Ecole Centrale de Lyon

UE INF tc3

Projet d'Application Web

Applications web, protocole
et mise en œuvre côté serveur

René Chalon
Rene.Chalon@ec-lyon.fr

Daniel Muller
Daniel.Muller@ec-lyon.fr

3. Applications web, protocole et mise en œuvre côté serveur

Plan de la séance

3.1 Uniform Resource Locators

3.1.1 Qu'est-ce qu'une URL ?

Le fonctionnement du Web repose sur trois mécanismes fondamentaux :

- **les URLs** : un système de nommage universel qui permet de localiser les *ressources* sur le réseau,
- **HTTP** : le protocole qui permet l'accès aux *ressources*, et assure le transfert de l'information depuis son lieu de stockage *(serveur)* vers le client,
- **HTML** : qui permet l'affichage sous forme d'hypertexte et la navigation d'une *ressource* à l'autre.

Une *ressource* est identifiée par un nom unique *(URL)* :

```
http://www.w3.org/TR/xml/
```

L'URL identifie la ressource, mais permet également de savoir comment se la procurer : *(protocole, nom du serveur, chemin d'accès, paramètres…)*

Une URL identifie une ressource en s'appuyant sur une méthode qui en permet l'accès.

[RFC 1738 #2. General URL Syntax]

3.1.2 Format d'une URL

Exemple d'URL :

```
http://www.ietf.org/rfc/rfc2396.txt
```

Cette URL comporte trois parties distinctes :

- le schéma *(scheme)* correspond en général à la méthode d'accès à la ressource :

  `http:`//www.ietf.org/rfc/rfc2396.txt

- l'autorité *(authority)* donne le nom du serveur ou son adresse IP :

  http://`www.ietf.org`/rfc/rfc2396.txt
  http://`104.20.0.85`/rfc/rfc2396.txt

  N.B. Une URL comportant une adresse IP est techniquement correcte, mais certains serveurs ont besoin du nom pour délivrer la ressource demandée...

- le chemin *(path)* d'accès à la ressource sur le serveur :

  http://www.ietf.org`/rfc/rfc2396.txt`

Autres exemples :

```
ftp://ftp.is.co.za/rfc/rfc1808.txt
mailto:Daniel.Muller@ec-lyon.fr
news:comp.infosystems.www.servers.unix
telnet://tic01.tic.ec-lyon.fr
gopher://gopher.hdh.org/00/the%20manifesto.txt
http://localhost/docs/display?task_id=action4
```

Ces exemples permettent d'inférer la syntaxe d'une URL :

```
<scheme>:<scheme-specific-part>
```

et de généraliser une syntaxe plus spécifique que l'on voit se dégager pour certains des schémas *(generic URI)* :

```
<scheme>://<authority><path>?<query>
```

[RFC 2396 #3. URI Syntactic Components]

Query String

L'URL du dernier exemple fait intervenir une chaîne de requête :

```
http://localhost/docs/display?task_id=action4
```

Les chaînes de requête sont couramment utilisées pour passer des paramètres lors d'un accès à une ressource dynamique *(application Web)*.

Fragment

Une URL peut être complétée par un identifiant de fragment :

```
http://www.cop.en/pilac/action14#tab3
```

Les identifiants de fragment permettent traditionnellement de désigner un emplacement particulier au sein d'une ressource *(cf. navigation intra-document html)* statique.

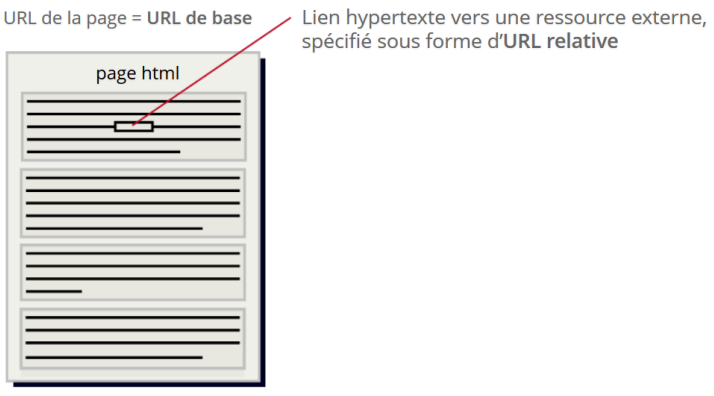
Dans le cas d'une application mono-page ils sont utilisés pour désigner l'interface actuellement présenté à l'utilisateur *(onglet courant par exemple)*.

N.B. Un navigateur ne recharge pas naturellement la page si l'on modifie l'identifiant de fragment à la main dans la barre d'adresse...

3.1.3 URLs relatives

Les URL vues jusqu'à présent sont des URL absolues. Une URL peut également être spécifiée sous forme relative *(par rapport à une **URL de base**)*.

Exemple de contexte d'utilisation :



Voici un lien hypertexte mentionnant une **URL relative** vers une ressourcve externe, tel qu'il peut apparaitre dans un document html :

```
<a href="page2.html">page2</a>
```

Une URL relative omet certains éléments *(schema, authorité, chemin…)* qui sont déduits de l'URL de base.

Avec par exemple l'URL de base :

```
http://www.srv.net/path/page1.html
```

l'URL relative ci-dessus correspond à la ressource :

```
http://www.srv.net/path/page2.html
```

[RFC 3986 #4.2 Relative references]

Une URL relative ne comporte pas de schéma *(scheme - méthode d'accès)* ni la plupart du temps d'autorité *(nom ou adresse du serveur)* et identifie en général une ressource accessible via le même serveur que la ressource de base.

En supposant par exemple que l'URL de base soit :

```
http://tic.ec-lyon.fr/cours/uri/slide4.xml
```

voici quelques URL relatives, avec l'URL absolue correspondante :

URL relative	URL absolue correspondante
`slide3.xml`	`http://tic.ec-lyon.fr/cours/uri/slide3.xml`
`../toc.xml`	`http://tic.ec-lyon.fr/cours/toc.xml`
`../td/td1.pdf`	`http://tic.ec-lyon.fr/cours/td/td1.pdf`
`/index.html`	`http://tic.ec-lyon.fr/index.html`
`?q=33`	`http://tic.ec-lyon.fr/cours/uri/slide4.xml?q=33`
`#par3`	`http://tic.ec-lyon.fr/cours/uri/slide4.xml#par3`

Rappel : Sauf en présence de code Javascript spécifique, un navigateur ne recharge pas la page si l'on actionne un lien qui modifie uniquement l'identifiant de fragment...

Les URLs relatives peuvent être classées en trois catégories suivant la nature de l'information omise *(et donc reprise depuis l'URL de base)* :

Le chemin réseau *(netpath)*

Reprend le même protocole d'accès que pour la ressource de l'URL de base.

N.B. Bien que parfaitement supporté par les navigateurs le chemin réseau n'a été que peu employé jusqu'à récemment, où il est utilisé pour charger les images et autres *assets* d'une page avec le même protocole *(http ou https)* que la page elle-même.

```
<script src="//code.jquery.com/jquery.min.js"></script>
<link rel="stylesheet" type="text/css" href="//w3.org/code.css">
```

Le chemin absolu

Spécifie le chemin d'une ressource qui partage avec l'URL de base le protocole et le serveur. Le chemin absolu possède un caractère "/" initial, et s'applique à partir de la racine du serveur.

```
<a href="/index.html">page d'accueil</a>
<a href="/tools/applweb.php">appli</a>
```

Le chemin relatif

Mêmes protocole et serveur, à partir du « répertoire » de la ressource identifiée par l'URL de base *(pas de caractère "/" initial)*.

