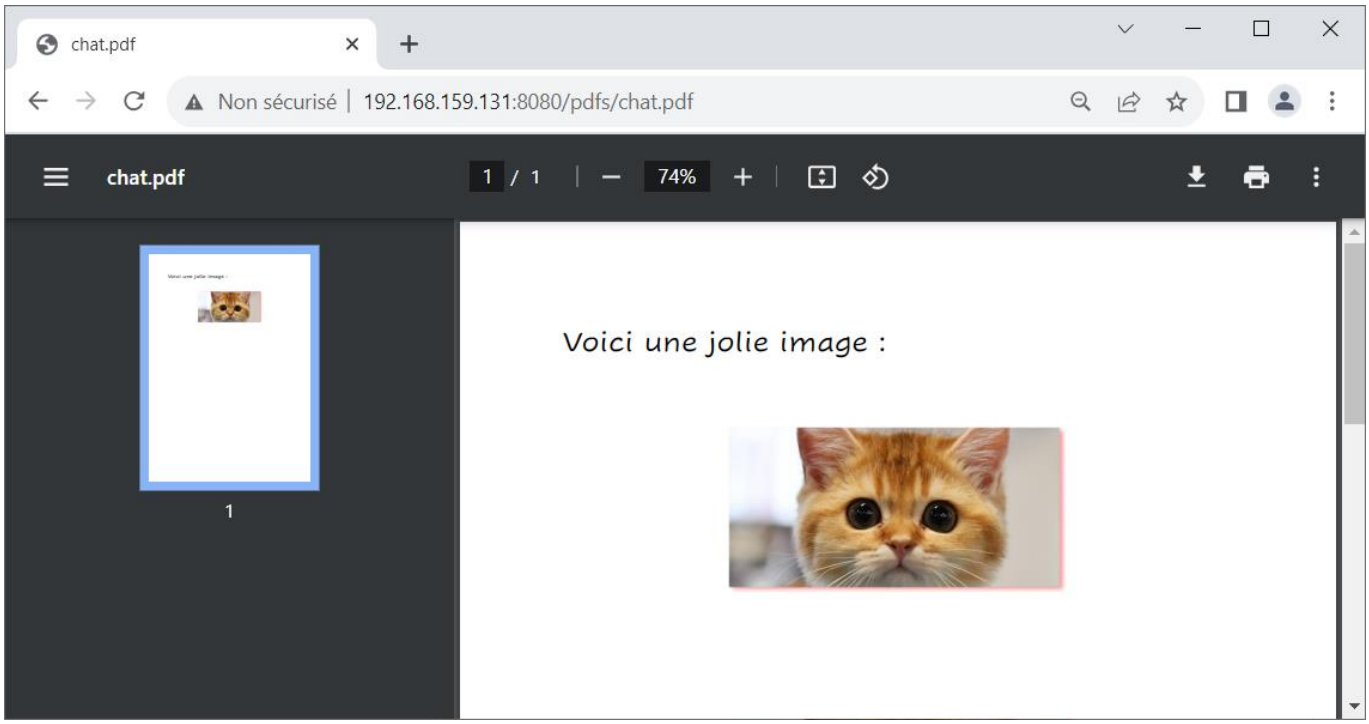
Bien évidemment, on peut tester notre site dans **Google Chrome** (par exemple) :



Si on clique sur le lien à droite, cela mène à



Et ensuite si on clique sur le lien de téléchargement, cela fournit



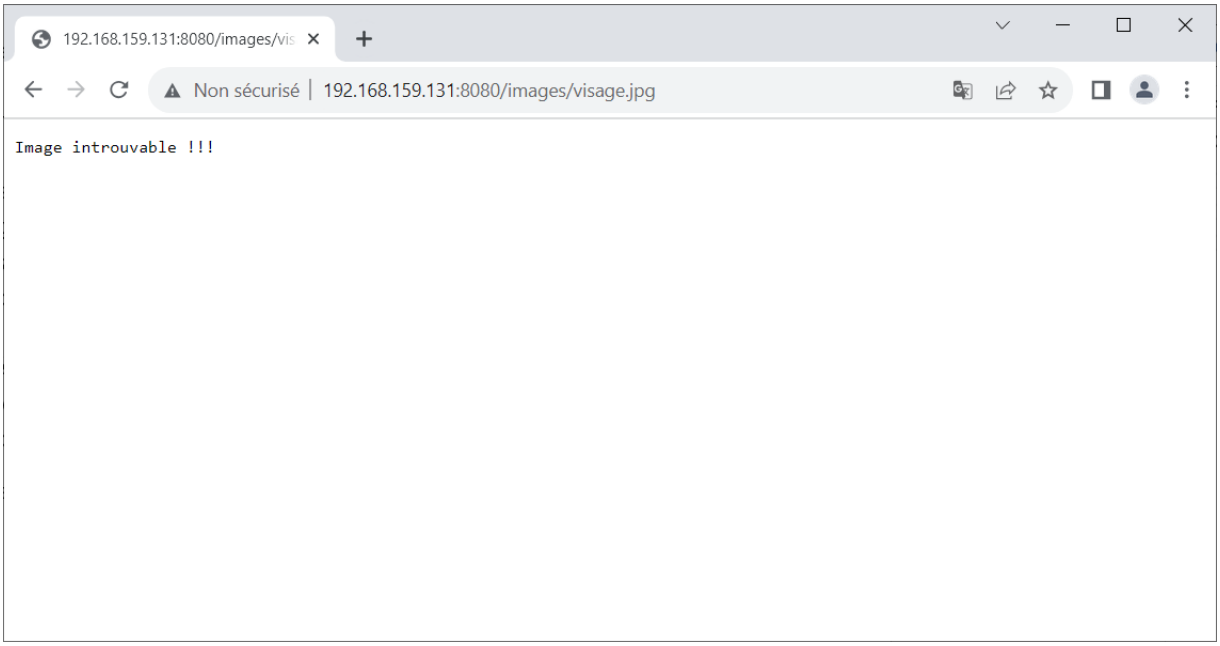
Sur la console du serveur, nous pouvons voir :

```
$ java ServeurWebHtmlCss.Main
Demarrage du serveur HTTP...
HandlerHtml (methode GET)    = /index.html --> OK
HandlerCss (methode GET)     = /css/style.css --> OK
HandlerImages (methode GET) = /images/petitchat.jpg --> OK
HandlerHtml (methode GET)    = /download.html --> OK
HandlerCss (methode GET)     = /css/style.css --> OK
HandlerPdfs (methode GET)    = /pdfs/chat.pdf --> OK
```

où on observe que

- les 3 premières lignes (en **rouge**) correspondent à l'obtention de la page index.htm : une 1ère requête a provoqué l'envoi du fichier index.html (seul, sans le fichier de style et l'image) → le browser a reçu la réponse et l'a interprétée → il s'est rendu compte qu'il y avait un fichier de style à obtenir et a donc envoyé une 2ème requête HTTP vers le serveur afin d'obtenir ce fichier → dans la même logique, le browser a ensuite envoyé une 3ème requête au serveur afin d'obtenir l'image petitchat.jpg
- Les 2 lignes en **vert** correspondent au fait que l'on cliqué sur le lien, ce qui a provoqué l'envoi d'une requête HTTP pour obtenir le fichier download.html → mais celui-ci utilise le fichier style.css et une seconde requête (2ème ligne verte) a été envoyée par le browser au serveur pour obtenir le fichier de style
- La dernière ligne (**violette**) correspond au clic sur le lien de téléchargement, ce qui provoqué l'envoi d'une requête HTTP pour obtenir le fichier chat.pdf

Si on tape une **URL invalide** dans le browser, cela fournit par exemple



Et au niveau de la console du serveur :

```
…
HandlerImages (methode GET) = /images/visage.jpg --> KO
```

Création et gestion d'un formulaire HTML

Jusque maintenant, on s'est contenté d'envoyer des requêtes de « consultation » au serveur web : récupérer des pages HTML, des images, fichiers de style CSS…

Nous allons à présent voir comment un utilisateur peut envoyer des données au serveur. Pour cela, on utilise les **formulaires HTML** qui utilisent les balises **<form>** et **<input>**

Le format général d'un formulaire HTML est le suivant :

```
<form action="URL de traitement" method="post">
    <!-- Champs de formulaire ici -->
    <!-- Libellés (labels) et éléments de saisie (input, textarea, etc.) -->
    <!-- Boutons de soumission (input type="submit) ou autre contenu -->
</form>
```

où

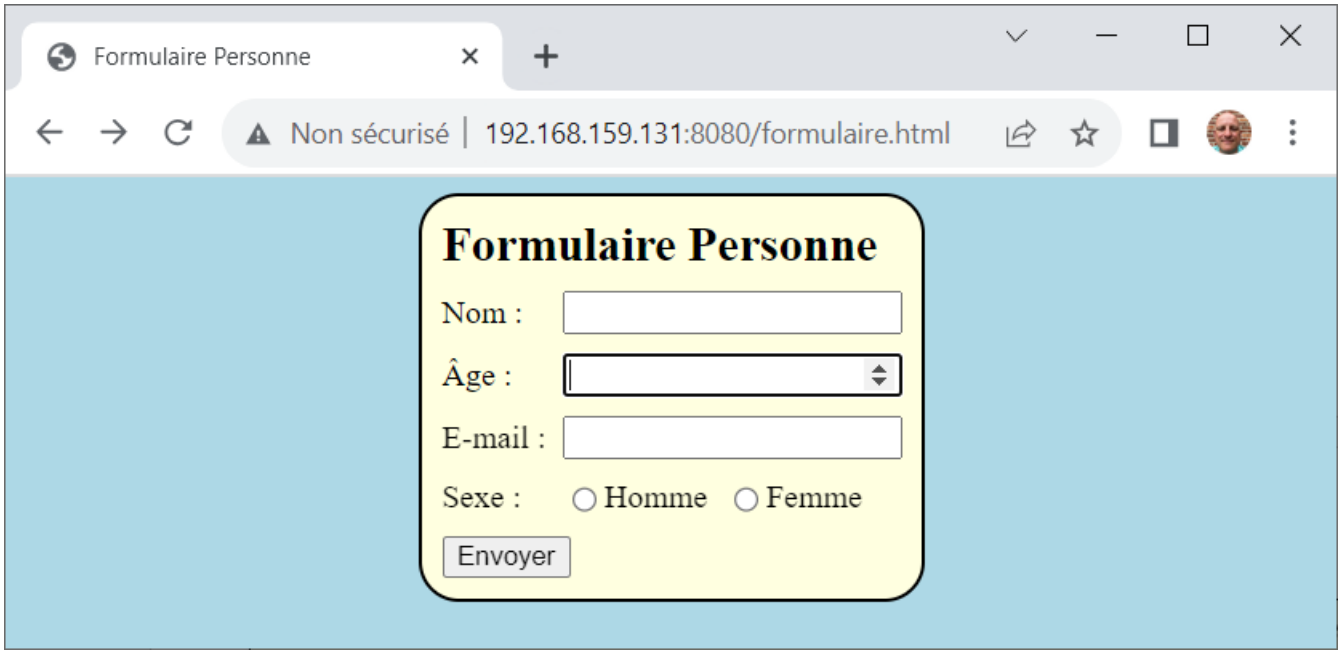
- L'**URL de traitement** est une des **"routes"** qui auront été définies dans le serveur web → le serveur web fournira alors le service demandé par ce formulaire

- **method** définit le type de la requête HTTP envoyée → par défaut, c'est la méthode GET qui est utilisée et dans ce cas les champs du formulaire passeront en clair dans l'URL, ce qui n'est pas terrible niveau sécurité → on utilise plutôt la **méthode POST** afin que les données du formulaire se trouvent dans le corps de la requête HTTP

- les données du formulaire, ainsi que le(s) bouton(s) de soumission seront mis en place à l'aide de la balise **<input>**

Exemple de formulaire HTTP et gestion par un service web Java

Dans cet exemple, nous mettons en place le formulaire suivant



Le but est simplement de transmettre les données d'une **personne** (nom, age, e-mail et sexe) au serveur. Le serveur web se contentera de renvoyer ces données au format texte.

Voici le code de la **page HTML** contenant le formulaire (fichier **formulaire.html**) :

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Formulaire Personne</title>
        <link href="css/styleForm.css" type="text/css" rel="stylesheet">
    </head>

    <body>
        <div id="conteneur">
        <div id="bloc">

        <h2 class="form-group">Formulaire Personne</h2>

        <form action="FormPersonne" method="post">
            <div class="form-group">
                <label for="nom">Nom :</label>
                <input type="text" id="nom" name="nom"><br>
            </div>

            <div class="form-group">
                <label for="age">Âge :</label>
                <input type="number" id="age" name="age"><br>
```

```
            </div>

            <div class="form-group">
                <label for="email">E-mail :</label>
                <input type="email" id="email" name="email"><br>
            </div>

            <div class="form-group">
                <label>Sexe :</label>
                <input type="radio" id="homme" name="sexe" value="Homme">
                <label for="homme">Homme</label>

                <input type="radio" id="femme" name="sexe" value="Femme">
                <label for="femme">Femme</label><br>
            </div>

            <input class="form-group" type="submit" value="Envoyer">
        </form>

        </div>
        </div>
    </body>
</html>
```

où on observe que

- pour chaque champ du formulaire, on peut définir un **type de donnée** :
  - **text** → pour un texte simple
  - **number** → pour un nombre → dans ce cas, un “spinner” apparaît
  - **email** → pour une adresse e-mail
  - **radio** → pour des “boutons radio” permettant de choisir parmi plusieurs choix imposées (ici “Homme” ou “Femme”)
  - **submit** → le bouton qui va permettre de soumettre le formulaire et donc de l’envoyer au serveur
- mais il existe d’autres types comme
  - **password** → pour les mots de passe → les caractères tapés n’apparaissent pas lors de la saisie
  - **checkbox** → permettant de choisir plusieurs options parmi celles proposées
  - **tel** → telephone
  - … → à vous de voir
- le **“name”** de chaque **input** est important → c’est grâce à ce nom que le serveur va pouvoir récupérer les champs du formulaire
- un fichier de style **styleForm.css** est utilisé afin de mettre en forme le formulaire

Voici le contenu du **fichier de style CSS** (fichier **formStyle.css**) :

```
#conteneur {
	display: flex;
	flex-direction: row;
	justify-content: space-around;
}

#bloc {
	border: 2px solid black;
	background: lightyellow;
	border-radius: 20px;
}

html {
  background: lightblue;
}

label {
  width: 60px;
}

.form-group {
  display: flex;
  flex-direction: row;
  align-items: center;
  margin-bottom: 10px;
  margin-top: 10px;
  margin-left: 10px;
  margin-right: 10px;
}
```

qui ne demande pas d'explication particulière.

Pour le déploiement de ces 2 fichiers, il suffit des les placer dans les repertoires **html** et **css** du repertoire **MonSite** contenant tous les fichiers du site web.

Mise en place du backend en Java

Il faut à present mettre à jour notre serveur web afin qu'il **traite les requêtes "POST"** correspondant à la route **"FormPersonne"**. Pour cela, nous créons un nouveau **HttpContext** dédié à celui. Celui-ci est représenté par la classe **HandlerFormulaire** dont voici le code (fichier **HandlerFormulaire.java**) :

```
import com.sun.net.httpserver.*;
import java.io.*;
import java.net.URLDecoder;
```

```
import java.nio.charset.StandardCharsets;
import java.util.*;

public class HandlerFormulaire implements HttpHandler
{
    @Override
    public void handle(HttpExchange exchange) throws IOException
    {
        if (exchange.getRequestMethod().equalsIgnoreCase("POST"))
        {
            String requestMethod = exchange.getRequestMethod();
            String requestPath = exchange.getRequestURI().getPath();
            byte[] requestBody = exchange.getRequestBody().readAllBytes();
            String formData = new String(requestBody,
StandardCharsets.UTF_8);
            System.out.println("HandlerFormulaire (methode " + requestMethod
+ ") = " + requestPath + " --" + formData + "--");
            Map<String, String> formDataMap = parseFormData(formData);

            // Traiter les données du formulaire
            String nom = formDataMap.get("nom");
            int age = Integer.parseInt(formDataMap.get("age"));
            String email = formDataMap.get("email");
            String sexe = formDataMap.get("sexe");

            String Reponse = "Nom : " + nom;
            Reponse += "\nAge : " + age;
            Reponse += "\nEmail : " + email;
            Reponse += "\nSexe : " + sexe;
            exchange.sendResponseHeaders(200, Reponse.length());
            exchange.getResponseHeaders().set("Content-Type", "text/plain");
            OutputStream outputStream = exchange.getResponseBody();
            outputStream.write(Reponse.getBytes());
            outputStream.close();
        }
        else
        {
            exchange.sendResponseHeaders(405, 0); // Méthode non autorisée -
POST requis
            exchange.close();
        }
    }

    private Map<String, String> parseFormData(String formData) throws
UnsupportedEncodingException
    {
        // Parse les données du formulaire et les retourne sous forme de Map
clé-valeur
        Map<String, String> formDataMap = new HashMap<>();
```