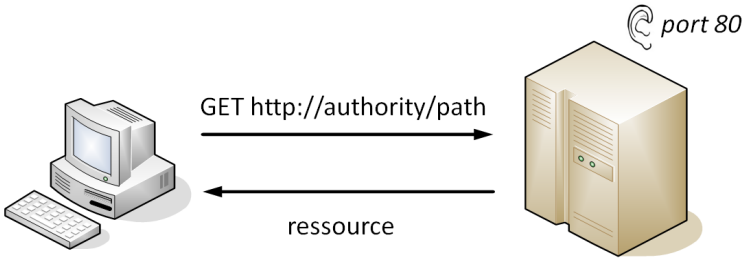
```
<a href="slide1.xml">transparent 1</a>
<a href="../toc.html">Table des matières</a>
```

3.2 Le protocole HTTP

Le protocole HTTP a déjà été rapidement présenté :

- On a vu la forme d'une **requête** et celle d'une **réponse**.
- La réponse comporte trois parties : la ligne de **statut**, les **entêtes**, et le **corps**.
- Le protocole HTTP prévoit différentes **méthodes** *(ou verbs)* pour accomplir certaines opérations sur la ressource.
- A ce propos, ont été définies les notions de méthode **sûre** et de méthode **idempotente**.
- Finalement, l'architecture **REST** qui prévoit de manipuler les ressources via des requêtes utilisant les verbes HTTP a été évoquée.

3.2.1 HTTP/0.9 - Principe et limites



- Le client ouvre une connexion vers le serveur *(port 80)*.
- Le client envoie au serveur la chaîne « GET url_de_la_ressource »

```
GET http://www.ec-lyon.fr/index.html
```

- Le serveur transmet la valeur de la ressource au client, via la connexion ouverte.
- Le serveur coupe la connexion.

[Spécifications HTTP/0.9]

Avantages

- Très simple, et donc facile à mettre en œuvre *(il est facile de développer un logiciel client ou serveur)*.
- Indépendant de la nature de la ressource *(texte, image, …)*
- Indépendant de la couche transport *(TCP / IP)*

Inconvénients

Simple, mais *trop* simple :

- Impossible d'avoir des requêtes avec un contenu *(corps).*
- Aucun moyen de gérer un cache.
- Quel est la statut de la réponse ? *(ok, erreur, …)*
- Connexions éphémères *(partiellement résolu par HTTP/1.1)*

[HTTP de 1991 à 1998]

3.2.2 Requête et réponse HTTP/1.0

Requête

```
GET /welcome.html HTTP/1.0
Host: localhost:8080
Referer: http://localhost:8080/service_tester.html
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:42.0) Firefox/42.0
```

La requête envoyée par le client au serveur comporte :

- une première ligne obligatoire mentionnant le mot-clé `GET` suivi par l'URL de la ressource demandée et le numéro de version HTTP de l'échange,
- une série de directives informationnelles constituant les entêtes à l'intention du serveur, comme ici le nom du serveur cible, l'URL de la ressource à l'origine de la requête *(page courante)* et l'identification du client,
- une ligne vide pour indiquer la fin de la requête.

Noter au passage que l'URL mentionnée dans cette requête est relative, l'URL de base étant rappelée par la directive `Referer`.

[Testeur de requêtes]
(à utiliser par exemple avec l'un des serveurs fournis ou développés dans le cadre du TD3)

Réponse

```
HTTP/1.0 200 Ok
Content-Length: 351
Content-Type: text/html
Date: Wed, 11 Nov 2015 19:47:36 GMT
Last-Modified: Mon, 28 Sep 2015 07:09:07 GMT
Server: Q3-5.py/0.1 Python/3.4.2

 <html>
 ...
 </html>
```

La réponse renvoyée par le serveur comporte :

- une ligne de statut avec la version HTTP du serveur, un code numérique et une description textuelle du statut de l'échange,
- des directives formant les entêtes à l'intention du client, comme ici le nombre de caractères et le type de la ressource, la date et l'identification du serveur, et la date de dernière modification de la ressource,
- un corps, séparé de l'entête par une ligne vide, qui fournit une représentation de la ressource *(contenu du document)* correspondant à l'URL de la requête.

3.2.3 Statuts de la réponse

Statuts 2xx *(Success)*

– 200 Ok

La requête a été satisfaite

Statuts 3xx *(Redirection)*

– 301 Moved Permanently

La ressource se trouve à une autre adresse, mentionnée dans la directive `Location` de la réponse. Le client enchaîne directement avec une deuxième requête vers cette nouvelle URL.

– 304 Not Modified

La version cachée du client est à jour. La ressource est à prendre dans le cache.

Statuts 4xx *(Client Error)*

– 400 Bad Request

Erreur de syntaxe dans la requête.

On ne voit jamais ce type de réponse avec avec un client stable, mais assez souvent si l'on compose la requête manuellement ;-)

– 401 Unauthorized

Accès interdit à l'utilisateur.

La ressource est protégée par un mot de passe, et soit celui-ci n'a pas été fourni, soit il est erroné, soit il ne donne pas accès à la ressource demandée.

– 403 Forbidden

Accès interdit au client.

La ressource est réservée à une certaine classe d'adresses IP *(cf. intranet par exemple)* dont la machine du client ne fait pas partie.

– 404 Not Found

Il n'y a pas de ressource correspondant à l'URL de la requête.

Statuts 5xx *(Server Error)*

– 500 Internal Server Error

Il y a eu un problème côté serveur *(par ex. une erreur de syntaxe dans le code de l'application)* .

[RFC 1945 #9. Status Code Definitions]

3.2.4 Entêtes de la réponse

```
HTTP/1.0 200 Ok
Content-Length: 2564
Content-Type: text/html
Last-Modified: Sun, 06 Nov 1994 08:49:37 GMT
Date: Mon, 31 Aug 1999 17:10:18 GMT
Server: Microsoft-IIS/4.0
```

Après la ligne de statut, la réponse comporte en général la date et l'heure du serveur, l'identification du logiciel serveur *(ici Microsoft-IIS/4.0)*, ainsi que des informations *(méta-informations)* concernant la ressource.

La directive `Content-Length` indique l'existence d'un corps à la réponse *(représentation de la ressource)* et en donne le nombre d'octets *(soit en général le nombre de caractères)*.

L'indication du type de contenu *(ici text/html)* **est essentielle** pour permettre au client de traiter *(afficher)* correctement l'information reçue.

La date de dernière modification est utilisée par le client pour la gestion du cache *(cf. infra)*.

3.2.5 HTTP/1.1

```
GET / HTTP/1.1
Host: www.cnam.fr
Connection:close
User-Agent:Mozilla/4.03 [fr]
```

La requête envoyée par le client au serveur comporte :

- le numéro de version HTTP/1.1 en fin de ligne de requête,
- une directive obligatoire `Host` mentionnant le nom du **serveur virtuel**.

  Cette directive était optionnelle en HTTP/1.0, bien que nécessaire pour permettre l'utilisation d'une URL relative dans la ligne de requête...
- Une directive optionnelle « `Connection: close` ».

  En son absence la connexion n'est pas immédiatement coupée par le serveur. Le client peut réutiliser la connexion pour récupérer d'autres ressources associées à la première *(images, feuilles de style, scripts, …)*.

  Plus récemment, la possibilité pour le serveur de ne pas fermer immédiatement la connexion a été exploitée pour implémenter du *server push*

  [HTTP Server Push]

Spécifications HTTP/1.1

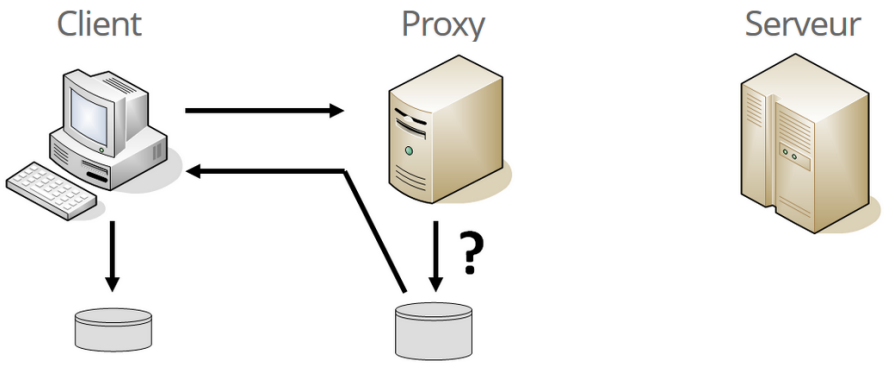
HTTP/1.1 a donné lieu à toute une série de spécifications, dont voici les principales :

- 1997 - [RFC 2068 - HTTP/1.1]
- 1997 - [RFC 2145 - HTTP/1.1 Version Numbers]
- 1999 - [RFC 2616 - HTTP/1.1]
- 2014 - [RFC 7230 - HTTP/1.1 Message Syntax and Routing]
- 2014 - [RFC 7231 - HTTP/1.1 Semantics and Content]
- 2014 - [RFC 7232 - HTTP/1.1 Conditional Requests]
- 2014 - [RFC 7233 - HTTP/1.1 Range Requests]
- 2014 - [RFC 7234 - HTTP/1.1 Caching]
- 2014 - [RFC 7235 - HTTP/1.1 Authentication]

3.2.6 Principe des caches

Un cache est une zone mémoire *(vive ou disque)* dédiée au stockage des ressources récemment consultées. Lors d'une requête ultérieure, le logiciel possédant un cache peut resservir la ressource cachée au lieu d'effectuer une nouvelle requête vers le serveur origine.

Il existe principalement deux types de caches, les caches privés gérés par le navigateur et les caches publics gérés par des machines intermédiaires ou *proxies*.



Problématique du cache

3.2.7 Méthode HEAD

Pour pouvoir servir la version cachée sans risquer de délivrer une version différente de celle présente sur le serveur origine, le gestionnaire de cache doit disposer de mécanismes qui lui permettent de vérifier auprès du serveur la validité de sa copie cachée.

La méthode `HEAD` permet à un client de ne demander au serveur que l'entête de la ressource spécifiée.

Si la réponse mentionne la date de dernière modification, le client peut décider en la comparant à celle de la version cachée, de servir celle-ci ou de refaire une requête `GET` vers le serveur pour récupérer une version plus récente.

Exemple de requête

```
HEAD /index.html HTTP/1.0
Host: www.ec-lyon.fr
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:42.0) Firefox/42.0
Connection: close
```

Exemple de réponse

```
HTTP/1.1 200 Ok
Content-Length: 1324
Content-Type: text/xml
Last-Modified: Thu, 06 Mar 2003 02:17:18 GMT
Date: Thu, 06 Mar 2003 11:45:20 GMT
Server: Apache
```

Une réponse à une requête `HEAD` ne possède pas de corps.

C'est l'un des rares cas où une directive `Content-Length` avec une valeur non nulle n'implique pas la présence d'un corps.

Pour la gestion d'un cache, cette méthode a l'inconvénient d'obliger le client à faire deux requêtes successives dans le cas où la copie cachée doit être actualisée *(date de dernière modification différente entre le serveur et la copie cachée)*.

3.2.8 GET conditionnel

La directive `If-Modified-Since` permet d'éviter au client d'émettre une nouvelle requête si la version cachée n'est plus d'actualité :

```
GET /rfc/rfc2616.txt HTTP/1.0
Host: www.ietf.org
If-Modified-Since: Fri, 11 Jun 1999 18:46:53 GMT
```

Si la version présente sur le serveur n'a pas été modifiée depuis la date spécifiée dans la requête *(qui est la date de dernière modification de la copie cachée)*, le serveur répond par un statut spécial, suivi par l'entête *(comme dans le cas d'une requête HEAD)* :

```
HTTP/1.1 304 Not Modified
Last-Modified: Fri, 11 Jun 1999 18:46:53 GMT
Date: Thu, 12 Nov 2015 17:10:37 GMT
Server: cloudflare-nginx
```

et le client peut servir la copie cachée…

Si par contre la ressource côté serveur a bien été modifiée depuis la date spécifiée dans la requête, le serveur répond comme pour une requête `GET` classique, en renvoyant la valeur de la ressource demandée dans le corps de la réponse :

```
GET /rfc/rfc2616.txt HTTP/1.0
Host: www.ietf.org
If-Modified-Since: Thu, 10 Jun 1999 00:00:00 GMT
```

```
HTTP/1.1 200 Ok
Content-Type: text/plain
Last-Modified: Fri, 11 Jun 1999 18:46:53 GMT
Date: Thu, 12 Nov 2015 17:19:45 GMT
Server: cloudflare-nginx
Connection: close
```

N.B. Lors du rechargement d'une page, les navigateurs récents utilisent systématiquement le mécanisme décrit ici pour comparer la version disponible dans leur cache avec celle du serveur, tout évitant un transfert intempestif via le réseau si la ressource n'a pas été modifiée.

***N.B.2** Il est aisé de vérifier ce point à l'aide des outils développeur de votre navigateur...*

Outre la date de dernière modification de la ressource, une réponse HTTP/1.1 possède également une directive `Etag` *(pour "Entity Tag")* qui fournit une chaîne unique *(en général un hash)* associée à une version donnée de la ressource :

```
GET /rfc/rfc2616.txt HTTP/1.0
Host: www.ietf.org
```

```
HTTP/1.1 200 Ok
Last-Modified: Fri, 11 Jun 1999 18:46:53 GMT
ETag: "8982d2a-67187-34d0931a3e140"
Date: Thu, 12 Nov 2015 17:31:38 GMT
Server: cloudflare-nginx
Connection: close
```

La directive `If-None-Match` fonctionne alors comme `If-Modified-Since` à ceci près qu'elle fournit l'Etag à comparer en lieu et place de la date :

```
GET /rfc/rfc2616.txt HTTP/1.0
Host: www.ietf.org
If-None-Match: "8982d2a-67187-34d0931a3e140"
```

La réponse sera du même type que pour `If-Modified-Since` , à savoir `304 Not Modified`, ou `200 Ok`.

Tout sur les requêtes conditionnelles : [RFC 7232 - HTTP/1.1 Conditional Requests]

Autres directives pour la gestion du cache

Il existe d'autres directives permettant de gérer au mieux le comportement des caches *(autorisation ou interdiction de cacher, en fonction de la nature du cache, durée de validité de la copie cachée, …)*

Voici par exemple comment s'y prendre pour interdire de cacher une réponse :

```
// HTTP/1.0
Pragma: no-cache

// Date dans le passé
Expires: Thu, 01 Jan 1970 00:00:00 GMT

// Date et heure courantes
Last-Modified: Thu, 12 Nov 2015 21:50:31 GMT

// HTTP/1.1
Cache-Control: no-store, no-cache, must-revalidate

// Extensions Microsoft
Cache-Control: post-check=0, pre-check=0
```

Tous les détails concernant la gestion des caches : [RFC 7234 - HTTP/1.1 Caching]

3.2.9 Redirections

Les codes de statut de la famille `3xx` indiquent que la réponse ne fournit pas la ressource demandée, mais plutôt le moyen de la trouver.

En général, le client doit enchaîner avec une autre requête pour récupérer la ressource. Cette nouvelle requête peut être menée de manière transparente pour l'utilisateur si la méthode pour cette nouvelle requête est `GET` ou `HEAD`.

Un client doit être capable de détecter des boucles de redirection infinies. Les premières versions des spécifications préconisaient un nombre maximum de 5 redirections successives.