```
        String[] formDataPairs = formData.split("&");
        for (String pair : formDataPairs) {
            String[] keyValue = pair.split("=");
            if (keyValue.length == 2) {
                String key = URLDecoder.decode(keyValue[0],
StandardCharsets.UTF_8.name());
                String value = URLDecoder.decode(keyValue[1],
StandardCharsets.UTF_8.name());
                formDataMap.put(key, value);
            }
        }

        return formDataMap;
    }
}
```

où

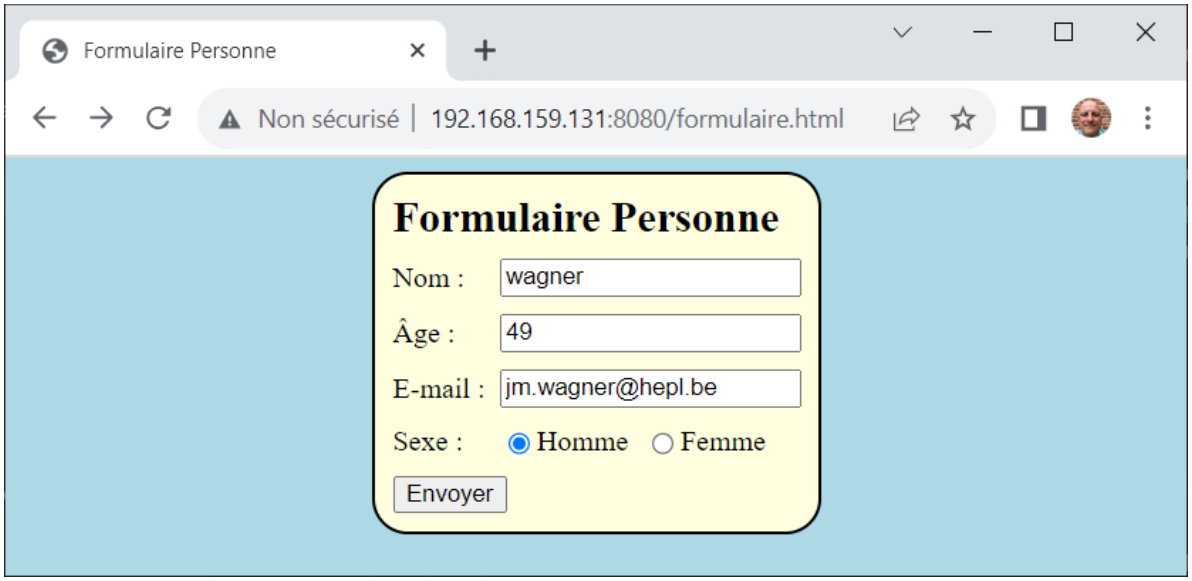
- le corps de la requête HTTP est récupéré dans la variable **requestBody** (au format byte[]) puis transformée en chaine de caractères **formData** → il est necessaire de "splitter" cette chaine afin de récupérer tous les champs du formulaire → appel à la méthode "maison" **parseFormData** qui retourne une map **formDataMap** (couples "clé/valeur")
- la réponse envoyée par le serveur est une simple chaine de caractères obtenue par concatenation des champs du formulaire → le **Content-Type** est donc fixé à **"text/plain"**
- on remarque aussi que si la méthode n'est pas POST, le service retournera une **erreur 405** (méthode non autorisée)
- bien sûr, les champs du formulaire pourraient être utilisés afin d'attaquer une base donnée par exemple.

Reste à mettre en place ce **HttpContext** en place dans notre serveur web en ajoutant la ligne

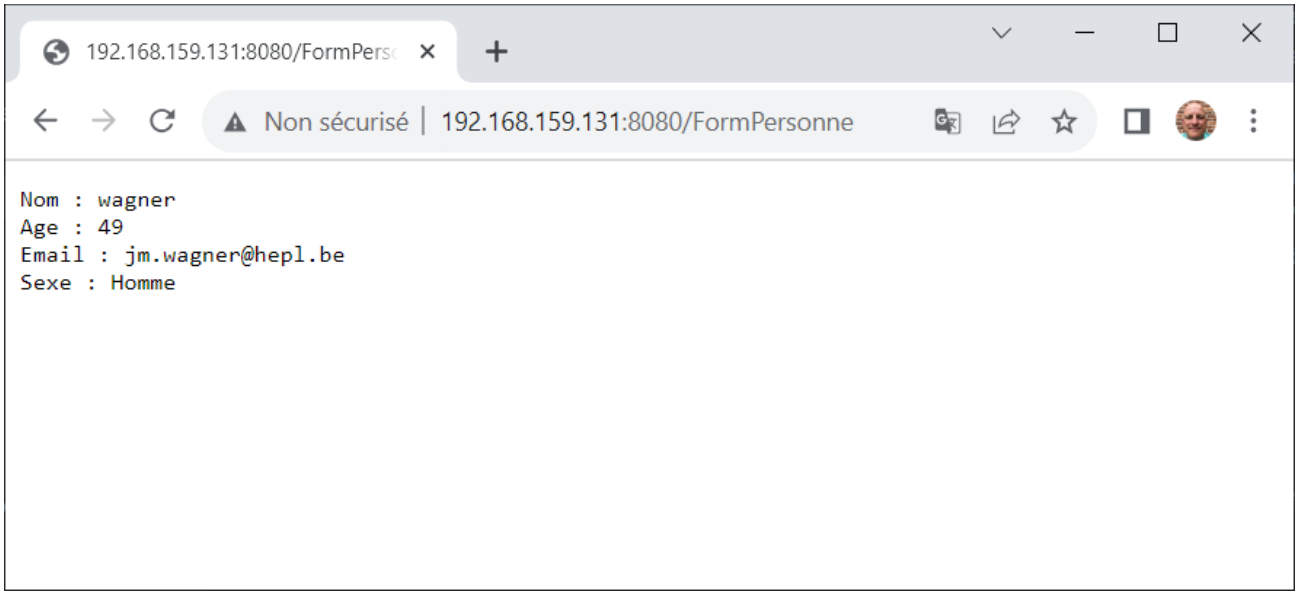
```
serveur.createContext("/FormPersonne",new HandlerFormulaire());
```

qui fait le lien avec la route **"FormPersonne"** apparaissant dans le fichier HTML.

Une fois le serveur web lancé, cela donne dans Google Chrome :



Après avoir encodé les champs et cliqué sur le bouton “Envoyer”, cela mène alors à



où on voit bien que la réponse envoyée par le serveur est simplement du **texte brut**.

Dans la console du serveur web, nous observons

```
$ java ServeurWebHtmlCss.Main
Demarrage du serveur HTTP...
HandlerHtml (methode GET)    = /formulaire.html --> OK
HandlerCss (methode GET)     = /css/styleForm.css --> OK
HandlerFormulaire (methode POST) = /FormPersonne --
nom=wagner&age=49&email=jm.wagner%40hepl.be&sexe=Homme--
```

On remarque bien que le corps de la requête est une chaine de caractères contenant tous les champs et les données du formulaire.

Définition et création d'une API REST

Jusque maintenant, nous avons vu qu'un client peut récupérer des ressources (fichiers HTML, CSS, …) à partir d'une serveur web en utilisant une requête HTTP de type **GET**, mais également comment un client peut envoyer des données à un serveur web en utilisant une requête HTTP de type **POST**. Une formulation et une généralisation de ces techniques a donné naissance au début des années 2000 à une architecture de développement appelée **REST**.

L'architecture **REST** (**REpresentational State Tranfer**) est un style d'architecture logicielle largement utilisé aujourd'hui pour concevoir des systèmes informatiques, en particulier des **services web** simples, évolutifs et efficaces.

Les principes généraux de **REST** sont :

- **Ressources** : Dans une architecture REST, tout est ressource qu'il s'agisse de données, d'objets ou de services → chaque ressource est identifiée par une URI (Uniform Resource Identifier)
- **Actions basées sur les verbes HTTP** : les opérations sur les ressources sont définies en utilisant les méthodes HTTP appropriées → les méthodes HTTP couramment utilisées sont
  - **GET** pour la récupération de données
  - **POST** pour la création de nouvelles ressources
  - **PUT** pour la mise à jour de ressources existantes
  - **DELETE** pour la suppression de ressources
- **Représentation** : les ressources peuvent avoir différentes représentations (**JSON**, **XML**, **HTML**, …) pour répondre aux besoins des clients → les clients peuvent spécifier le type de représentation qu'ils préfèrent en utilisant les en-têtes HTTP
- **Etat sans session** : chaque requête HTTP doit contenir toutes les informations nécessaires pour comprendre la demande → le serveur ne stocke pas d'état de session côté serveur entre les requêtes du client

- **Liens hypertexte (HATEOS)** : Utilisation de liens hypertexte pour permettre aux clients de naviguer à travers les ressources → chaque réponse doit contenir des liens vers d'autres ressources associées
- **Système client/serveur** : les responsabilités du client et du serveur sont clairement définies → le client est responsable de l'interface utilisateur et de la gestion de l'état, tandis que le serveur est responsable du stockage des données et de la gestion des ressources
- **Sans état (stateless)** : chaque requête client au serveur doit être indépendante et ne doit pas dépendre d'une requête précédente → le serveur ne conserve pas d'état des clients entre les requêtes

Lors de la création d'une API web, Il est difficile de respecter scrupuleusement toutes les contraintes citées ci-dessus. Une API qui respecte scrupuleusement tous ces contraintes est qualifiée d'**API RESTful**, sinon elle sera simplement qualifiée d'**API REST**.

L'acronyme **REST** (donc **RE**presentational **S**tate **T**ransfer) est justifié par

- **RE**presentational → cela fait référence à la manière dont les données sont représentées dans le système → dans une architecture REST, le ressources (telles que des utilisateurs, des tâches, des produits, …) sont représentées sous une forme spécifique, généralement au format **JSON** ou **XML**, qui permet de décrire leur structure et leurs attributs
- **S**tate → l'idée clé est que les ressources représentent l'état d'un système à un moment donné. Par exemple, une ressource « Liste de tâches » pour représenter l'état actuel d'une liste de tâches (nombre de tâches et leur description) → le terme « état » rappelle que les ressources sont des instantanés de données à un moment précis
- **T**ransfer → cela fait référence au transfert de ces représentations d'état entre un client et un serveur → les clients (applications, navigateurs, …) peuvent envoyer des requêtes HTTP au serveur pour manipuler des ressources (GET, POST, PUT, DELETE)

Le format JSON

JSON (**JavaScript Object Notation**) est un format léger et largement utilisé pour représenter et échanger des données structurées → il est facile à écrire et à lire pour les humains, et facile à analyser et à générer pour les applications logicielles. **JSON** est utilisé dans de nombreuses applications :

- les web services
- les communications client/serveur
- le stockage des données
- …

La structure générale des données dans un format **JSON** est la suivante :

- **Objet JSON (JSON Object)** : un **objet JSON** est une collection non ordonnée de paires clé-valeur
  - il est délimité par des accolades **{ }**
  - chaque paire de clé-valeur est séparée par une **virgule**
  - la **clé** est toujours une chaîne de caractères
  - la **valeur** peut être de différents types, y compris un autre objet JSON

Exemple d'un objet JSON :

```
{
   "nom": "Jean Dupont",
   "age": 30,
   "adresse": {
       "rue": "123 Rue de la Gaufre",
       "ville": "Bruxelles"
   }
}
```

Cet objet JSON comporte 3 **attributs** : un nom (chaîne de caractères), un age (un nombre), une adresse (un autre objet JSON comportant 2 attributs de type chaînes de caractères).

- **Tableau JSON (JSON Array)** : un **tableau JSON** est une séquence ordonnée de valeurs
  - Il est délimité par des crochets **[ ]**
  - Les valeurs à l'intérieur du tableau sont séparées par des **virgules**

Exemple de tableau JSON :

```
[
    "pomme",
    "banane",
    "orange"
]
```

- **Valeurs JSON** : les **valeurs JSON** peuvent être de plusieurs types :
  - Chaînes de caractères (ex : “Hello World !”)
  - Nombres (ex : 42 ou 3.1416)
  - Booléens (true ou false)
  - Null (valeur vide)
  - Objets JSON
  - Tableaux JSON

Il est important de noter que la **clé** dans un objet JSON doit être une chaîne de caractères et doit être entourée de guillements doubles **“ “** conformément à la spécification JSON.

Exemple d’API REST

Prenons l’exemple d’une **API REST** responsible de la gestion d’une **liste de tâches** simples qui permet de créer, lire, mettre à jour et supprimer des tâches à l’aide des méthodes HTTP appropriées :

- **Lecture (GET) :**
  - “**GET /api/tasks**” pour obtenir la liste complete des tâches
- **Création (POST) :**
  - “**POST /api/tasks**” pour créer une nouvelle tâche
  - Le corps de la requête POST contiendra la description de la tâche sous la forme d’une chaîne de caractères
- **Mise à jour (PUT) :**
  - “**PUT /api/tasks/id=2**” pour mettre à jour la tâche 2 (par exemple) de la liste des tâches
  - Le corps de la requête PUT contiendra la nouvelle description de la tâche
- **Suppression (DELETE) :**
  - “**DELETE /api/tasks/id=4**” pour supprimer la tâche 4 (par exemple) de la liste des tâches

Donc **chaque opération** est associée à une **méthode HTTP** (GET, POST, PUT, DELETE) et à une **URI** qui identifie la resource (dans cet exemple, la liste de tâches) sur laquelle l'opération doit être effectuée.

Les **réponses de l'API** (contenue dans le corps des réponses HTTP) pourraient être de simples chaînes de caractères dans le cas de POST, PUT et GET, et dans le cas du GET, cela pourrait être un **tableau JSON** d'objets contenant chacun le numéro de la tâche ainsi que sa description :

```
[
    {
        "id": 1,
        "task": "Laver la voiture"
    },
    {
        "id": 2,
        "task": "Aller faire les courses"
    },
    {
        "id": 3,
        "task": "Preparer le souper"
    }
]
```

Cette **API REST** simple permet aux clients qui l'utilisent d'effectuer des operations **CRUD** (**Create – Read – Update – Delete**) sur une liste de tâches de manière simple, cohérente et standardisée.

Exemple d'API REST « To Do List » en Java

Dans l'exemple qui suit, nous implémentons l'exemple d'API REST décrit ci-dessus : la **liste de tâches**. Pour cela, nous utilisons à nouveau la classe **HttpServer** de Java. Pour les tests de cette API, nous utiliserons le logiciel **Postman** mais aussi la commande **http**.

Voici le code de notre **API REST** (fichier **TodoApi.java**) :

```
import com.sun.net.httpserver.*;
import java.io.*;
import java.util.*;
import java.net.*;
```

```
public class TodoApi
{
    private static List<String> tasks = new ArrayList<>();

    public static void main(String[] args) throws IOException
    {
        System.out.println("API Rest demarree...");
        HttpServer server = HttpServer.create(new InetSocketAddress(8080), 0);
        server.createContext("/api/tasks", new TaskHandler());
        server.start();
    }

    static class TaskHandler implements HttpHandler
    {
        @Override
        public void handle(HttpExchange exchange) throws IOException
        {
            String requestMethod = exchange.getRequestMethod();

            if (requestMethod.equalsIgnoreCase("GET"))
            {
                System.out.println("--- Requête GET reçue (obtenir la liste) ---");
                // Récupérer la liste des tâches au format JSON
                String response = convertTasksToJson();
                sendResponse(exchange, 200, response);
            }
            else if (requestMethod.equalsIgnoreCase("POST"))
            {
                System.out.println("--- Requête POST reçue (ajout) ---");
                // Ajouter une nouvelle tâche
                String requestBody = readRequestBody(exchange);
                System.out.println("requestNody = " + requestBody);
                addTask(requestBody);
                sendResponse(exchange, 201, "Tache ajoutee avec succes");
            }
            else if (requestMethod.equalsIgnoreCase("PUT"))
            {
                System.out.println("--- Requête PUT reçue (mise a jour) ---");
                // Mettre à jour une tâche existante
                Map<String, String> queryParams =
parseQueryParams(exchange.getRequestURI().getQuery());
                if (queryParams.containsKey("id"))
                {
                    int taskId = Integer.parseInt(queryParams.get("id"));
                    System.out.println("Mise a jour tache id=" + taskId);
                    String requestBody = readRequestBody(exchange);
                    System.out.println("requestNody = " + requestBody);
                    updateTask(taskId, requestBody);
                    sendResponse(exchange, 200, "Tache mise a jour avec succes");
                }
                else sendResponse(exchange, 400, "ID de tache manquant dans les
parametres");

            }
            else if (requestMethod.equalsIgnoreCase("DELETE"))
            {
```

```
                System.out.println("--- Requête DELETE reçue (suppression) ---");
                // Supprimer une tâche
                Map<String, String> queryParams =
parseQueryParams(exchange.getRequestURI().getQuery());
                if (queryParams.containsKey("id"))
                {
                    int taskId = Integer.parseInt(queryParams.get("id"));
                    System.out.println("Suppression tache id=" + taskId);
                    deleteTask(taskId);
                    sendResponse(exchange, 200, "Tache supprimee avec succes");
                }
                else sendResponse(exchange, 400, "ID de tache manquant dans les
parametres");
            }
            else sendResponse(exchange, 405, "Methode non autorisee");