Statuts de redirection

Le code 304 *(déjà vu)* accompagne une réponse à une requête conditionnelle, et signifie que le client peut trouver la ressource dans son cache :

```
HTTP/1.1 304 Not Modified
```

Le code 301 a initialement été conçu pour permettre au serveur d'indiquer que la ressource demandée existe toujours, mais que son URL a changé *(par exemple après réorganisation du contenu d'un serveur)*.

La nouvelle URL doit être indiquée par une directive `Location`.

```
GET http://www.w3c.org/ HTTP/1.0
Host: www.w3c.org
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:42.0) Firefox/42.0
```

```
HTTP/1.1 301 Moved permanently
Location: http://www.w3.org/
```

Lorsqu'un client reçoit ce type de réponse, il peut effectuer automatiquement une requête vers la nouvelle URL à condition que la requête initiale ne soit pas `POST` *(méthode non sûre)* car l'utilisateur peut ne pas vouloir envoyer les informations à cette nouvelle adresse.

De plus, lors de toute requête ultérieure vers l'ancienne adresse, le client est autorisé à effectuer une requête directement vers l'URL de remplacement. Sauf indication contraire explicite, une réponse de ce type peut être cachée.

N.B. Ce mécanisme est très souvent utilisé pour simplifier ou maquiller les URLs d'accès à certains services *(cf. raccourcisseurs d'URLs)*.

Le code 302 fonctionne comme 301, à ceci près que la redirection est réputée temporaire.

```
GET / HTTP/1.1
Host: campus.ec-lyon.fr
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:42.0) Firefox/42.0
```

```
HTTP/1.1 302 Found
Location: https://cas.ec-lyon.fr:443/login?service=http%3A%2F%2F...
Date: Fri, 13 Nov 2015 14:37:18 GMT
Server: Apache/1.3.37
```

Lorsqu'un client reçoit ce type de réponse, il peut effectuer automatiquement une requête vers la nouvelle URL à condition que la requête initiale ne soit pas `POST`, car l'utilisateur peut ne pas vouloir envoyer les informations à cette nouvelle adresse.

Comme la redirection est temporaire, un client doit continuer à utiliser l'URL initiale. Une réponse de ce type ne pourra être cachée qu'en cas d'autorisation explicite *(ce qui n'est pas le cas de cet exemple)*.

N.B. Comme illustré ci-dessus, ce code de statut est notamment utilisé par les mécanismes de sécurité lors de l'accès à certains intranets *(portails captifs)*.

Tout sur les statuts de redirection : [RFC 7231 - #6.4 Redirection 3xx]

3.2.10 Authentification

Le protocole HTTP prévoit des mécanismes permettant de gérer l'accès à des ressources protégées via un nom d'utilisateur et un mot de passe.

N.B. Le mécanisme permettant de protéger des ressources est propre à chaque serveur et ne sera pas explicité ici.

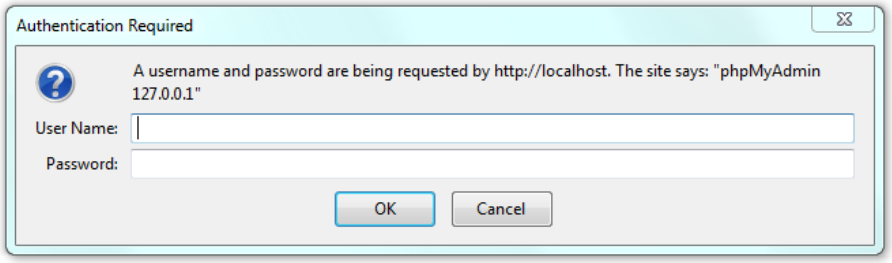
HTTP Basic

En cas de requête vers une ressource protégée via `HTTP Basic`, le serveur renvoie un code statut 401 avec une directive `WWW-Authenticate` qui indique au client comment obtenir de l'utilisateur les accréditations nécessaires *(realm)* et renvoyer les informations obtenues :

```
GET /modules/phpmyadmin3522x120919161207/ HTTP/1.1
Host: localhost
```

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="phpMyAdmin 127.0.0.1"
Content-Type: text/html; charset=utf-8
Content-Length: 1045
```

Si le client ne connait pas encore l'identification de l'utilisateur *(nom de login et mot de passe)* associée au service concerné *(realm)*, il fait apparaitre une boîte de dialogue qui mentionne le contexte *(realm)* et demande un nom d'utilisateur et un mot de passe :



Une fois le nom d'utilisateur et le mot de passe connus, le client effectue une seconde requête comportant une directive `Authorization` :

```
GET /modules/phpmyadmin3522x120919161207/ HTTP/1.1
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

On remarque que le premier champ de la directive `Authorization` rappelle au serveur le nom de la méthode d'authentification employée, à savoir `Basic`.

N.B. L'exemple ci-dessus correspond à l'envoi du nom d'utilisateur « Aladdin » et du mot de passe « open sesame » au serveur.

La série de caractères envoyés au serveur via la directive `Authorization` correspond au codage en `base64` de la chaîne obtenue en concaténant le nom d'utilisateur avec le mot de passe séparés par un caractère ":" *(douple-point)* :

```
code_base_64(user_id:password)
```

Dans le cas standard *(i.e. sauf si la session est elle-même cryptée, à l'aide de https par exemple)* cette méthode d'authentification *(qui ne s'appelle pas Basic pas hasard...)* n'est pas sûre du tout, puisque le mot de passe circule sur le réseau sous forme certes encodée, mais **non crypté**. [base 64]

Une démonstration interactive l'illustre aisément : il est aussi facile de **décoder** une chaîne base64, que de l'**encoder**.

Un pirate qui intercepterait la requête sur le réseau pourrait facilement remonter au mot de passe. Cette méthode est donc à réserver pour protéger des ressources sans véritable enjeu sécuritaire.

HTTP Digest

La méthode d'authentification `HTTP Digest` résoud le problème de la méthode `Basic`. Lorsqu'un client reçoit une réponse du type :

```
HTTP/1.1 401 Authorization Required
WWW-Authenticate: Digest realm="BE HTTP", nonce="1078748855"
```

et après récupération des accréditations de l'utilisateur, cette réponse provoque une nouvelle requête, comportant une directive `Authorization` :

```
Authorization: Digest
 username="be-http",
 realm="BE HTTP",
 uri="/~muller/be-http/digest.html",
 nonce="1078748855",
 response="53c504d3ddb566bb4233db0b2b046c50"
```

où le champ « response » de la directive `Authorization` est calculé de la façon suivante :

```
A1 = MD5(username:realm:password)
A2 = MD5(method:digestURI)
response = MD5(A1:nonce:A2)
```

`MD5` [RFC 1321] est un algorithme permettant d'encrypter les données sous la forme d'une chaîne de 128 octets *(checksum)*, ensuite représentée sous la forme de 32 caractères hexadécimaux.

A réception de la requête avec les informations d'identification, le serveur effectue le même calcul de son côté.

Il utilise pour cela des données renvoyées par le client et le mot de passe de l'utilisateur dont il dispose de son côté et qui n'a **jamais circulé en clair sur le réseau**.

Il lui suffit ensuite de comparer la chaîne obtenue avec la chaîne `response` pour décider si le client dispose bien du mot de passe.

La chaîne `nonce` peut être unique pour chaque réponse du type 401.

N.B. La méthode Digest est spécifiée et implémentée par les serveurs et les navigateurs raisonnables depuis 1997. Il a toutefois été très longtemps impossible de l'utiliser puisque apparue dans IE 7 uniquement en 2006… ! Ceci explique qu'elle soit moins répandue que Basic.