        }
    }

    private static void sendResponse(HttpExchange exchange, int statusCode, String
response) throws IOException
    {
        System.out.println("Envoi de la réponse (" + statusCode + ") : --" + response
+ "--");
        exchange.sendResponseHeaders(statusCode, response.length());
        OutputStream os = exchange.getResponseBody();
        os.write(response.getBytes());
        os.close();
    }

    private static String readRequestBody(HttpExchange exchange) throws IOException
    {
        BufferedReader reader = new BufferedReader(new
InputStreamReader(exchange.getRequestBody()));
        StringBuilder requestBody = new StringBuilder();
        String line;
        while ((line = reader.readLine()) != null)
        {
            requestBody.append(line);
        }
        reader.close();
        return requestBody.toString();
    }

    private static Map<String, String> parseQueryParams(String query)
    {
        Map<String, String> queryParams = new HashMap<>();
        if (query != null)
        {
            String[] params = query.split("&");
            for (String param : params)
            {
                String[] keyValue = param.split("=");
                if (keyValue.length == 2)
                {
                    queryParams.put(keyValue[0], keyValue[1]);
```

```
                }
            }
        }
        return queryParams;
    }

    private static String convertTasksToJson()
    {
        // Convertir la liste des tâches en format JSON
        StringBuilder json = new StringBuilder("[");
        for (int i = 0; i < tasks.size(); i++)
        {
            json.append("{\"id\": ").append(i + 1).append(", \"task\":
\"").append(tasks.get(i)).append("\"}");
            if (i < tasks.size() - 1) json.append(",");
        }
        json.append("]");
        return json.toString();
    }

    private static void addTask(String task)
    {
        tasks.add(task);
    }

    private static void updateTask(int taskId, String updatedTask)
    {
        if (taskId >= 1 && taskId <= tasks.size())
        {
            tasks.set(taskId - 1, updatedTask);
        }
    }

    private static void deleteTask(int taskId)
    {
        if (taskId >= 1 && taskId <= tasks.size())
        {
            tasks.remove(taskId - 1);
        }
    }
}
```

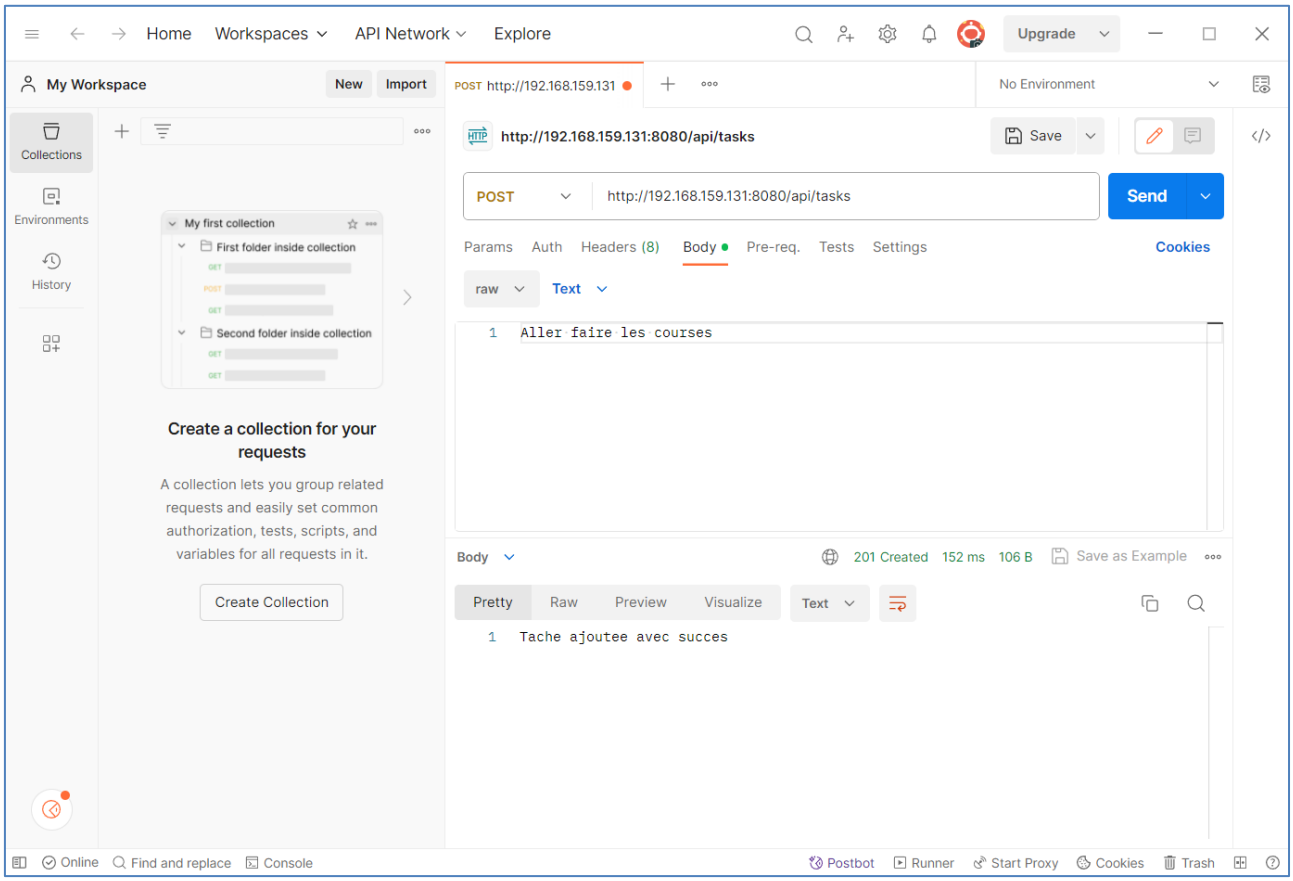
où

- La variable statique **tasks** de type **List<String>** représente la liste des tâches
- Les méthodes **addTask**, **updateTask** et **deleteTask** gèrent la logique métier de l'API, en permettant l'ajout, la mise à jour et la suppression d'une tâche
- La méthode **convertTasksToJson** permet de créer une chaîne de caractères dont le contenu est un tableau JSON contenant autant d'objets JSON qu'il y a de tâches dans la liste des tâches. Ces objets ont 2 attributs : le numéro de la tâche ("**id**"), ainsi que le descriptif de la tâche ("**task**").

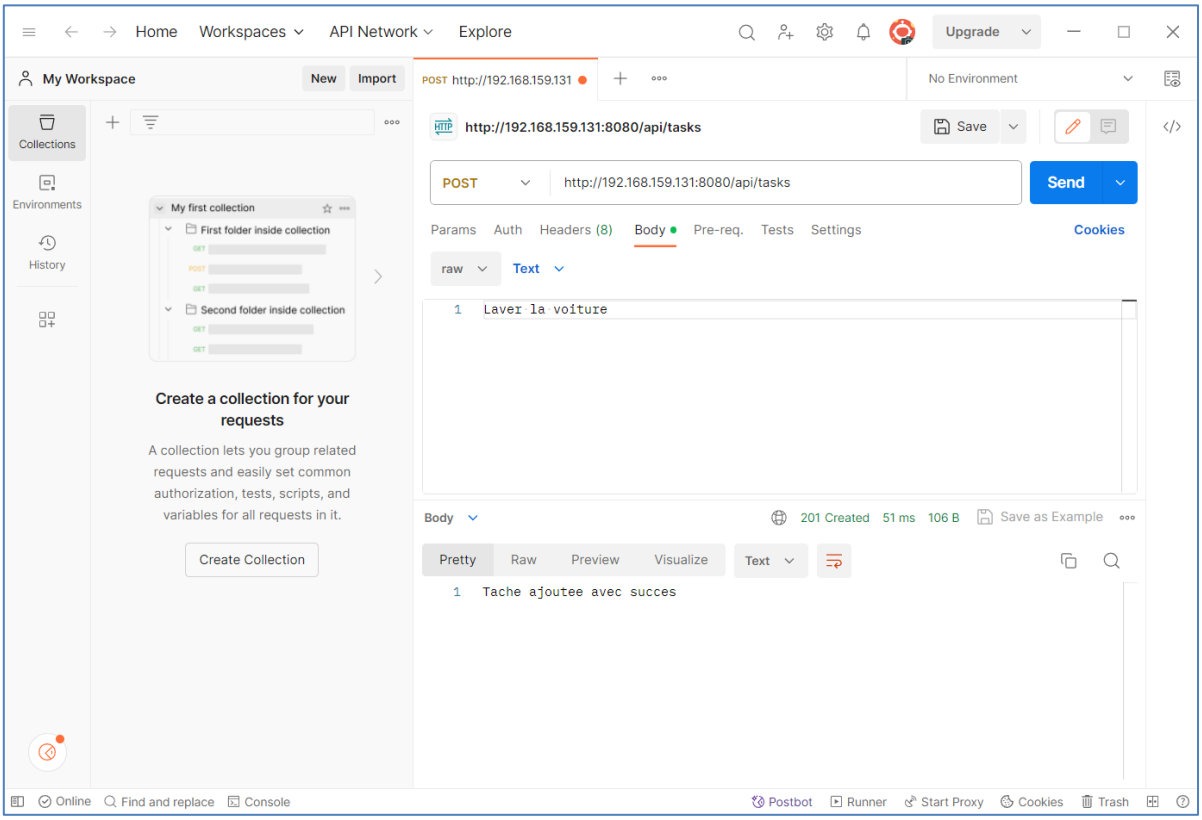
- La méthode **readRequestBody** permet de lire le corps d'une requête HTTP (à partir de l'objet **exchange**) et le retourner sous la forme d'un objet String
- La méthode **parseQueryParams** permet de récupérer les paramètres de la requête situés dans l'URL de la requête elle-même → ces paramètres sont retournés sous la forme d'un objet **Map<String,String>**
- La méthode **sendResponse** permet d'envoyer la réponse au client en précisant le code HTTP résultant, ainsi que le corps de la requête
- La méthode **handle** de la classe **TaskHandler** fait office de "contrôleur" :
    - elle réalise le parsing de la requête HTTP et en identifie le verbe (GET, POST, PUT ou DELETE)
    - elle récupère les paramètres éventuels de la requête
    - elle appelle la méthode métier adequate
    - elle répond au client en précisant le code HTTP adéquat

Une fois l'API REST démarée, on peut la tester dans **Postman** :

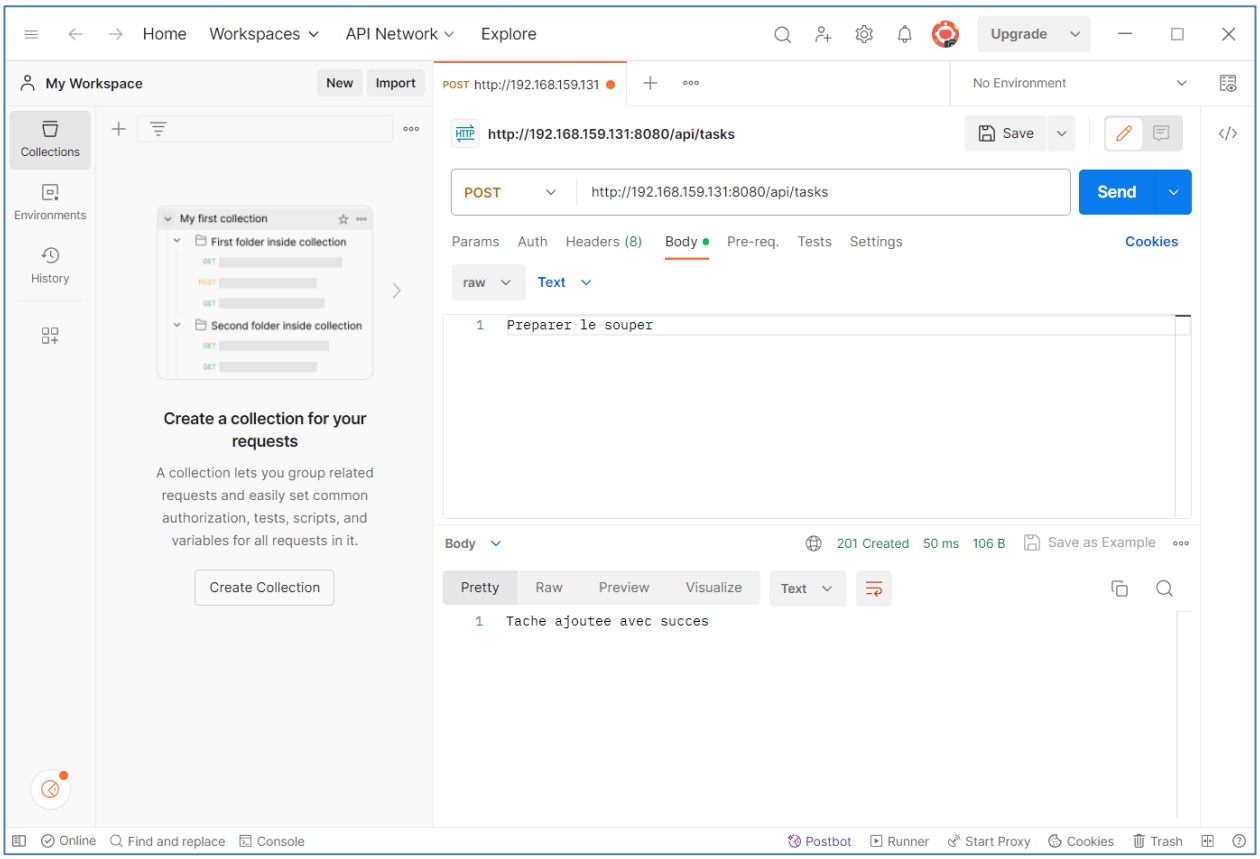
Un premier ajout (**POST**) fournit



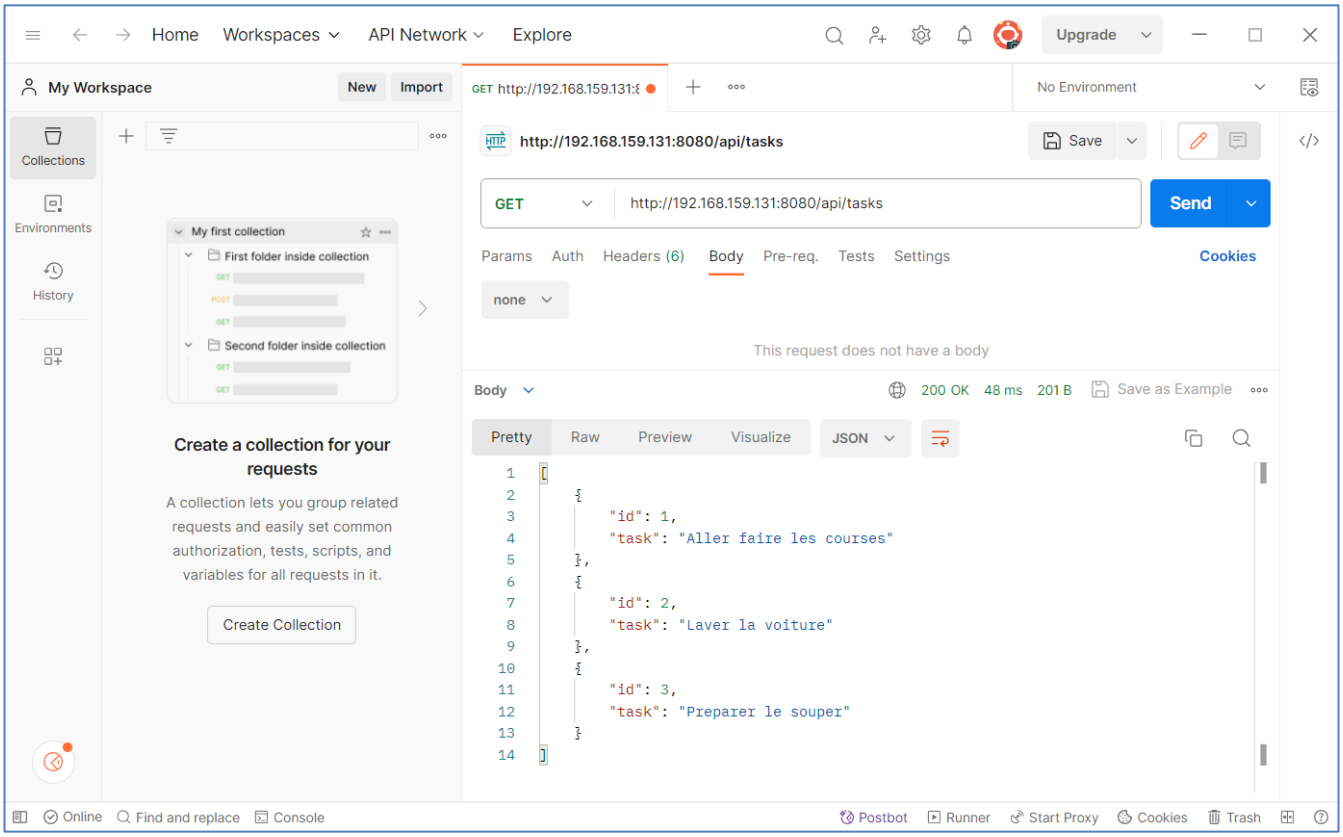
Un second ajout (**POST**) fournit



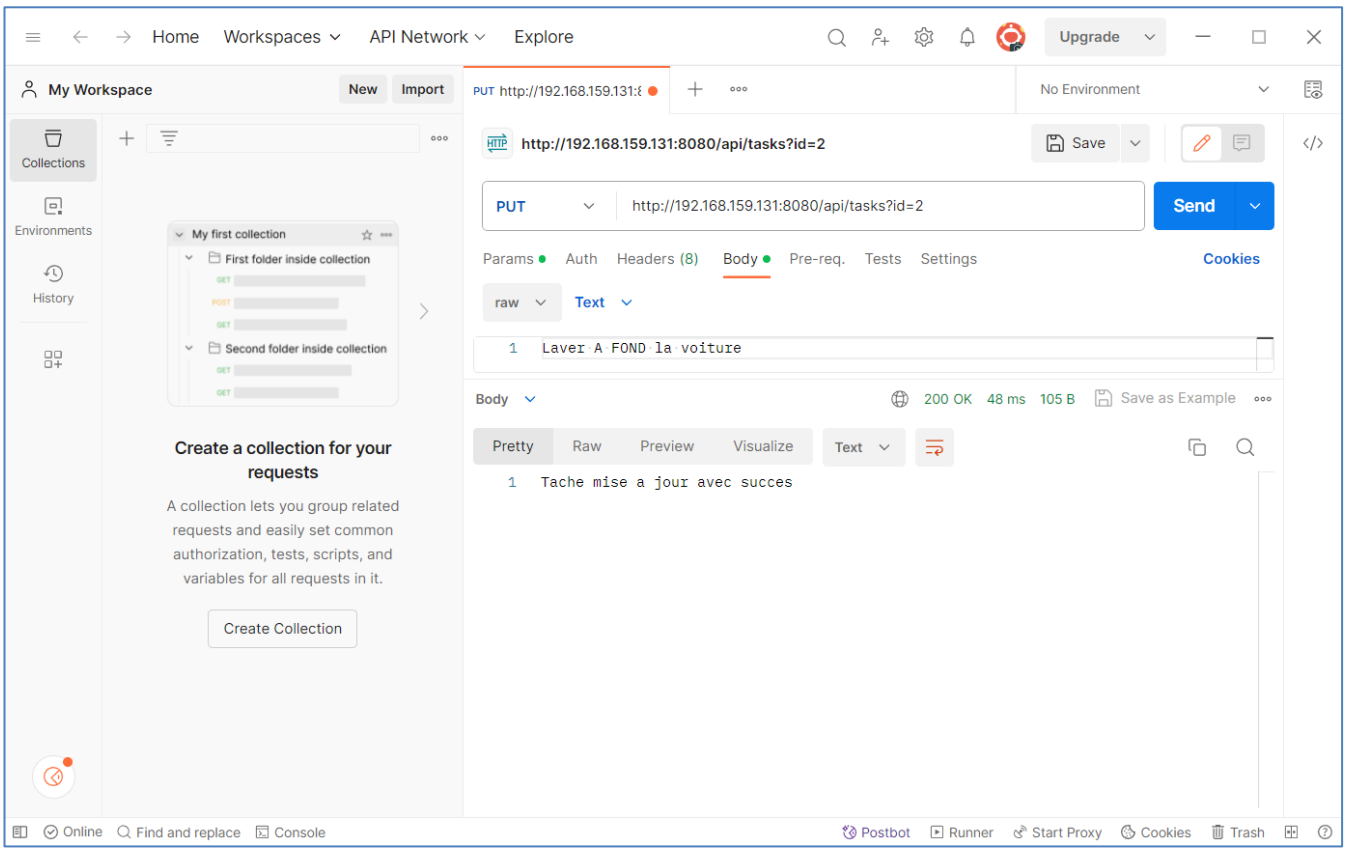
Un dernier ajout (**POST**) fournit



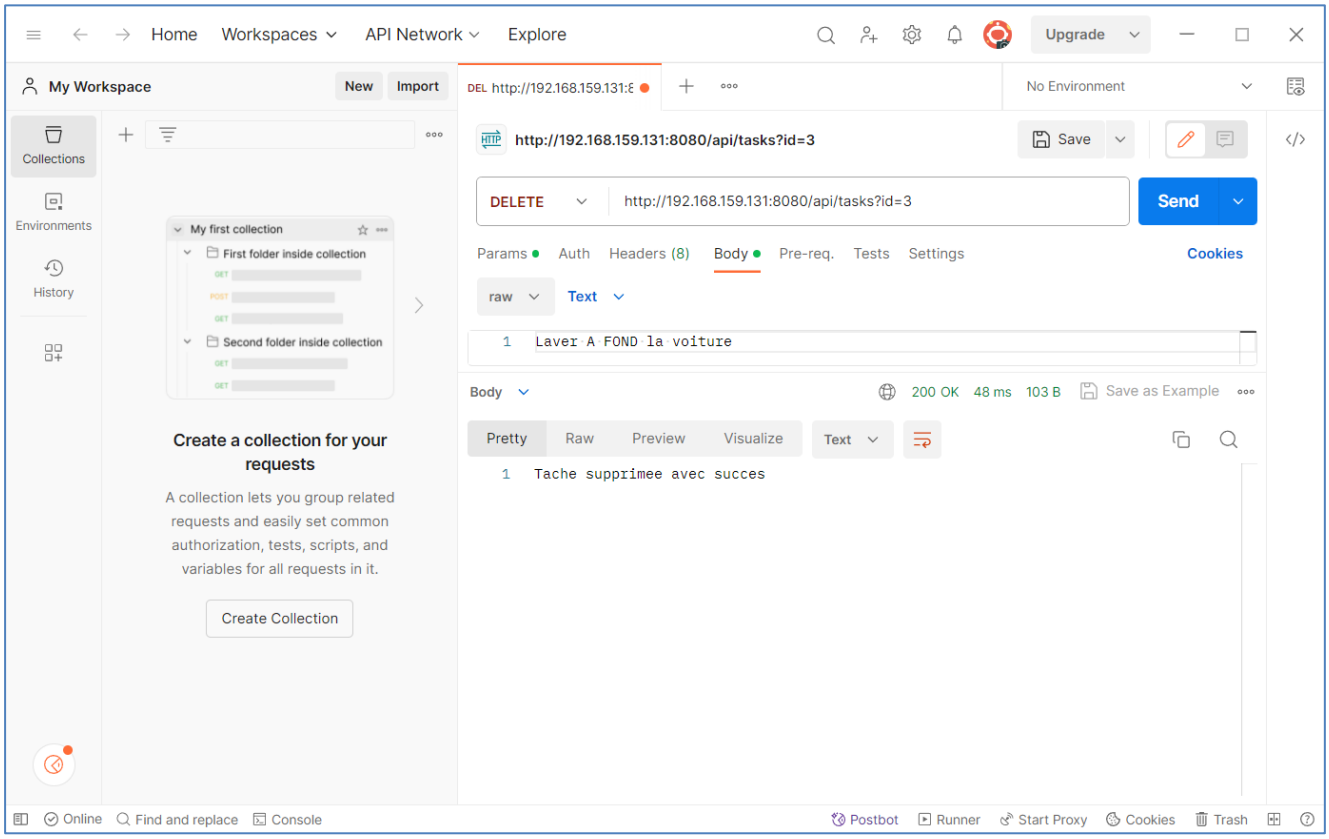
Une consultation de la liste des tâches (**GET**) fournit



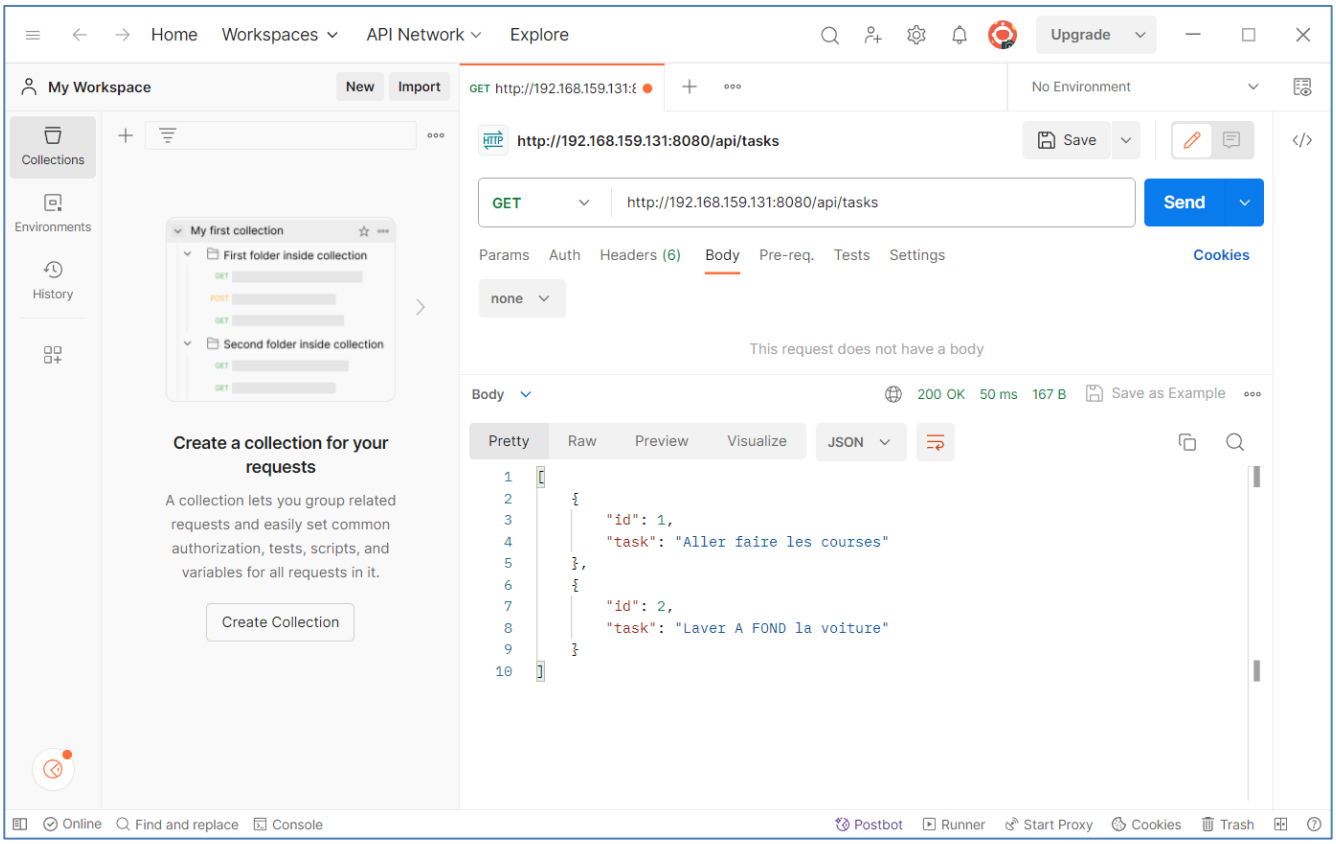
Une modification d'une tâche (**PUT**) fournit



Une suppression d'une tâche (**DELETE**) fournit



Et enfin une dernière consultation (**GET**) fournit



Mais on peut encore réaliser une consultation (**GET**) à l'aide de la commande **http** de linux :

```
$ http http://192.168.159.131:8080/api/tasks --verbose
GET /api/tasks HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: 192.168.159.131:8080
User-Agent: HTTPie/2.6.0



HTTP/1.1 200 OK
Content-length: 91
Date: Thu, 05 Oct 2023 14:30:54 GMT

[{"id": 1, "task": "Aller faire les courses"},{"id": 2, "task": "Laver A FOND
la voiture"}]

$
```

Tandis que dans la console du serveur, on peut observer

```
# java TodoApi
API Rest demarree...
--- Requête POST reçue (ajout) ---
requestNody = Aller faire les courses
Envoi de la réponse (201) : --Tache ajoutee avec succes--
--- Requête POST reçue (ajout) ---
requestNody = Laver la voiture
Envoi de la réponse (201) : --Tache ajoutee avec succes--
--- Requête POST reçue (ajout) ---
requestNody = Preparer le souper
Envoi de la réponse (201) : --Tache ajoutee avec succes--
--- Requête GET reçue (obtenir la liste) ---
Envoi de la réponse (200) : --[{"id": 1, "task": "Aller faire les
courses"},{"id": 2, "task": "Laver la voiture"},{"id": 3, "task": "Preparer
le souper"}]--
--- Requête PUT reçue (mise a jour) ---
Mise a jour tache id=2
requestNody = Laver A FOND la voiture
Envoi de la réponse (200) : --Tache mise a jour avec succes--
--- Requête DELETE reçue (suppression) ---
Suppression tache id=3
Envoi de la réponse (200) : --Tache supprimee avec succes--
--- Requête GET reçue (obtenir la liste) ---
Envoi de la réponse (200) : --[{"id": 1, "task": "Aller faire les
courses"},{"id": 2, "task": "Laver A FOND la voiture"}]--
```

```
--- Requête GET reçue (obtenir la liste) ---
Envoi de la réponse (200) : --[{"id": 1, "task": "Aller faire les courses"},{"id": 2, "task": "Laver A FOND la voiture"}]--
```

Interrogation d'une API REST par un frontend Javascript

Nous allons à présent mettre en place un frontend **Javascript** (+ HTML + CSS) capable d'interroger notre API REST. Afin de comprendre ce qui suit, il est conseillé de lire la section **« Introduction à Javascript »** ci-dessous.

Pour cela, nous utilisons **AJAX** (« **A**synchronous **JA**vascript and **X**ML » - soit « Javascript et XML asynchrone) :

- cette technologie va permettre à un script Javascript de construire et d'envoyer une **requête HTTP** à un serveur web et de récupérer la réponse envoyée par ce serveur
- ceci se fera sans devoir recharger la page web ou être redirigé vers une autre page web → cela va permettre la création d'une application web du type « **S**ingle **P**age **A**pplication » (**SPA**)
- la **charge utile** des requêtes/réponses HTTP auront pour format
  - du **texte simple**
  - des données au format **HTML**
  - des données au format **XML** (« e**X**tensible **M**arkup **L**anguage ») → permet de stocker des données dans un langage de balisage semblable à HTML
  - des données au format **JSON** → le plus courant
- la classe Javascript utilisée est **XMLHttpRequest**

L'objet XMLHttpRequest

Principe :

- une **requête HTTP** est envoyée à l'adresse spécifiée d'un serveur web
- la réponse est alors attendue de la part du serveur web

L'utilisation d'un objet de ce type se fait en 2 étapes bien distinctes :

1. Préparation et envoi de la requête
2. Réception de la réponse et extraction des données

Il est tout d'abord nécessaire d'instancier un objet **XMLHttpRequest** :

```
var xhr = new XMLHttpRequest();
```

La préparation de la requête se fait par le biais de la méthode **open()** dont voici les prototypes :

- **XMLHttpRequest.open(method, url);**
- **XMLHttpRequest.open(method, url, async);**
- **XMLHttpRequest.open(method, url, async, user);**
- **XMLHttpRequest.open(method, url, async, user, password);**

Les **2 premiers** paramètres sont **obligatoires** tandis que les **3 derniers** sont **optionnels** :

- **method** correspond à la **méthode HTTP** : GET, POST, DELETE, …
- **url** correspond à l'**URL** à laquelle on souhaite envoyer la requête
- **async** est un booléen qui par défaut vaut **true**,
  - **true** signifie que la requête sera **aysnchrone** → le script ne reste pas bloqué en attente de la réponse
  - **false** signifie que la requête sera **synchrone** → le script reste bloqué jusqu'à réception de la réponse
- **user** et **password** correspondent aux login et mot de passe en cas d'identification nécessaire sur le serveur

En considérant notre API REST développée précédemment, si nous prenons le cas de la requête **GET**, cela donnerait

```
xhr.open("GET","http://192.168.159.131:8081/api/tasks",true);
xhr.responseType = "json";
```

si notre API « **api/tasks** » tourne sur une machine dont le **domaine** est « 192.168.159.131:8081 ». La seconde ligne précise que le réponse attendue (la charge utile) est au format JSON.

Notre requête est à présent prête (pas de corps ici). Reste à l'envoyer avec la méthode send() :

```
xhr.send();
```

Si la requête est choisie « **synchrone** », le script est bloqué sur la méthode **send()** jusqu'au moment où la réponse est reçue ou que la requête a échoué → cette méthode est déconseillée car elle « frise » l'interface utilisateur pendant un certain temps

Les requêtes « **asynchrones** » sont donc à privilégier → mais alors comment est-on prévenu que la réponse est arrivée ? → par le mécanisme des **événements** de Javascript

Ceci se réalise en ajoutant un « EventListener » à notre objet **xhr** :

```
xhr.addEventListener('readystatechange', function() {
    // code à exécuter en cas de réception de l'événement 'readystatechange'
}) ;
```

La fonction créée à la volée porte le nom de « **callback** ».

Cependant, cet événement ne se déclenche pas seulement lorsque la requête est terminée, mais plutôt à chaque changement d'état. Il existe 5 états différents représentés par des constantes spécifiques à l'objet **XMLHttpRequest** :

Constante	Valeur	Description
UNSENT	**0**	L'objet **xhr** a été créé, mais n'a pas été initialisé (la méthode **open()** n'a pas encore été appelée)
OPENED	**1**	La méthode **open()** a été appelée, mais la requête n'a pas encore été envoyée par la méthode **send()**
HEADERS_RECEIVED	**2**	La méthode **send()** a été appelée et toutes les informations ont été envoyées au serveur
LOADING	**3**	Le serveur traite les informations et a commencé à renvoyer les données → tous les en-têtes des fichiers ont été reçus
DONE	**4**	Toutes les données ont été réceptionnées

L'utilisation de la propriété **readyState** de l'objet **xhr** est nécessaire pour connaître l'état de la requête → l'état qui nous intéresse est le dernier (DONE ou 4) car nous voulons savoir si notre requête est terminée

Nous pouvons donc faire évoluer notre code ci-dessus en