Tout savoir sur l'authentification Basic et Digest :
[RFC 2617 - Basic and Digest]

3.3 Mise en pratique

3.3.1 Serveur de ressources statiques

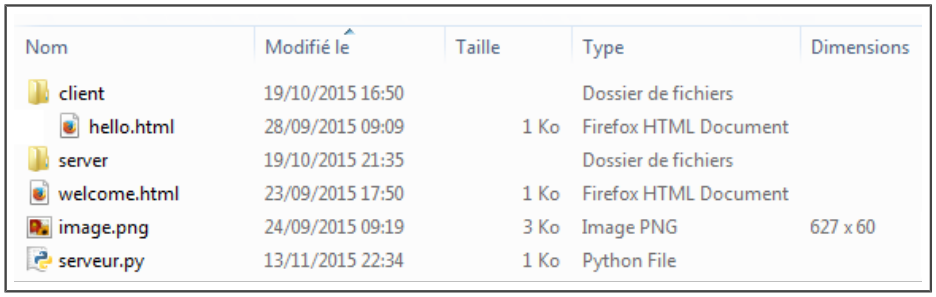
Il est très simple de réaliser un serveur de ressources statiques en python :

```
import http.server
import socketserver

# on crée une instance de la classe socketserver.TCPServer
# qui correspond au type de serveur qui nous convient
httpd = socketserver.TCPServer(

        # adresse IP par défaut 127.0.0.1 et port 8080
        ("", 8080),

        # gestionnaire de requêtes pour servir des ressources statiques
        http.server.SimpleHTTPRequestHandler
    )

# on démarre le serveur, qui se lance dans une boucle infinie
# en l'attente de requêtes provenant de clients éventuels...
httpd.serve_forever()
```

Ce programme sert tous les fichiers du répertoire dans lequel il se trouve, et de ses sous-répertoires.

[Doc. SimpleHTTPRequestHandler]

En supposant que l'on ait démarré ce serveur sous le nom `serveur.py` dans le répertoire suivant :



Nom	Modifié le	Taille	Type	Dimensions
client	19/10/2015 16:50		Dossier de fichiers	
hello.html	28/09/2015 09:09	1 Ko	Firefox HTML Document	
server	19/10/2015 21:35		Dossier de fichiers	
welcome.html	23/09/2015 17:50	1 Ko	Firefox HTML Document	
image.png	24/09/2015 09:19	3 Ko	Image PNG	627 x 60
serveur.py	13/11/2015 22:34	1 Ko	Python File	

Voici les URLs permettant d'accéder aux fichiers disponibles dans cette arborescence :

fichier	URL
client	http://localhost:8080/client/
hello.html	http://localhost:8080/client/hello.html
welcome.html	http://localhost:8080/welcome.html
image.png	http://localhost:8080/image.png
serveur.py	http://localhost:8080/serveur.py

Ce tableau ainsi que l'implémentation et l'utilisation de ce serveur appellent plusieurs remarques :

- Ce serveur sait délivrer divers types de ressources *(html, png...)*.
- Si l'on demande une ressource dont l'adresse correspond à un répertoire, le serveur génère une page qui affiche le contenu du répertoire.
- Le chemin *(path)* de l'URL d'une ressource correspond au chemin de cette ressource sur le disque, relatif au répertoire dans lequel le serveur a été démarré. On appelle ce répertoire le *répertoire racine* du serveur.

D'autres expériences intéressantes peuvent êtres tentées, en analysant les résultats obtenus à l'aide des outils développeurs de votre navigateur :

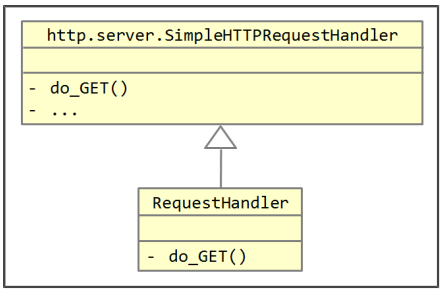
URL	Réponse
`http://localhost:8080/tagada.html`	`404 File not found`
`http://localhost:8080/client`	`301 Moved Permanently` `Location: /client/`

Bref, ce serveur élémentaire se comporte de manière globalement conforme à ce qu'on attend...

Personnalisation du serveur

Toutefois, dans cette configuration, ce serveur n'est **pas du tout sécurisé**, puisqu'il va notamment jusqu'à délivrer son propre code source ! La prochaine étape consiste à lui faire délivrer uniquement le contenu du répertoire nommé `client`.

Pour cela il faut personnaliser le comportement du serveur en créant une sous-classe qui hérite de `http.server.SimpleHTTPRequestHandler` :



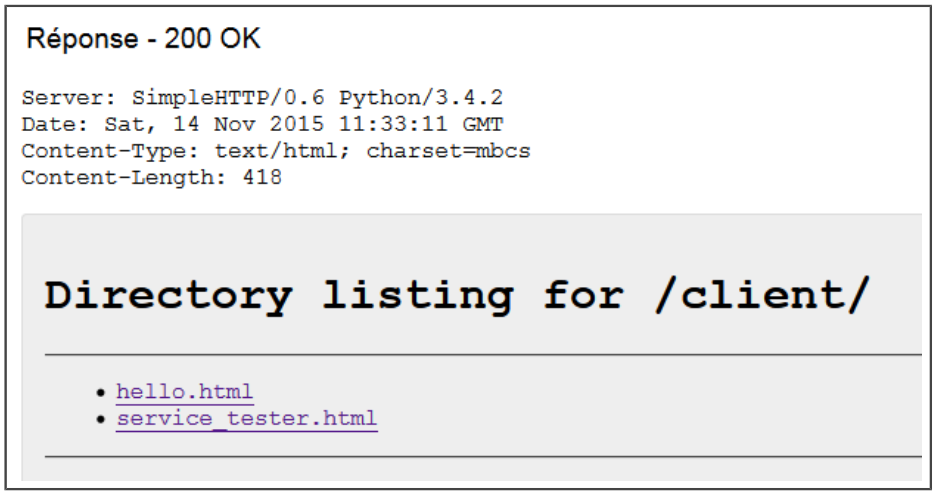
La sous-classe `RequestHandler` surchargera la méthode `do_GET()` qui traite les requêtes `GET` pour aller chercher les ressources dans le répertoire `/client` au lieu du répertoire racine du serveur.

Voici le code source de ce nouveau serveur :

```
import http.server
import socketserver

# définition du nouveau handler
class RequestHandler(http.server.SimpleHTTPRequestHandler):

  # sous-répertoire racine des documents statiques
  static_dir = '/client'

  # on surcharge la méthode qui traite les requêtes GET
  def do_GET(self):

    # on modifie le chemin d'accès en insérant un répertoire préfixe
    self.path = self.static_dir + self.path

    # on traite la requête via la classe parent
    http.server.SimpleHTTPRequestHandler.do_GET(self)

# instanciation et lancement du serveur
httpd = socketserver.TCPServer(("", 8080), RequestHandler)
httpd.serve_forever()
```

Et les caractéristiques de la réponse obtenue pour la requête :

```
GET / HTTP/1.1
Host: localhost:8080
```



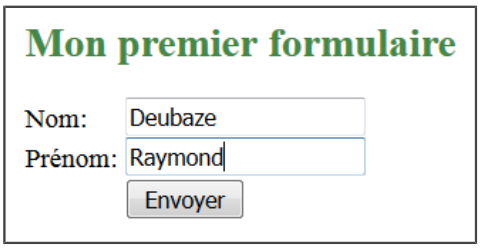
Où l'on observe bien le contenu du répertoire `/client` alors que la requête portait sur le répertoire racine `/`.

N.B. Ce résultat a été obtenu avec le testeur de service *(dont on aperçoit l'existence dans le contenu du répertoire ci-dessus)* qui vous sera fourni pour le TD N°2.