```
xhr.addEventListener('readystatechange', function() {
   if (xhr.readyState == 4)
   {
     // code à exécuter en cas de réception finale de la réponse
   }
});
```

Cependant, cela ne signifie toujours pas que la réponse reçue correspond à ce que nous attendons. Il faut encore tester le code HTML statut de la réponse grâce à la propriété **status** de l'objet **xhr** :

```
xhr.addEventListener('readystatechange', function() {
   if (xhr.readyState == 4 && xhr.status == 200)
   {
     // code à exécuter en cas de réception d'une réponse correcte
   }
});
```

Ceci bien sûr si le code de retour attendu est bien 200 (fichier trouvé). On peut évidemment tester d'autres codes de retour HTML.

Extraction des données de la réponse

Pour accéder au corps de la réponse, la variable **xhr** dispose de plusieurs propriétés :

- **response** : qui fournit la donnée « brute » contenue dans la réponse, sous forme d'un objet Javascript en fonction du type de contenu envoyé par le serveur → par exemple, si le champ **responseType** de l'objet **xhr** a été spécifiée à **'json'**, l'objet obtenu est une instance de la classe **JSON** de Javascript
- **responseText** : fournit la réponse sous la forme d'une chaîne de caractères → valable quel que soit le contenu de la réponse
- **responseXML** → fournit les données extraites de la réponse sous la forme d'un objet de la classe **DOM** de Javascript

Si, par exemple, les données contenues dans la réponse sont au format **JSON**, on peut récupérer ces données ainsi :

```
var data = JSON.parse(xhr.responseText) ;
```

Mais si la propriété **responseType** a été spécifiée à **'json'**, les données à extraire sont automatiquement dans le format adéquat et on peut écrire :

```
var data = xhr.response ;
```

où **data** est un objet **JSON** de Javascript.

Envoyer des données dans le corps de la requête

Imaginons que nous souhaitions envoyer des données dans le corps de la requête, ce qui sera nécessaire par exemple pour notre requête POST de notre API « api/tasks »

Pour cela, nous pouvons utiliser

- la méthode **setRequestHeader()** de l'objet **xhr** pour préciser le « **Content-Type** » de la requête
- la méthode **send()** de l'objet **xhr** pour transmettre un contenu dans le corps de la requête

Par exemple, dans le cas de notre API « **api/tasks** », une requête **POST** pourrait ressembler à ceci :

```
xhr.open("POST","http://192.168.159.131:8081/api/tasks",true);
xhr.responseType = "text";
xhr.setRequestHeader("Content-type","text/plain");

var body = 'Aller faire les courses';
xhr.send(body);
```

On a donc précisé que

- le contenu de la requête est du texte brute
- la réponse attendue est du texte brute

Les requêtes « cross-domain »

Les requêtes **cross-domain** sont des requêtes effectuées depuis un nom de domaine A vers un nom de domaine B. Par exemple, supposons que

- nous ayons obtenu le code **HTML**, **CSS** et **Javascript** à partir du domaine **192.168.159.131:8080**
- le code Javacript reçu réalise une **requête AJAX** vers notre API dont le domaine est **192.168.159.131:8081**

Pour des raisons de sécurité, cela est impossible à cause du principe de la « **same origin policy** » qui est une politique de sécurité importante mise en place par les navigateurs web pour garantir la sécurité des applications web. Cependant cette politique de sécurité est un frein dans le développement des applications web actuelles.

Pour contourner cette sécurité, il faut modifier les **entêtes « CORS »** (« **C**ross-**O**rigin **R**essource **S**haring ») des requêtes reçues de notre API.

Pour faire court, il suffit d'ajouter les lignes suivantes dans la méthode **handle** de notre **API Java** :

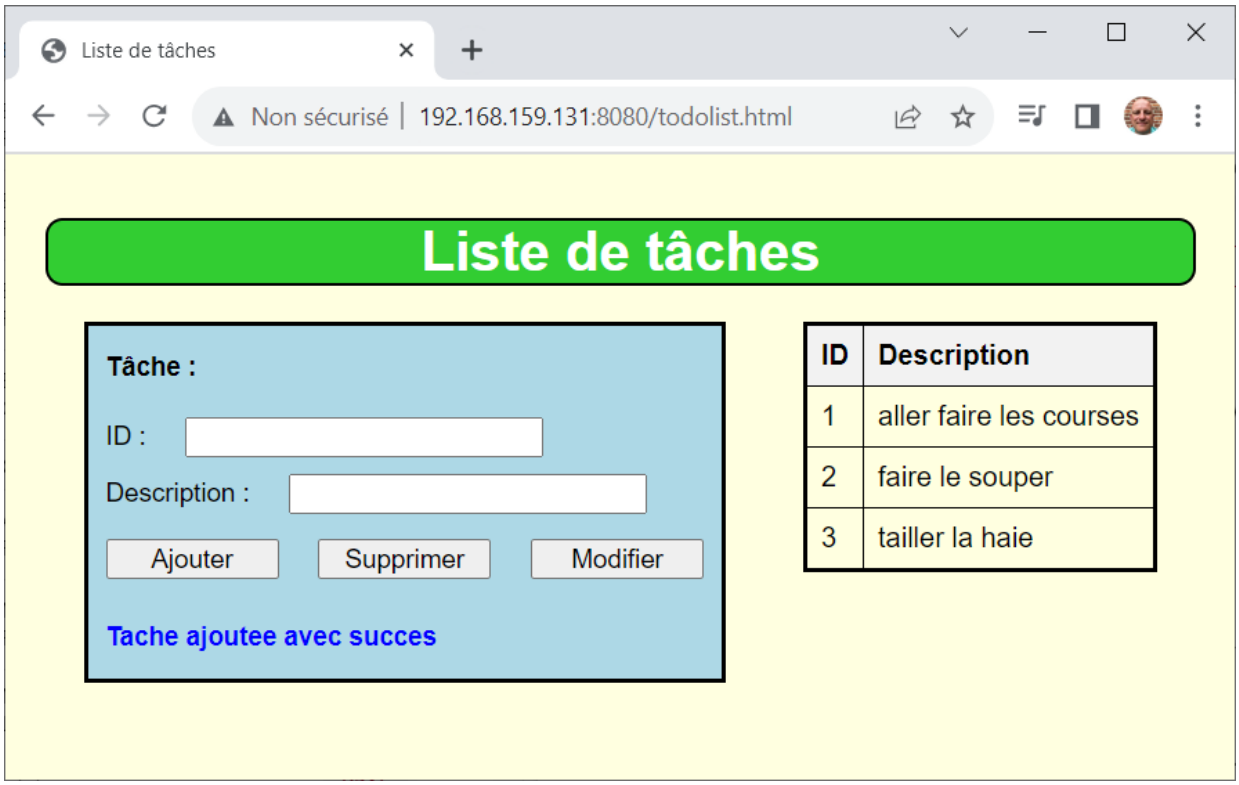
```
// CORS (Cross-Origin Resource Sharing)
exchange.getResponseHeaders().add("Access-Control-Allow-Origin", "*");
exchange.getResponseHeaders().add("Access-Control-Allow-Methods", "GET, POST, DELETE, PUT");
exchange.getResponseHeaders().add("Access-Control-Allow-Headers",   "Content-Type");
```

ce qui donne l'accès à notre API de n'importe quel (*) domaine.

Exemple de frontend HTML/CSS/Javascript pour notre API REST « To Do List » Java

Dans l'exemple qui suit, nous utilisons telle quelle l'API REST « To Do List » décrite plus, mis à part le fait que les entêtes CORS ont été adaptées.

Notre frontend ressemblera à ceci :



où

- la **table** à droite montre le contenu de la liste des tâches
- le bouton « **Ajouter** » permet d'ajouter une tâche à la liste de tâches, tâche dont la description est récupérée dans le champs correspondant
- le bouton « **Supprimer** » permet de supprimer une tâche de la liste de tâches, tâche dont l'ID est récupéré dans le champ correspondant
- le bouton « **Modifier** » permet de modifier une tâche de la liste, tâche dont on récupère l'ID et la nouvelle description dans les champs correspondants
- la **ligne de texte bleue** correspond à la réponse de l'API dans le cas d'une réponse purement textuelle

La page HTML (en particulier la table à droite) est modifiée en cours d'exécution sans rechargement d'aucun autres éléments HTML ou CSS → il s'agit d'une application web **SPA** (« Single Page Application »)

Voici le contenu du fichier **HTML** (fichier **todolist.html**) :

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="utf-8">
		<title>Liste de tâches</title>
		<link href="css/styleTodolist.css" type="text/css"
rel="stylesheet">
	</head>

	<body>
		<h1 class="monTitre">Liste de tâches</h1>

		<div id="conteneur">

			<div id="bloc1">
				<p style="color: black;font-weight: bold;font-family:
Arial;font-size: 15px; padding-left: 10px">Tâche : </p>
				<div>
					<label>ID :</label>
					<input type="number" id="id" name="id"><br>
				</div>

				<div>
					<label>Description :</label>
					<input type="text" id="description"
name="description"><br>
				</div>

				<button id="add">Ajouter</button>
				<button id="remove">Supprimer</button>
				<button id="update">Modifier</button><br>

				<p style="color: blue; font-weight: bold; font-family:
Arial; font-size: 15px; padding-left: 10px" id="monTexte">...statut de la
requête...</p>
			</div>

			<div id="bloc2">
				<table id="maTable">
					<tr>
						<td>ID</td>
						<td>Description</td>
					</tr>
					<tr>
						<td>3</td>
						<td>Aller faire les courses</td>
					</tr>
				</table>
```

```
            </div>

        </div>

        <script src="js/app.js"></script>
    </body>
</html>
```

On remarque que :

- ce fichier HTML utilise un fichier de style **css/styleTodolist.css** mais aussi un script Javascript **js/app.js** qui contient toute la logique de l'application
- un conteneur « global » (**div**) contient deux conteneurs secondaires (**div** et **div**), le premier pour l'interaction (entrées + boutons) avec l'utilisateur et le second pour afficher la table
- plusieurs éléments HTML ont reçu un **id** car ils seront manipulés par le code Javascript

Voici le **fichier de style** (fichier **styleTodolist.css**) :

```
#bloc1 {
    background-color: lightblue;
    color: black;
    font-size: 12px;
    border: 3px solid black;
}

#bloc2 {
    background-color: lightseagreen;
    color: yellow;
}

#conteneur {
    display: flex;
    flex-direction: row;
    justify-content: space-around;
}

button {
    font-size: 15px;
    font-family: Arial;
    width: 100px;
    margin-bottom: 10px;
    margin-top: 10px;
    margin-left: 10px;
    margin-right: 10px;
}

html {
```

```
      background: lightyellow;
      padding: 15px;
}

label,input {
      font-size: 15px;
      font-family: Arial;
      width: 200px;
      margin-bottom: 5px;
      margin-top: 5px;
      margin-left: 10px;
      margin-right: 10px;
}

.monTitre {
      border: 2px solid black;
      background-color: limegreen;
      text-align: center;
      font-family: Arial, sans-serif;
      color: white;
      border-radius: 10px;
}

#maTable {
  border-collapse: collapse;
  border: 3px solid black;
}

#maTable td {
  border: 1px solid black;
  padding: 8px;
  font-family: Arial;
}

#maTable tr:first-child {
  background-color: #f2f2f2;
  font-weight: bold;
  color: black;
  font-family: Arial;
}
```

où plusieurs sélecteurs CSS différents ont été utilisés

Voici à présent le fichier **Javascript** (**app.js**) :

```
(function() {
  console.log("Ceci est une fonction auto-exécutante !");
  miseAJourTable();
})();

document.getElementById('add').addEventListener("click",function(e) {
      var xhr = new XMLHttpRequest();

      xhr.onreadystatechange = function()
      {
            console.log(this);
            if (this.readyState == 4 && this.status == 201)
            {
                  console.log(this.response);
                  miseAJourTable();

                  var monTexte = document.getElementById("monTexte");
       monTexte.innerHTML = this.responseText;
            }
            else if (this.readyState == 4) {
                  alert("Une erreur est survenue...");
            }
      };

      xhr.open("POST","http://192.168.159.131:8081/api/tasks",true);
      xhr.responseType = "text";
      xhr.setRequestHeader("Content-type","text/plain");

      var body = document.getElementById('description').value;
      xhr.send(body);

      document.getElementById('description').value = "";
      document.getElementById('id').value = "";
});

document.getElementById('remove').addEventListener("click",function(e) {
      var xhr = new XMLHttpRequest();

      xhr.onreadystatechange = function()
      {
            console.log(this);
            if (this.readyState == 4 && this.status == 200)
            {
                  console.log(this.response);
                  miseAJourTable();

                  var monTexte = document.getElementById("monTexte");
       monTexte.innerHTML = this.responseText;
```

```
            }
            else if (this.readyState == 4) {
                  alert("Une erreur est survenue...");
            }
      };

      var url = "http://192.168.159.131:8081/api/tasks?id=" +
document.getElementById('id').value;
      xhr.open("DELETE",url,true);
      xhr.responseType = "text";
      xhr.setRequestHeader("Content-type","text/plain");

      xhr.send();

      document.getElementById('description').value = "";
      document.getElementById('id').value = "";
});

document.getElementById('update').addEventListener("click",function(e) {
      var xhr = new XMLHttpRequest();

      xhr.onreadystatechange = function()
      {
            console.log(this);
            if (this.readyState == 4 && this.status == 200)
            {
                  console.log(this.response);
                  miseAJourTable();

                  var monTexte = document.getElementById("monTexte");
       monTexte.innerHTML = this.responseText;
            }
            else if (this.readyState == 4) {
                  alert("Une erreur est survenue...");
            }
      };

      var url = "http://192.168.159.131:8081/api/tasks?id=" +
document.getElementById('id').value;
      xhr.open("PUT",url,true);
      xhr.responseType = "text";
      xhr.setRequestHeader("Content-type","text/plain");
      var body = document.getElementById('description').value;
      xhr.send(body);

      document.getElementById('description').value = "";
      document.getElementById('id').value = "";
});
```

```
function miseAJourTable()
{
	var xhr = new XMLHttpRequest();

	xhr.onreadystatechange = function()
	{
		console.log(this);
		if (this.readyState == 4 && this.status == 200)
		{
			console.log(this.response);

			articles = this.response;
			videTable();
			articles.forEach(function(article) {
				ajouteLigne(article.id,article.task);
			})

		}
		else if (this.readyState == 4) {
			alert("Une erreur est survenue...");
		}
	};

	xhr.open("GET","http://192.168.159.131:8081/api/tasks",true);
	xhr.responseType = "json";
	xhr.send();
}

function ajouteLigne(id,description)
{
	var maTable = document.getElementById("maTable");
	// Créer une nouvelle ligne
	var nouvelleLigne = document.createElement("tr");
	// Créer des cellules
	celluleId = document.createElement("td");
	celluleId.textContent = id;
	celluleDescription = document.createElement("td");
	celluleDescription.textContent = description;
	// Ajouter les cellules à la ligne
	nouvelleLigne.appendChild(celluleId);
	nouvelleLigne.appendChild(celluleDescription);
	// Ajouter la nouvelle ligne au tableau
	maTable.appendChild(nouvelleLigne);
}

function videTable()
{
	var maTable = document.getElementById("maTable");
	while (maTable.rows.length > 1) {
		maTable.deleteRow(-1);   // supprimer dernière ligne
```

```
    }
}
```

où on observe que

- les 4 premières lignes correspondent à une fonction **« auto-exécutante »** qui s'exécute dès que le fichier **app.js** est téléchargé et qui appelle la fonction **MiseAJourTable()** qui remplit la table au démarrage avec ce que contient déjà la liste de tâches de notre API.
- La fonction **MiseAJourTable()** réalise une requête AJAX de type **GET** et demande à obtenir une réponse au format **JSON**. Le code HTTP de retour attendu et donc testé est **200** (comme cela été mis en place dans notre API) → la variable **articles** extraite de la réponse reçue du serveur est donc directement un objet Javascript JSON → cet objet JSON est en fait un tableau d'objets où chaque objet contient les propriétés **id** et **task** → la boucle permet d'itérer tous ces objets et la fonction **ajouteLigne()** permet d'ajouter une ligne à la table d'identifiant **« maTable »**
- Un clic sur le bouton **« Ajouter »** provoque l'envoi d'une requête AJAX de type **POST** → la variable **body** correspond au corps de la requête, c'est-à-dire la tâche (chaîne de caractères → **text/plain** pour le **Content-type**) à ajouter → le code de retour HTTP attendu et testé est ici **201** → dans ce cas, une mise à jour de la table est réalisée par l'appel de la fonction **MiseAJourTable()** → de plus, la réponse à la requête (**responseText**) est affichée dans l'élément HTML identifié par « monTexte » (le texte en bleu)
- Un clic sur le bouton **« Supprimer »** provoque l'envoi d'une requête AJAX de type **DELETE** dans laquelle l'**id** de la tâche à supprimer est passé directement dans l'**URL** de la requête → ici le corps de la requête est vide → le retour et l'affichage de la réponse se fait exactement comme dans le cas précédent
- Un clic sur le bouton **« Modifier »** provoque l'envoi d'une requête AJAX de type **UPDATE** dans laquelle l'**id** de la tâche à supprimer est passé dans l'**URL** et où la nouvelle description est placée dans le corps (**body**) de la requête → le retour et l'affichage de la réponse se fait à nouveau exactement comme dans le cas précédent

Introduction à Javascript

Javascript est un langage de programmation interprété (le plus souvent par le browser – donc au niveau du client) et qui permet rendre une page HTML **interactive** → **HTML + CSS** vont constituer l'interface graphique tandis que **Javascript** va fournir la **logique** cachée derrière

Le code Javascript d'une application web se trouve le plus souvent dans un fichier d'extension **.js** et celui-ci est intégré à la page HTML de la manière suivante (fichier **introJavacript1.html**) :

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="utf-8">
		<title>Introduction à Javascript</title>
	</head>

	<body>
		<h1>Introduction a Javascript</h1>

		<script src="js/app1.js"></script>
	</body>
</html>
```

où on a utilisé la balise **<script>** pour importer le fichier Javascript **app1.js** → cette balise est toujours inséré dans le corps (body) du fichier HTML et à la fin de celui-ci

Variables et console Javascript

Javascript est langage très faiblement typé (du fait que c'est un langage interprété et non compilé). On déclare une variable avec le mot clé « **var** » et on distingue 3 types de variables

- **number** → pour des nombres
- **string** → pour des chaînes de caractères
- **boolean** → pour des booléens

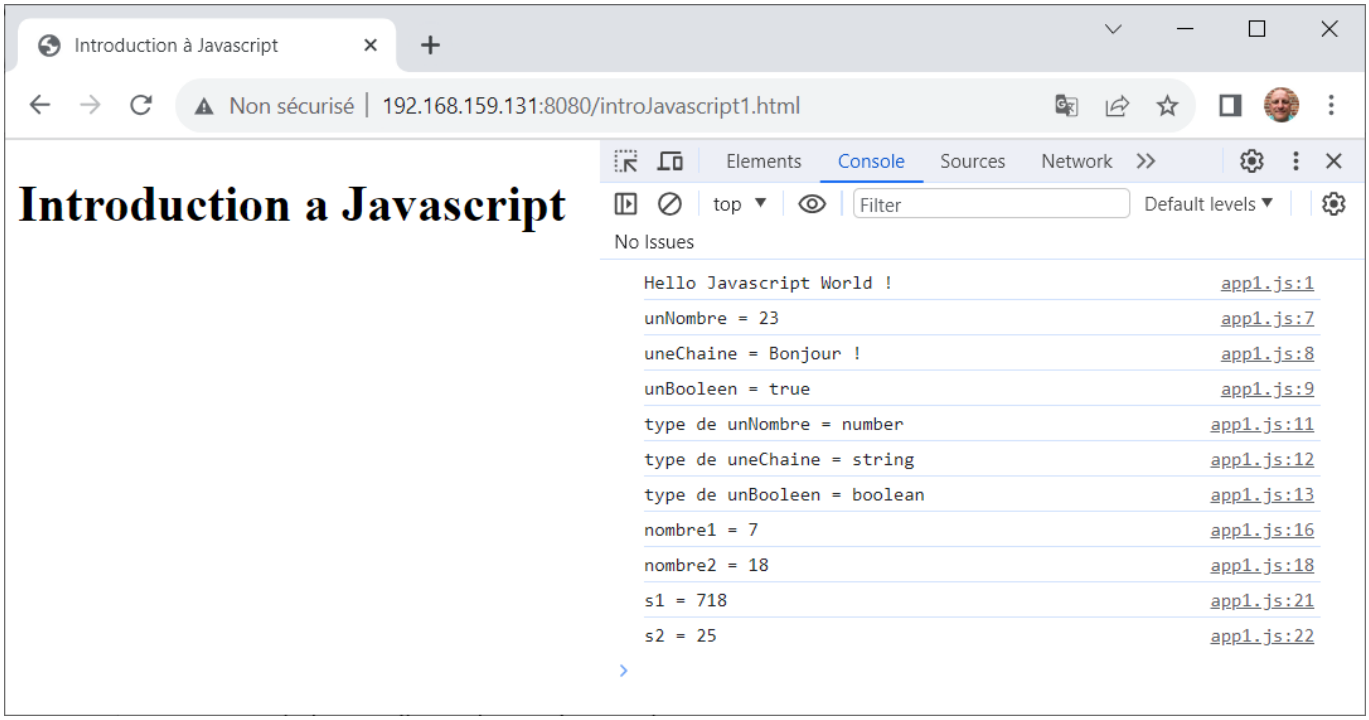
Un premier exemple de fichier Javascript (fichier **app1.js**) est

```
console.log("Hello Javascript World !");

var unNombre = 23;
var uneChaine = 'Bonjour !';
var unBooleen = true;

console.log('unNombre = ' + unNombre);
console.log('uneChaine = ' + uneChaine);
console.log('unBooleen = ' + unBooleen);

console.log('type de unNombre = ' + typeof unNombre);
console.log('type de uneChaine = ' + typeof uneChaine);
console.log('type de unBooleen = ' + typeof unBooleen);

var nb1 = prompt("Entrez un 1er nombre :");
console.log('nombre1 = ' + nb1);
var nb2 = prompt("Entrez un 2eme nombre :");
console.log('nombre2 = ' + nb2);
var s1 = nb1 + nb2 ;
var s2 = Number(nb1) + Number(nb2);
console.log('s1 = ' + s1);
console.log('s2 = ' + s2);
alert('La somme de vos de nombres est egale a ' + s2);
```

où

- **console** est un objet Javascript contenant la fonction **log** qui permet de faire une trace dans la console Javascript du brower → pour accéder à la console Javascript sur Chrome, touche « F12 » puis « console »
- **prompt** est une fonction globale de Javascript permettant de faire apparaître une boîte de dialogue de saisie dans le browser et de retourner ce qui a été tapé sous la forme d'une chaîne de caractères
- **alert** est une autre fonction globale de Javascript permettant de faire apparaître une boîte de dialogue qui permet d'afficher une chaîne de caractères
- **Number(…)** est une fonction convertissant une chaîne de caractères en nombre → sans cela la somme nb1 + nb2 fournit une chaîne de caractères correspondant à la concaténation des 2 chaînes nb1 et nb2

Le résultat de ce script dans Google Chrome fournit



Il est également possible de taper directement du code Javascript dans la console → **>**

Les alternatives et les boucles

La programmation des alternatives et des boucles n'est pas très différente d'un langage comme Java ou C. Quelques exemples sont fournis dans le fichier **app2.js** ci-dessous qui remplace ici app1.js dans le fichier HTML présentés ci-dessus :

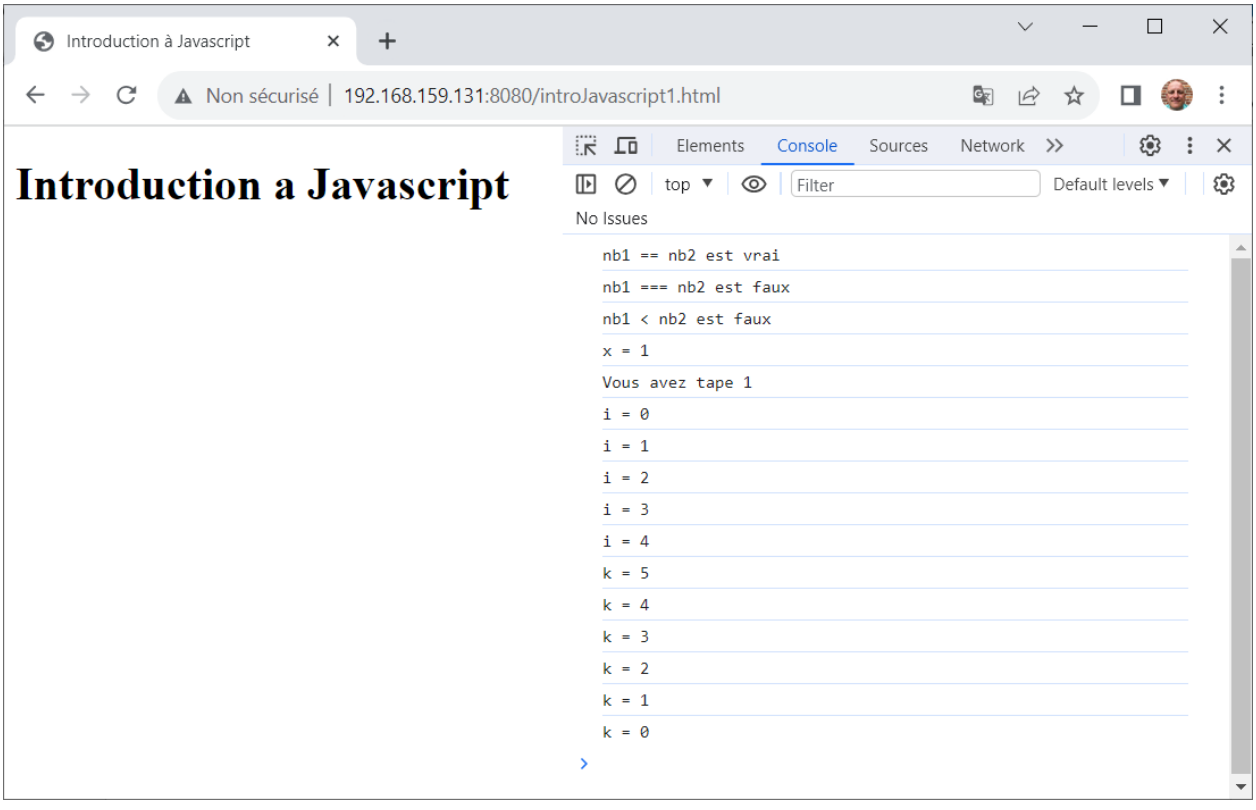
```
var nb1 = 10;
var nb2 = '10';

if (nb1 == nb2) console.log('nb1 == nb2 est vrai');
else console.log('nb1 == nb2 est faux');
if (nb1 === nb2) console.log('nb1 === nb2 est vrai');
else console.log('nb1 === nb2 est faux');
if (nb1 < nb2) console.log('nb1 < nb2 est vrai');
else console.log('nb1 < nb2 est faux');

var onContinue = confirm('On continue ?');
if (onContinue)
{
      var x = prompt('Entrez une valeur :')
      console.log("x = " + x);
      switch(x)
      {
            case '1' : console.log("Vous avez tape 1");
                       break;
            case 'a' : console.log('Vous avez tape a');
```

```
                        break;
                default : console.log('Vous avez tape autre chose 1 et a');
        }
}

var i=0;
while (i<5)
{
        console.log('i = ' + i);
        i++;
}

for(var k=5 ; k>=0 ; k--)
{
        console.log('k = ' + k);
}
```

Rien de bien compliqué… on observe que

- L'opérateur **==** compare le **contenu** sans tenir compte du type de variable → 10 et donc égale à '10'
- L'opérateur **===** compare non seulement le **contenu** mais également le **type** de variable → 10 n'est donc pas égal à '10'
- **confirm** est une fonction globale qui ouvre une boîte de dialogue qui affiche une chaîne de caractère et qui retourne un booléen selon l'action de l'utilisateur

Le résultat de ce script dans Google Chrome fournit



Les tableaux

Les tableaux en Javascript se créent par l'intermédiaire des crochets **[** et **]** → une variable de type « tableau » peut donc être déclarée ainsi

```
var tableau = [];
var tab = [‘Hello’,’C++’,5,’45’] ;
```

Un tableau peut contenir des éléments hétérogènes, comme dans le 2ème exemple. Les éléments sont référencés par un index qui commence à 0.

Un tableau est en réalité un objet qui possède des méthodes membres comme

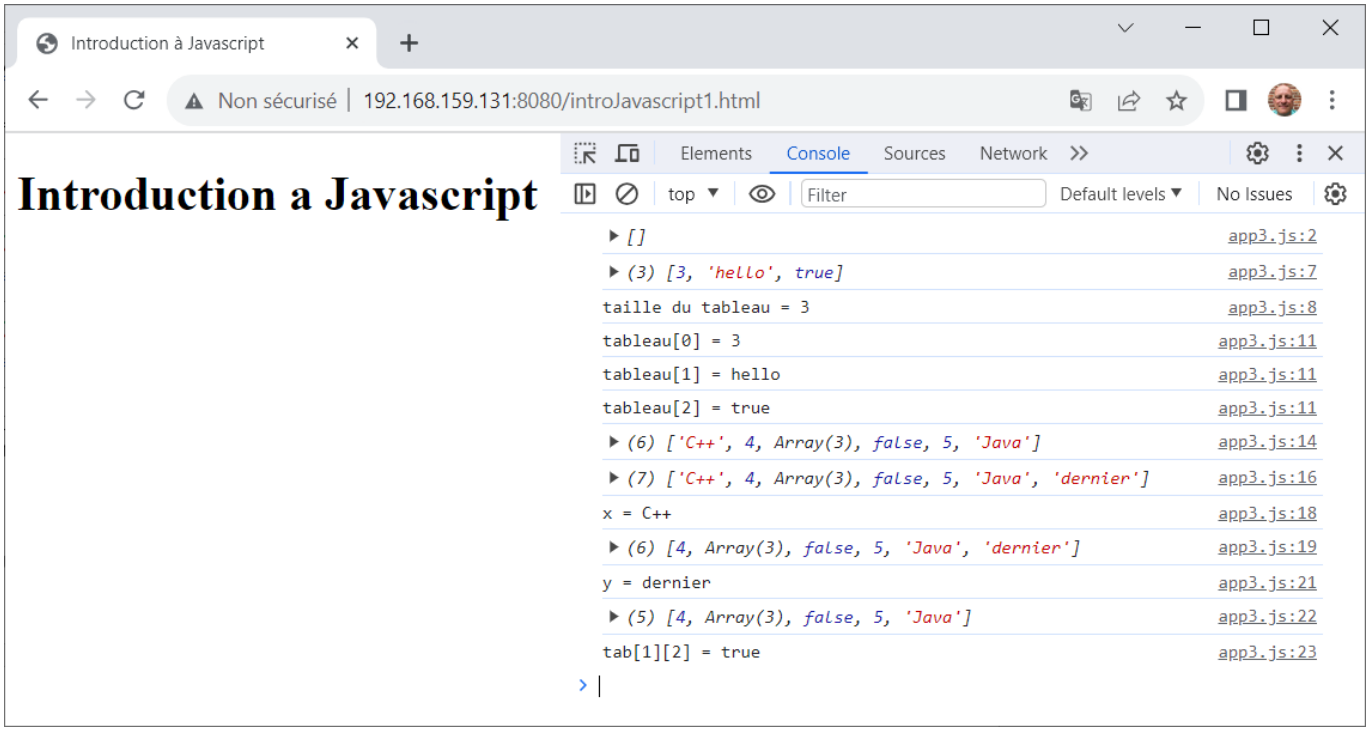
- **push()** → qui permet d'insérer une ou plusieurs valeurs à la fin
- **pop()** → qui permet de retirer la dernière valeur
- **shift()** → qui permet de retirer la première valeur

Quelques exemples sont fournis dans le fichier **app3.js** ci-dessous qui remplace ici app1.js dans le fichier HTML présentés ci-dessus :

```
var tableau = [];
console.log(tableau);

tableau.push(3);
tableau.push('hello');
tableau.push(true);
console.log(tableau);
console.log('taille du tableau = ' + tableau.length);

for(var k=0 ; k<3 ; k++)
  console.log('tableau[' + k + '] = ' + tableau[k]);

var tab = ['C++',4,tableau,false,5,'Java'];
console.log(tab);
tab.push('dernier');
console.log(tab);
var x = tab.shift();
console.log('x = ' + x);
console.log(tab);
var y = tab.pop();
console.log('y = ' + y);
console.log(tab);
console.log('tab[1][2] = ' + tab[1][2]);
```

où on remarque simplement qu'un tableau peut contenir d'autres tableaux

Le résultat de ce script dans Google Chrome fournit



Les fonctions

Pour déclarer/définir une fonction en Javascript, on utilise le mot clé « **function** » et une fonction pourrait donc ressembler à ceci :

```
function maFonction(param1, param2, param3)
{
  // code de la fonction
  return … // si la fonction retourne quelque chose
}
```

où « **maFonction** » est le nom de la fonction, le nombre de paramètres peut être quelconque et le type de chaque paramètre n'a pas besoin d'être précisé.

Pour exécuter cette fonction, il faut encore l'appeler :

```
maFonction(3,'hello',2) ;
```

On peut également définir une fonction sans lui donner de nom mais alors il faut la stocker dans une variable :