3.3.2 GET avec chaîne de requête

Lorsqu'on crée une page HTML contenant un formulaire il faut préciser la ressource qui sera appelée par le navigateur lors de la soumission du formulaire pour le traitement des valeurs entrées par l'utilisateur :

```
<form action="/service">
 <label>Nom: <input name="Nom"></label><br>
 <label>Prénom: <input name="Prenom"></label><br>
 <input type="submit" value="Envoyer">
</form>
```



Lors de la soumission de ce formulaire, le client va former la requête suivante :

```
GET /service?Nom=Deubaze&Prenom=Raymond HTTP/1.1
```

Il s'agit alors, côté serveur :

1. d'effectuer un traitement spécial lorsque l'URL commence par `/service`,
2. de récupérer les valeurs entrées par l'utilisateur,
3. de les traiter de manière appropriée,
4. de renvoyer un compte-rendu d'exécution pertinent à l'utilisateur.

1. Sur la base du serveur précédent, voici comment tester si le chemin commence par la chaîne `/service` :

```
  def do_GET(self):

    # on vérifie si le chemin commence par /service
    if self.path[0:8] == "/service":
      # on traite le retour de formulaire

    else:
      # on traite la requête vers un document statique
```

2. Un peu plus de réflexion est nécessaire pour le décodage de la chaîne de requête et la récupération des paramètres :

```
from urllib.parse import unquote

  def init_params(self):
    self.params = {}

    info = self.path.split('?',2)
    self.query_string = info[1] if len(info) > 1 else ''

    for c in self.query_string.split('&'):
      (k,v) = c.split('=',2) if '=' in c else ('',c)
      self.params[unquote(k)] = unquote(v)
```

Pour comprendre à quoi sert la fonction `unquote()` du module `urllib.parse`, voici la requête envoyée par le navigateur lorsque le formulaire contient un champ nommé `Prénom` que l'utilisateur a renseigné avec la valeur `Dédé` :

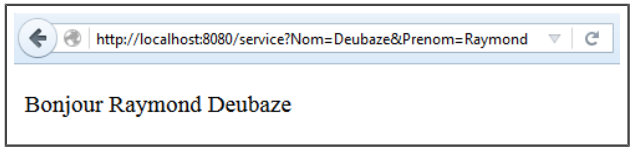
```
GET /service?Nom=Deubaze&Pr%C3%A9nom=D%C3%A9d%C3%A9 HTTP/1.1
```

Tout sur l'encodage des URLs : [Percent Encoding]

3. et 4. En supposant que le traitement des informations de notre formulaire consiste à saluer l'utilisateur en renvoyant les valeurs qu'il a communiquées au serveur, voici une solution :

```
def do_GET(self):
  if self.path[0:8] == "/service":
    self.init_params()
    response = '<!DOCTYPE html><title>hello</title>' + \
      '<meta charset="utf-8"><p>Bonjour {} {}</p>' \
      .format(self.params['Prenom'],self.params['Nom'])
    self.send(response)

def send(self,body):
   self.send_response(200)
   self.end_headers()
   encoded = bytes(body, 'UTF-8')
   self.wfile.write(encoded)
```

Le serveur ainsi modifié permet d'obtenir le résultat suivant après soumission du formulaire :



3.3.3 RESTful GET

Comme il a été vu, l'architecture REST ne préconise pas le passage d'information au serveur via la chaîne de requête, mais utilise l'URL pour identifier une ressource.

Le message de bienvenue destiné à une personne serait ainsi identifié par l'URL :

```
/greetings/=prénom/=nom
```

Avec les valeurs entrées par l'utilisateur, le formulaire précédent devrait alors être soumis à l'adresse :

```
/greetings/Raymond/Deubaze
```

Il n'est pas très compliqué de modifier notre serveur pour répondre correctement à une telle requête :

```
def do_GET(self):
  if self.path[0:10] == "/greetings":
    self.init_params()
    response = '<!DOCTYPE html><title>hello</title>' +\
      '<meta charset="utf-8"><p>Bonjour {} {}</p>' \
      .format(*self.path_info[1:])
    self.send(response)

def init_params(self):
  info = self.path.split('?',2)
  self.path_info = [unquote(v) for v in info[0].split('/')[1:]]
```

Côté navigateur par contre, il n'est pas naturel d'appeler une telle URL suite à la soumission d'un formulaire. Il est donc nécessaire de recourir à du code Javascript pour effectuer un appel AJAX :

```
<form action="#" id="form">
 <label>Nom: <input id="Nom"></label><br>
 <label>Prénom: <input id="Prenom"></label><br>
 <input type="submit" value="Envoyer">
</form>
<script>
form.onsubmit = function(e) {
 var xhr = new XMLHttpRequest();
 xhr.open('GET', '/greetings/'+Prenom.value+'/'+Nom.value, true);
 xhr.onload = function(e) {
   if ( this.status == 200 ) {
     document.body.innerHTML = this.response;
   }
 };
 xhr.send();
 e.preventDefault();
}
</script>
```

N.B. Le code ci-dessus est fortement basé sur des particularités HTML5 et fonctionne uniquement dans les navigateurs récents.

N.B.2 La programmation côté client sera abordée lors du prochain cours... Pour les impatients : [XMLHttpRequest2]

3.3.4 POST de formulaire

Dans le cas de formulaires complexes *(en nombre de champs et taille de l'information attendue)*, lorsqu'on ne désire pas que l'utilisateur ait les paramètres de l'application sous les yeux *(via l'URL)*, ou tout simplement lorsque le traitement côté serveur n'est pas **sûr** et **idempotent** on recourt à la méthode `POST` plutôt que `GET`.

Le formulaire lui-même n'est pas différent, à part l'attribut `method="POST"` :

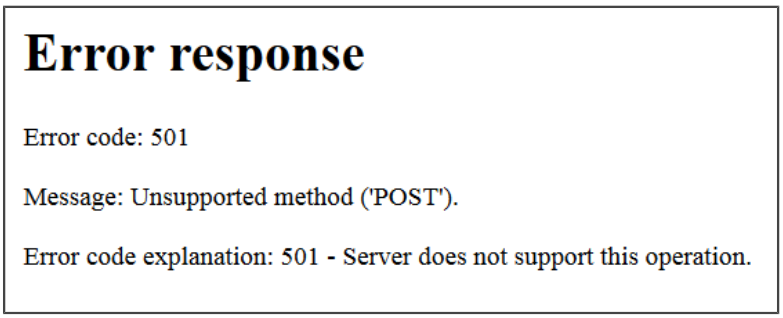
```
<form action="/service" method="POST">
 <label>Nom: <input name="Nom"></label><br>
 <label>Prénom: <input name="Prenom"></label><br>
 <input type="submit" value="Envoyer">
</form>
```

À la soumission de ce formulaire le navigateur va cette fois-ci formuler une requête `POST` et placer les paramètres **dans le corps de la requête** au lieu de les transmettre via l'URL :

```
POST /service HTTP/1.0
Host: localhost:8080
Content-Type: application/x-www-form-urlencoded
Content-Length: 26

Nom=Deubaze&Prenom=Deubaze
```

Noter la présence de `Content-Length` qui indique la présence d'un corps à la requête, et le type de contenu qui suggère qu'il s'agit des paramètres d'un formulaire Web encodés comme dans une URL...