```
var monAutreFonction = function(param1, param2) {
  // code de la fonction
} ;
```

Et celle-ci peut s'appeler comme auparavant, c'est-à-dire

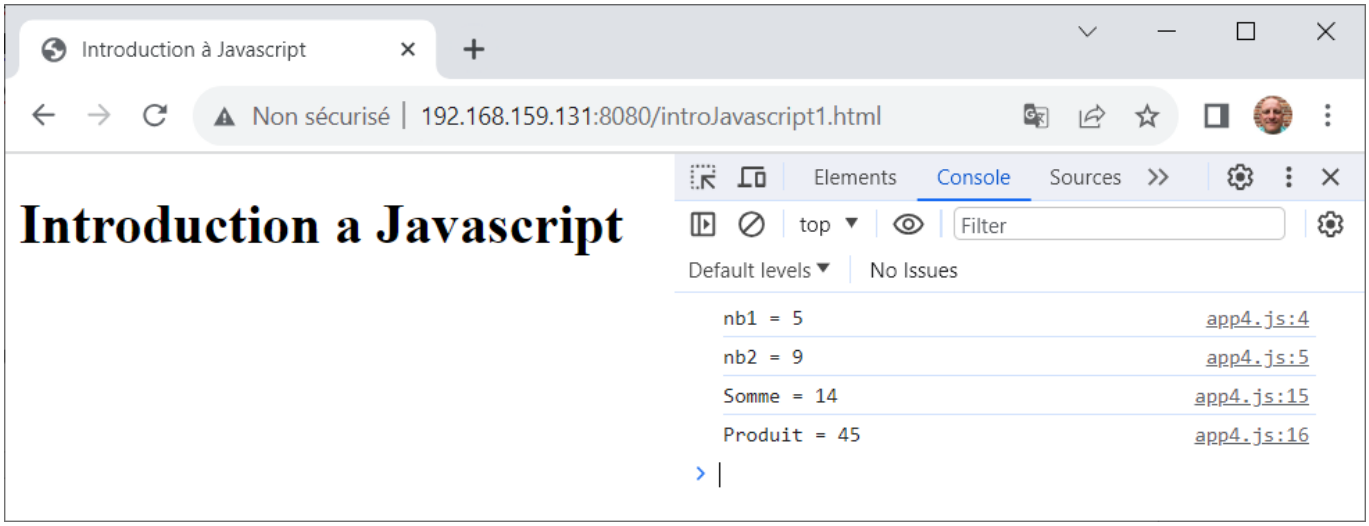
```
monAutreFonction('Java',4) ;
```

Quelques exemples sont fournis dans le fichier **app4.js** ci-dessous qui remplace ici app1.js dans le fichier HTML présentés ci-dessus :

```
nb1 = prompt('Entrez un 1er nombre :');
nb2 = prompt('Entrez un 2eme nombre : ');

console.log('nb1 = ' + nb1);
console.log('nb2 = ' + nb2);

var s = somme(nb1,nb2);

var produit = function(x,y) {
  return Number(x) * Number(y);
};

var p = produit(nb1,nb2);

console.log('Somme = ' + s);
console.log('Produit = ' + p);
alert('Leur somme est egale a : ' + s);
alert('Leur produit est egal a : ' + p);

function somme(x,y)
{
  return Number(x) + Number(y);
}
```

où la fonction **produit**, étant stockée dans une variable a dû être définie **avant** d'être appelée. Par contre la fonction **somme** est définie après et on remarque qu'il n'est pas nécessaire de la déclarer avant.

Le résultat de ce script dans Google Chrome fournit



Les objets

Les objets en Javascript sont créés à l'aide des accolades { et } → un objet possède des « **propriétés** » ou « **attributs** » qui correspondent aux variables membres du C++ ou du Java.

Le plus simple est d'exposer un exemple. Voici le fichier **app5.js** qui remplace app1.js dans la page HTML montrée ci-dessus :

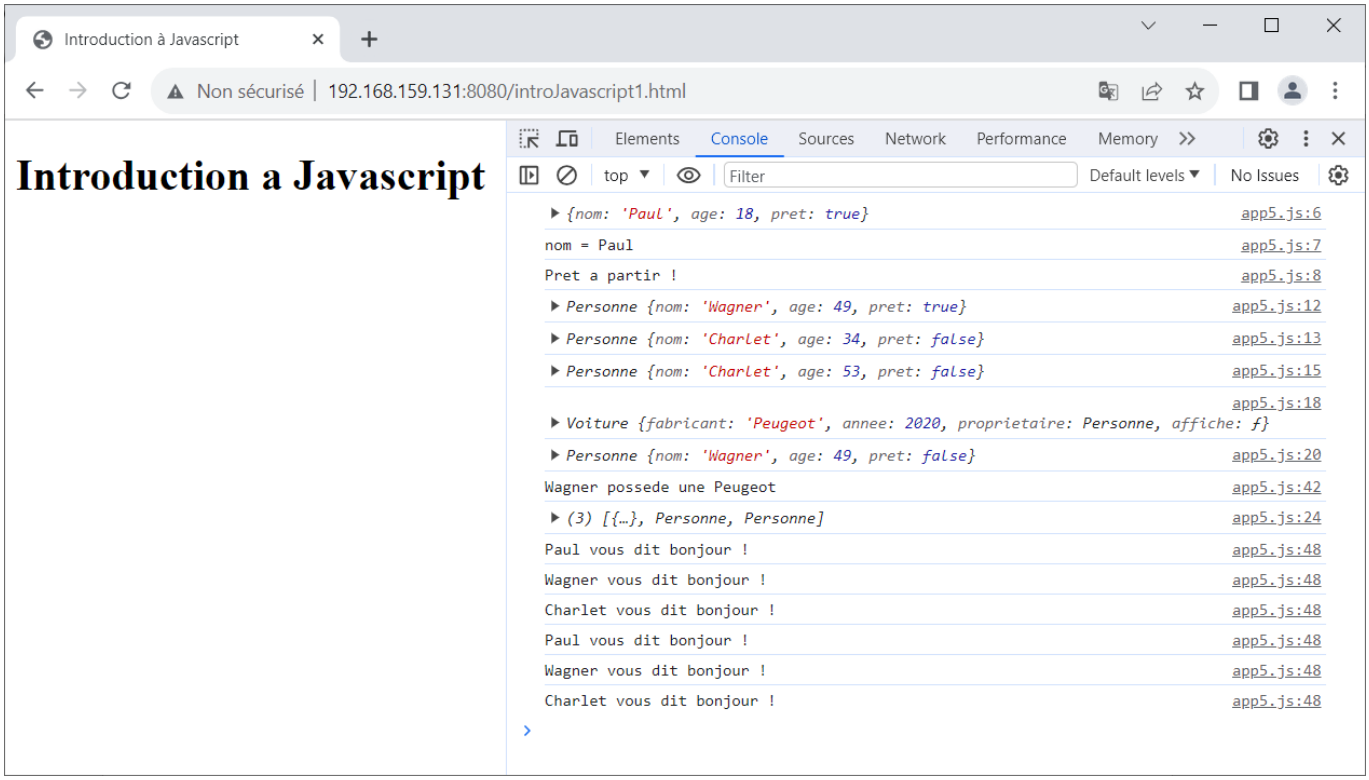
```
var p1 = {
      nom: "Paul",
      age: 18,
      pret: true
};
console.log(p1);
console.log('nom = ' + p1.nom);
if (p1.pret) console.log('Pret a partir !');

var p2 = new Personne("Wagner",49,true);
var p3 = new Personne("Charlet",34,false);
console.log(p2);
console.log(p3);
p3.age = 53;
console.log(p3);

var v = new Voiture("Peugeot",2020,p2);
console.log(v);
v.proprietaire.pret = false;
console.log(p2);
v.affiche();
```

```
var tab = [p1, p2, p3];
console.log(tab);
for(var i=0 ; i<3 ; i++) { DitBonjour(tab[i]); }
tab.forEach(function(p) { DitBonjour(p); });

function Personne(nom,age, pret)
{
	this.nom = nom;
	this.age = age;
	this.pret = pret;
}

function Voiture(fab,an,prop)
{
	this.fabricant = fab;
	this.annee = an;
	this.proprietaire = prop;

	this.affiche = function() {
		console.log(this.proprietaire.nom + ' possede une ' +
this.fabricant);
	}
}

function DitBonjour(p)
{
	console.log(p.nom + ' vous dit bonjour !');
}
```

où on observe que

- **p1** est un objet Javascript « **créé à la volée** » → aucune classe n'a été nécessaire pour créer cet objet → cet objet présente 3 propriétés : **nom**, **age** et **pret** dont le type a été précisé lors de l'affectation de « Paul », 18 et true
- La classe **Personne** a été créée → pour cela on a créé une fonction Javascript qui fait office de constructeur → **p2** et **p3** sont des instances de la classe **Personne**
- Un objet peut contenir d'autres objets → la classe **Voiture** possède une propriété **proprietaire** qui est une instance de la classe **Personne**
- Une classe peut contenir des **méthodes membres** → il suffit de définir une variable et de lui affecter une fonction → ici la fonction **affiche** de la classe **Voiture**
- On peut créer des tableaux d'objets → ici **tab** est un tableau contenant 3 objets → on observe 2 méthodes différentes pour parcourir ce tableau → la boucle for classique et la méthode **forEach** du tableau qui reçoit en paramètre une fonction qui sera appliquée à chaque objet du tableau

L'exécution de ce script dans Google Chrome fournit



où on observe essentiellement que l'objet **v** (de type Voiture) est construit en recevant **p2** en paramètre qui devient donc son propriétaire. En cours d'exécution le propriétaire de **v** a été modifié (pret est passé à false) → on observe (ligne 20) que **p2** a également été modifié de la même manière !! → les objets passés en paramètre à une fonction sont passés par **référence** !

Récupérer et manipuler des éléments HTML et CSS

Le code Javascript est capable d'interagir avec la page HTML dans laquelle il se trouve → Plus précisément, il va interagir avec le **DOM** (**D**ocument **O**bject **M**odel) de la page HTML → celui-ci est représenté par l'objet Javascript global appelé **document**.

L'objet **document** possède les méthodes suivantes :

- **getElementById(…)** → reçoit l'**id** d'un élément HTML et le retourne sous la forme d'un objet que l'on pourra manipuler pour en récupérer/modifier le contenu/style
- **getElementsByTagName(…)** → reçoit le nom d'un **tag** (par exemple 'p' ou 'h1') d'un (ou plusieurs) éléments HTML et retourne un tableau d'objets ayant tous le tag passé en paramètre

- **getElementsByName(…)** → reçoit le **nom** d'un (ou plusieurs) élément HTML et retourne un tableau d'objets ayant tous le nom passé en paramètre

Les éléments HTML récupérés sont tous des objets ayant des propriétés (« variables membres ») que l'on peut récupérer/modifier. Par exemple,

- **innerHTML** → dans le cas par exemple d'une base h1 ou p → permet de récupérer ce qui se trouve entre la balise ouvrante et la balise fermante, généralement sous forme de texte
- **value** → dans le cas d'une balise input, ce qui se trouve dans la balise → on peut récupérer/modifier l'attribut « value » de la balise

Considérons le fichier HTML (fichier **introJavascript2.html**) suivant :

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="utf-8">
		<title>Introduction à Javascript</title>
	</head>

	<body>
		<h1 id="titre">Introduction a Javascript</h1>

		<p id="para1">Ceci est le 1er paragraphe...</p>

		<img id="image" src="images/petitchat.jpg" width=200>

		<p id="para2">Ceci est le 2eme paragraphe...</p>

		<input name="nom" type="text" value="votre nom">
		<input name="age" type="number" value=30>

		<script src="js/app6.js"></script>
	</body>
</html>
```

où

- des **identifiants** uniques ont été donnés à différents éléments HTML
- en plus de ces identifiants, certains de ces éléments HTML ont des **attributs** comme **src** et **width** pour l'image et **value** pour les balises <input>

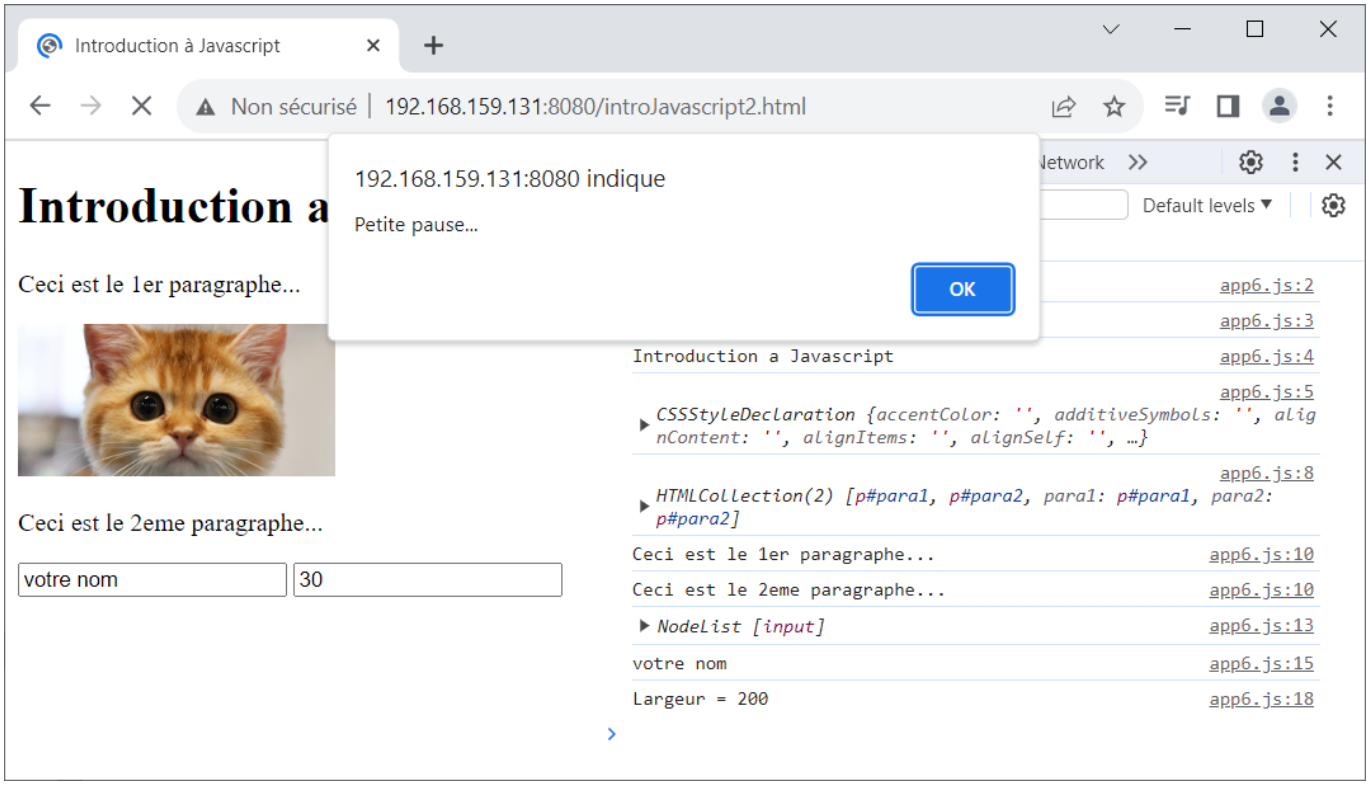
Voici le fichier Javascript (fichier **app6.js**) utilisé par cette page HTML :

```
var elem = document.getElementById('titre');
console.log(elem);
console.log(typeof elem);
console.log(elem.innerHTML);
console.log(elem.style);

var elems1 = document.getElementsByTagName('p');
console.log(elems1);
for(var i=0 ; i<elems1.length ; i++)
      console.log(elems1[i].innerHTML);

var elems2 = document.getElementsByName('nom');
console.log(elems2);
for(var i=0 ; i<elems2.length ; i++)
      console.log(elems2[i].value);

var imw = document.getElementById('image').getAttribute('width');
console.log('Largeur = ' + imw);

alert('Petite pause...');

elem.innerHTML = 'Nouveau Titre !';
elem.style.backgroundColor = "red";
elem.style.color = "yellow";
console.log(elem.style);
elems2[0].value = 'your name';
document.getElementById('image').setAttribute('width',300);
```

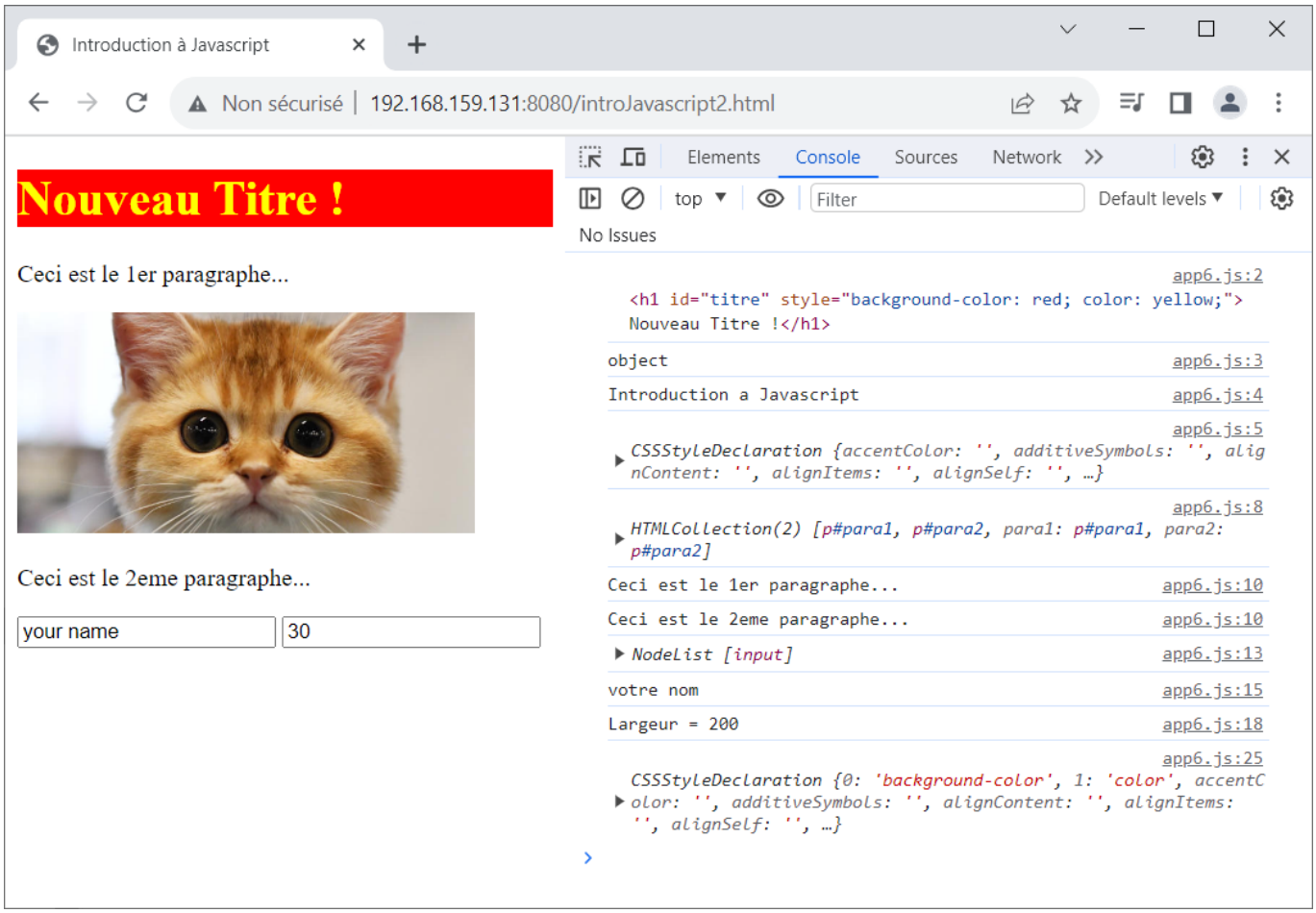
où on observe que

- **elem** est un objet Javascript qui possède les propriétés **innerHTML** (ce qui se trouve entre les balises <h1> et </h1> → « Introduction a Javascript ») et **style** qui contient le style CSS de l'élément
- **elems1** est un tableau d'objets Javascript où chacun des objets représente une balise **<p>** de la page HTML → **elems1** reprend toutes les balises <p> présentes dans la page HTML
- **elems2** est un tableau d'objets Javascript contenant tous les éléments HTML dont le nom (attribut **name**) est égal à **'nom'** → ici il n'y en a qu'un : la balise <input> permettant de saisir un texte
- **imw** contient l'attribut **width** de l'objet HTML dont l'**id** est **'image'** → il s'agit ici de la largeur de l'image
- Après affichage de la boîte de dialogue (**alert**), certains attributs sont modifiés : le contenu et le style CSS du titre, la valeur de la balise <input> correspondant au 'nom', ainsi que la largeur de l'image

L'exécution (jusque **alert**) de ce script dans Google Chrome fournit



Après clic sur le bouton « ok », cela donne



Les événements

Tout comme en Java, il est possible de gérer les événements comme les clics sur les boutons, les mouvements de la souris, etc…

Pour cela, il suffit d'ajouter un **« listener »** à l'élément susceptible de lancer un tel événement → un listener est simplement ici une fonction que l'on associe à un type d'événement pour un élément HTML. Pour cela, tout objet Javascript représentant un élément HTML dispose de la méthode

- **addEventListener(<événement> , <fonction>)**

Le plus simple est de se baser un exemple. Considérons le fichier HTML (**introJavascript3.html**) suivant

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="utf-8">
		<title>Introduction à Javascript</title>
	</head>

	<body>
		<h1 id="titre">Introduction a Javascript</h1>

		<label>Nom :</label>
		<input id="nom" type="text" value="votre nom"> <br><br>
		<label>Age :</label>
		<input id="age" type="number" value=30> <br><br>

		<button id="bouton">Cliquez ici</button>

		<p id="para">Ceci est un paragraphe...</p>

		<script src="js/app7.js"></script>
	</body>
</html>
```

où

- 2 balises <input> vont permettre d'encoder un nom et un age → ces 2 balises ont des identifiants qui permettront au code Javascript de récupérer les « **value** » de ces balises
- Un bouton est identifié → cela va permettre au code Javascript de le récupérer et de lui attribuer un **« listener »**

- Un paragraphe est identifié → le code Javascript pourra le récupérer et le modifier en fonction des données entrées par l'utilisateur

Voici le code Javascript (fichier **app7.js**) utilisé par cette page HTML :

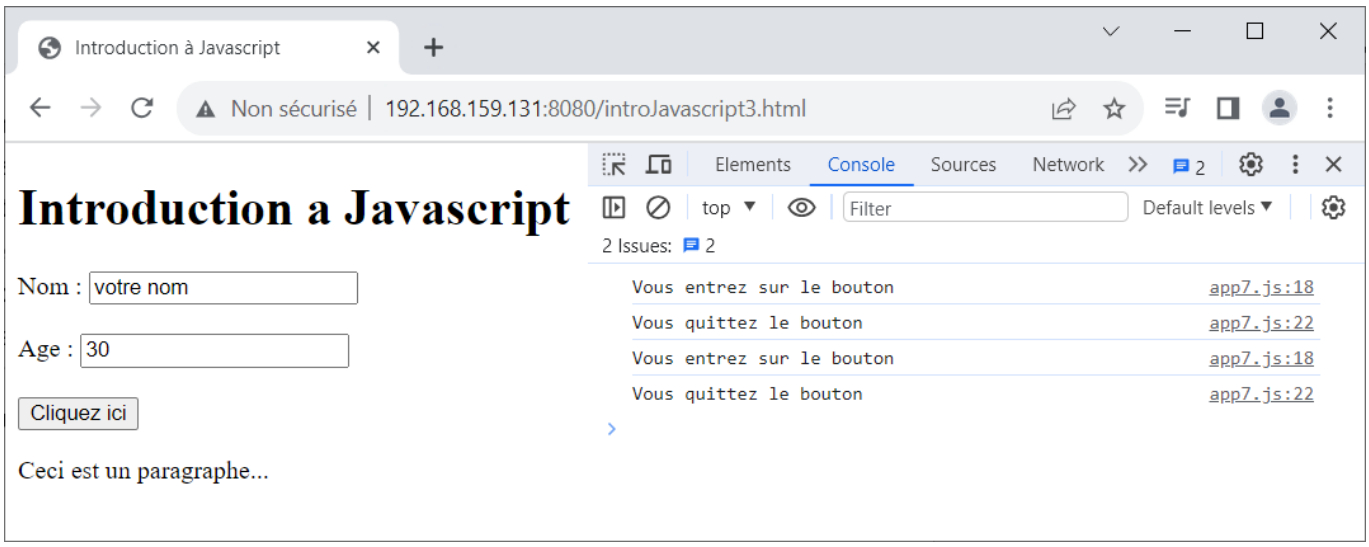
```
document.getElementById('bouton').addEventListener('click', function(e){
      var nom = document.getElementById('nom').value;
      var age = document.getElementById('age').value;
      var para = document.getElementById('para');
      para.innerHTML = 'Bonjour ' + nom + ' ! Vous avez ' + age + ' ans.';
      para.style.color = "red";
      para.style.backgroundColor = "lightgreen";
      para.style.fontFamily = "Arial";
      console.log(e);
});

document.getElementById('bouton').addEventListener('contextmenu',
function(e){
      e.preventDefault();
      alert('Clic droit sur le bouton !');
});

document.getElementById('bouton').addEventListener('mouseenter', function(e){
      console.log('Vous entrez sur le bouton');
});

document.getElementById('bouton').addEventListener('mouseleave', function(e){
      console.log('Vous quittez le bouton');
});
```

où

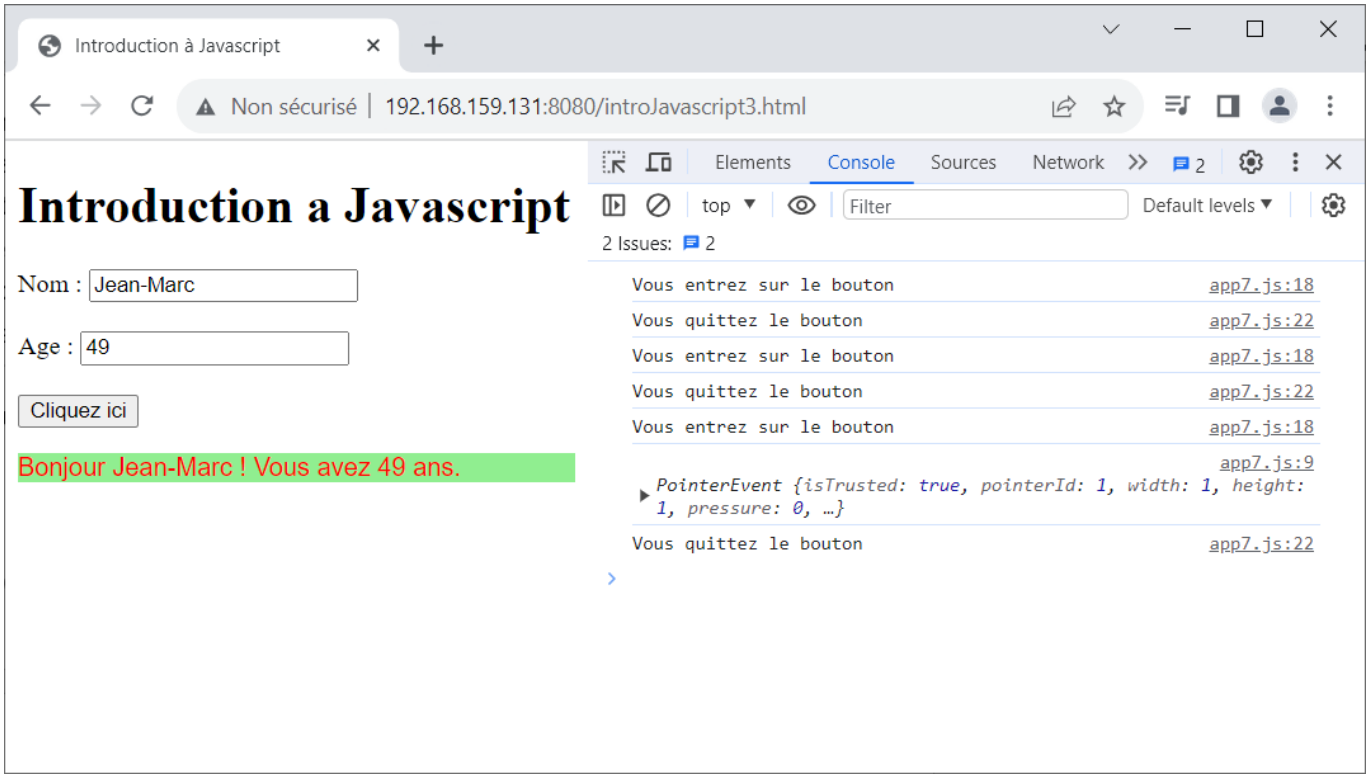
- **4 listeners** ont été associés au bouton : clic gauche (**'click'**), clic droit (**'contextemenu'**), entrée de la souris sur le bouton (**'mouseenter'**), sortie de la souris du bouton (**'mouseleave'**)
- Un clic gauche sur le bouton récupère le nom et l'age de l'utilisateur, crée une phrase lui disant bonjour et place le texte construit dans le paragraphe en lui donnant un style CSS
- **e** est un objet de type « événement » → celui-ci contient toute l'information sur l'événement qui a eu lieu → ce qui peut être affiché dans la console du navigateur
- un clic droit sur le bouton fait apparaître une boîte de dialogue affichant un petit message → l'appel à **e.preventDefault()** retire le comportement par défaut de cet événement qui consiste à afficher le menu contextuel
- entrer et sortir du bouton avec la souris affiche simplement un petit message dans la console

Il existe énormément d'autres événements que l'on pourrait gérer. Le lecteur est invité à se rendre sur la page https://www.w3schools.com/jsref/dom_obj_event.asp pour plus d'informations.

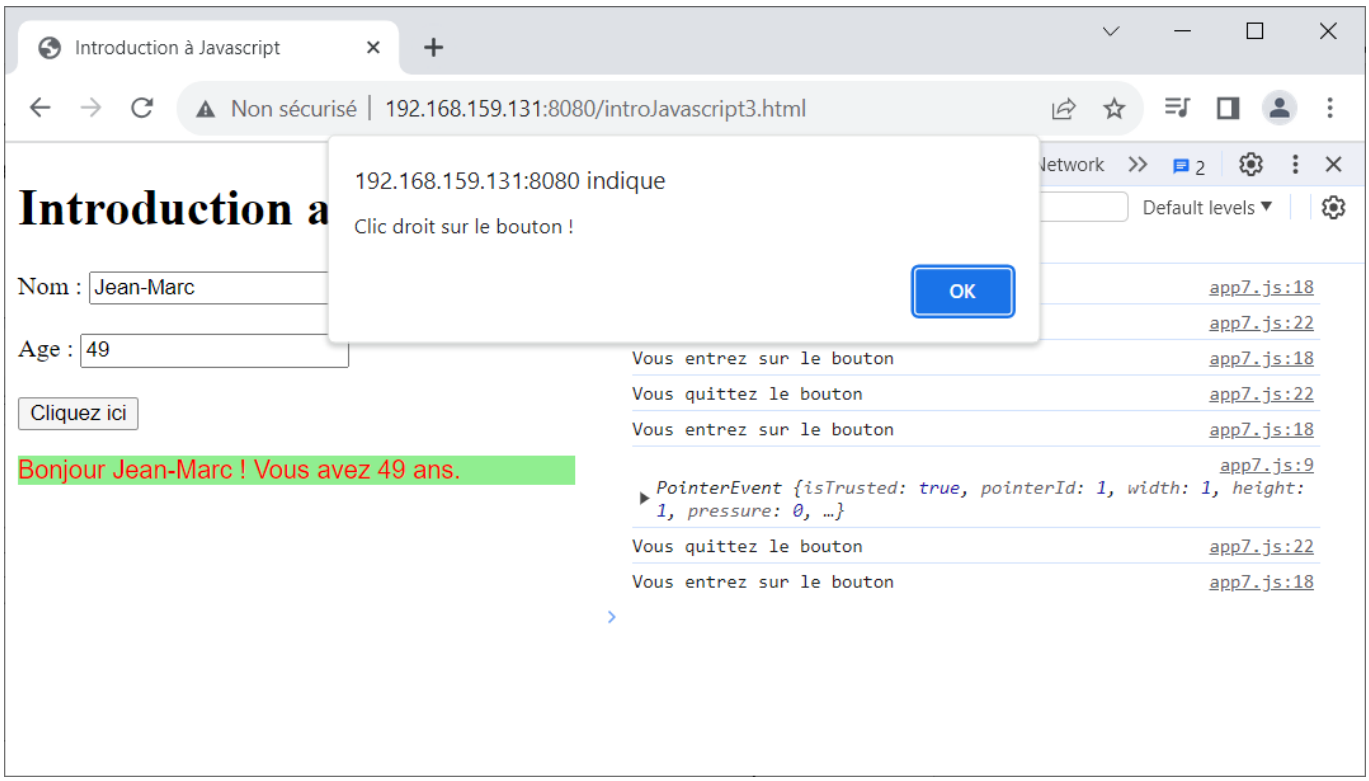
L'exécution du script ci-dessus dans la page HTML fournit



où on a déplacé la souris sur le bouton puis on l'a sortie, et cela deux fois. Après avoir encodé **« Jean-Marc »** et **49** dans les 2 champs, et cliqué (bouton de gauche) sur le bouton, cela fournit



où on voit que l'objet « événement » est de type **PointerEvent**. Finalement, un clic droit sur le bouton fournit


Introduction à Javascript
Non sécurisé | 192.168.159.131:8080/introJavascript3.html
192.168.159.131:8080 indique
Clic droit sur le bouton !
OK
Introduction a
Nom : Jean-Marc
Age : 49
Cliquez ici
Bonjour Jean-Marc ! Vous avez 49 ans.
Default levels
app7.js:18
app7.js:22
Vous entrez sur le bouton
app7.js:18
Vous quittez le bouton
app7.js:22
Vous entrez sur le bouton
app7.js:18
app7.js:9
PointerEvent {isTrusted: true, pointerId: 1, width: 1, height: 1, pressure: 0, …}
Vous quittez le bouton
app7.js:22
Vous entrez sur le bouton
app7.js:18

Quelques exemples d'architecture d'API
(dont certaines directement basée sur le protocole HTTP)