En soumettant ce formulaire tel quel à notre serveur on obtient une réponse avec un code d'erreur car il n'est pas programmé pour accepter les requêtes `POST` :



Pour accepter les requêtes `POST`, il faut munir le serveur d'une méthode nommée `do_POST()` :

```
  def do_POST(self):
    self.init_params()
    print(self.path_info[0])
    if self.path_info[0] == 'service':
      response = '<!DOCTYPE html><title>hello</title>' + \
        '<meta charset="utf-8"><p>Bonjour {} {}</p>' \
        .format(self.params['Prenom'][0],self.params['Nom'][0])
      self.send(response)
    else:
      self.send_error(405)
```

Il faut également modifier la méthode `init_params()` pour récupérer non seulement les éléments du chemin d'accès et les paramètres de la requête, mais aussi le corps de la requête et les paramètres qui s'y trouvent :

```
from urllib.parse import urlparse, parse_qs, unquote

  def init_params(self):
    # analyse de l'adresse
    info = urlparse(self.path)
    self.path_info = [unquote(v) for v in info.path.split('/')[1:]]
    self.query_string = info.query
    self.params = parse_qs(info.query)

    # récupération du corps
    length = self.headers.get('Content-Length')
    ctype = self.headers.get('Content-Type')
    if length:
      self.body = str(self.rfile.read(int(length)),'utf-8')
      if ctype == 'application/x-www-form-urlencoded' :
        self.params = parse_qs(self.body)
```

Noter un meilleur usage du module `urllib` que précédemment, pour récupérer les paramètres de l'URL. [Doc. urllib.parse]

Que renvoie le serveur suite à une requête `POST` dont l'URL ne commence pas par `/service` ?

3.4 Gestion de la persistance côté serveur

3.4.1 Fichiers texte

La manière la plus simple pour assurer la persistance de données côté serveur consiste à écrire les informations souhaitées, sous forme de texte, dans un fichier.

Il est possible pour cela d'inventer son propre format *(structure et contenu du fichier)*. On parle alors de **format propriétaire**.

Il est toutefois plus courant d'utiliser un format de fichier standard pour bénéficier d'outils existants qui en facilitent la lecture et l'écriture.

Format CSV

Un fichier CSV *(Comma-Separated Values)* [CSV sur Wikipédia] est constitué de lignes correspondant chacune à l'enregistrement d'un objet. L'ensemble du fichier correspond ainsi à une **liste d'objets**.

Au sein de chacune des lignes, les valeurs des attributs sont séparées par un caractère particulier appelé **séparateur**. Le caractère séparateur est souvent le *point-virgule* « ; », la *tabulation*, ou plus rarement la *virgule*. [RFC 4180 - CSV]

Parfois, la première ligne sert à identifier les noms des attributs :

```
Prénom;Nom;email
Raymond;Deubaze;rdeubaze@ec-lyon.fr
Jean;Aymard;jeanAymard@wanadoo.fr
Elsa;Plique;elsaPlique@caramail.com
```

En Python, la lecture et l'écriture de fichiers CSV peuvent s'effectuer grâce au module `csv`.
[Python Module csv]

Voici comment lire le fichier précédent, que l'on suppose nommé `personnes.csv` :

```
import csv

personnes = []
with open('personnes.csv',encoding="utf-8") as csvfile:
    reader = csv.DictReader(csvfile,delimiter=';')
    for row in reader:
        personnes.append(row)
```

L'écriture du même fichier s'effectue de la manière suivante :

```
with open('personnes.csv', 'w', encoding='utf-8', newline='\n') as f:
    fieldnames = ['Prénom', 'Nom', 'email']
    writer = csv.DictWriter(f, fieldnames=fieldnames, delimiter=';')

    writer.writeheader()
    for p in personnes:
        writer.writerow(p)
```

N.B. *L'implémentation d'un serveur `REST` pour la gestion d'une liste de personnes enregistrée dans un fichier `CSV` est laissée à la sagacité du lecteur...*

Format JSON

JSON *Javascript Simple Object Notation* est un format texte qui permet de représenter des objets sous forme de listes et de dictionnaires éventuellement imbriqués. [Syntaxe JSON] [JSON sur Wikipédia]

Voici comment enregistrer au format JSON la même liste de personnes que précédemment à l'aide d'un programme en Python :

```
import json

with open('personnes.json', 'w', encoding='utf-8') as jsonfile:
    json.dump(personnes, jsonfile, indent=2)
```

La relecture du fichier JSON est tout aussi simple :

```
with open('personnes.json',encoding="utf-8") as jsonfile:
    personnes = json.load(jsonfile)
```

[Python Module json]

Lors de l'enregistrement nous avons spécifié que nous désirions indenter le résultat afin de le rendre plus lisible par des humains. Sous cette condition le contenu du fichier ressemblera à :

```
[
  {
    "Pr\u00e9nom": "Raymond",
    "Nom": "Deubaze",
    "email": "rdeubaze@ec-lyon.fr"
  },
  {
    "Pr\u00e9nom": "Jean",
    "Nom": "Aymard",
    "email": "jeanAymard@wanadoo.fr"
  }
]
```

Sans cette option le résultat aurait été plus compact et donc moins lisible, tout en gardant la même signification :

```
[{"Pr\u00e9nom": "Raymond", "Nom": "Deubaze", "email": "rdeubaze@ec-ly
on.fr"}, {"Pr\u00e9nom": "Jean", "Nom": "Aymard", "email": "jeanAymard
@wanadoo.fr"}]
```

Format XML

Le format XML est un peu plus complexe à travailler que les précédents. Il reste toutefois très utilisé car il possède un écosystème riche avec énormément d'outils. [XML sur Wikipédia]

En Python, la création d'un fichier XML à partir de notre liste de personnes peut s'effectuer de la manière suivante :

```
from lxml import etree

# création de l'arbre XML en mémoire
root = etree.fromstring('<personnes/>')
for p in personnes:
    personne = etree.SubElement(root, 'personnes')
    for a in ['Prénom','Nom','email']:
            tag = etree.SubElement(personne, a)
            tag.text = p[a]

# sérialisation dans un fichier
with open('personnes.xml', 'wb') as f:
    f.write(etree.tostring(root, pretty_print=True, encoding='utf-8'))
```

N.B. Là encore, le paramètre `prettyPrint=True` passé à la génération du fichier indique que nous désirons une version indentée, plus facilement lisible par les humains. Sans cette option le fichier aurait été plus compact, tout en restant opérationnel.

Voici le contenu du fichier XML généré :

```
<personnes>
  <personne>
    <Prénom>Raymond</Prénom>
    <Nom>Deubaze</Nom>
    <email>rdeubaze@ec-lyon.fr</email>
  </personne>
  <personne>
    <Prénom>Jean</Prénom>
    <Nom>Aymard</Nom>
    <email>jeanAymard@wanadoo.fr</email>
  </personne>
  ...
</personnes>
```

Et voilà comment le relire :

```
personnes = []
doc = etree.parse("personnes.xml")
for p in doc.getroot():
    personne = {}
    for a in p:
        personne[a.tag] = a.text
    personnes.append(personne)
```

[Python Module lxml].

3.4.2 Bases de données relationnelles

Lorsque le volume des données ou le nombre d'accès croissent ou lorsque le modèle devient plus complexe, il n'est plus possible de lire ou d'écrire des fichiers entiers à chaque modification. On a alors recours à une base de données.

Base SQLite

La solution la plus simple à mettre en oeuvre consiste à utiliser SQLite. Il s'agit d'un format de base de données relationnelle stockée dans un fichier plat. [SQLite]

Pour enregistrer notre liste de personnes, il faut tout d'abord créer la base et une table pour recevoir les données : [Python Module sqlite3]

```
import sqlite3
conn = sqlite3.connect('personnes.sqlite')

c = conn.cursor()
c.execute("DROP TABLE IF EXISTS personnes")
c.execute("CREATE TABLE personnes (id INTEGER PRIMARY KEY, Prénom, Nom, email)")
conn.commit()
```

logo SQLite

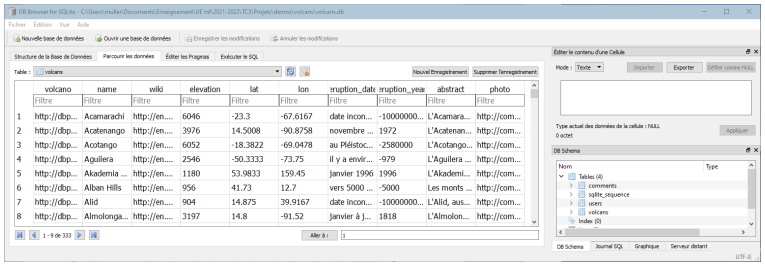
On peut ensuite enregistrer les personnes dans la table éponyme :

```
for p in personnes:
  c.execute('INSERT INTO personnes VALUES (NULL, ?, ?, ?)',
            (p['Prénom'], p['Nom'], p['email']))
  conn.commit()
```

et relire les données de la manière suivante :

```
conn.row_factory = sqlite3.Row
c = conn.cursor()

c.execute("SELECT * FROM personnes ORDER By Nom,Prénom")
a = c.fetchall()

personnes = []
for r in a:
    personne = {}
    for k in r.keys():
        personne[k] = r[k]
    personnes.append(personne)
```

N.B. L'utilisation d'une base de données SQLite a fait l'bjet d'un TD INF-tc2 et sera reprise lors du TD n°1



DB Browser for SQLite

Autres systèmes de bases de données relationnelles

Il existe d'autres systèmes de gestion de bases de données relationnelles qui, au contraire de SQLite, fonctionnent en mode client-serveur :

- La base de données est gérée par un programme serveur qui s'occupe de la gestion des droits d'accès, des accès concurrents, des transactions...
- Les accès se font via un protocole propriétaire par des applications clientes, développées à l'aide de bibliothèques ou modules dédiés.

Entrent dans cette catégorie des bases de données comme MySQL, MariaDB, PostgreSQL, Oracle...

[MySQL sur Wikipédia] [MariaDB sur Wikipédia] [PostgreSQL sur Wikipédia] [Oracle sur Wikipédia]

N.B.L'utilisation de ces bases de données n'entre pas dans le cadre de ce cours.

3.4.3 Bases NoSQL

> *« En informatique, NoSQL (Not only SQL en anglais) désigne une catégorie de systèmes de gestion de base de données (SGBD) qui n'est plus fondée sur l'architecture classique des bases relationnelles. L'unité logique n'y est plus la table, et les données ne sont en général pas manipulées avec SQL. »*
>
> —Wikipédia

Les bases NoSQL ont été initialement développées pour répondre aux énormes besoins des géants de l'Internet *(Google, Amazon, Facebook, Twitter...)* pour lesquels les bases relationnelles ne convenaient plus :

- distribuer les **traitements** sur un grand nombre de machines *(besoins CPU)*,
- répartir les **données** entre un grand nombre de machines *(besoins en espace de stockage)*,
- dupliquer les données entre plusieurs datacenters pour en assurer la disponibilité même en cas de panne.

Limites des SGBD classiques

Le modèle relationnel *(comme son nom l'indique)* est basé sur :

- l'existence de liens *(jointures)* entre entités *(tables)*,
- un principe d'intégrité référentielle qui garantit que les liens entre tables sont toujours valides.

La plupart des SGBD relationnels sont également transactionnels, c'est à dire qu'ils respectent un certain nombre de conditions connues sous l'acronyme ACID. Une transaction est dite ACID si elle respecte les propriétés d'**Atomicité**, de **Cohérence**, d'**Isolation** et de **Durabilité** *(cf. infra.)*. [ACID]

Des études théoriques [Théorème CAP] ont montré par ailleurs qu'un système distribué ne peut pas offrir à la fois :

- de la cohérence *(tous les noeuds du système disposent à tout instant des mêmes données)*,
- de la disponibilité *(toutes les requêtes recoivent une réponse)*,
- la tolérance au partionnement *(coupures aléatoires des liaisons entre noeuds du système)*.

En fonction de leur modèle les bases NoSQL sacrifient souvent l'intégrité référentielle et l'existence de transactions *(qui sont très difficiles à garantir à l'échelle d'un système distribué),* et l'un des points du Théorème CAP.

Atomicité

L'atomicité est la propriété qui garantit qu'une transaction est effectuée en totalité ou pas du tout. Lors d'un virement bancaire, nul ne souhaiterait que le premier compte soit débité alors que le second ne serait pas crédité...

Cohérence *(Consistency)*

Cette propriété signifie qu'après exécution d'une transaction le système sera toujours dans un état stable *(liens entre entités valides)*.

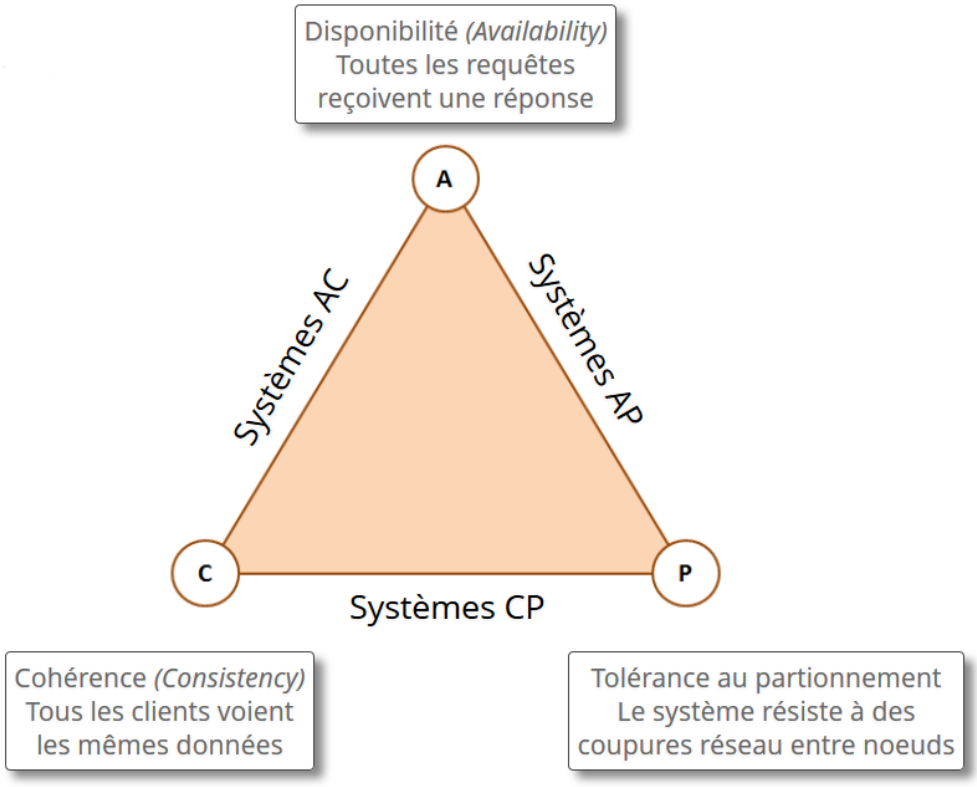
Isolation

Lorsque plusieurs transactions sont menées en parallèle sur le système, elles ne doivent pas interagir et se dérouler comme si elles s'effectuaient en séquence. Par exemple, si une transaction est en cours, une requête de lecture ne doit jamais montrer un état intermédiaire des données.

Durabilité

Cette propriété assure qu'une transaction ne peut pas être «oubliée» ou perdue, même dans les pires circonstances *(crash système)*. Toutes les transactions validées doivent être sauvegardées dans un fichier journal pour, au pire, pouvoir les rejouer à partir d'une version antérieure de la base.

Théorème CAP



Les bases de données relationnelles sont des systèmes AC *(Cohérence et disponibilité)*. Les bases de données NoSQL sont des systèmes CP *(Cohérence et Résistance au partitionnement)* ou AP *(Disponibilité et Résistance au partitionnement)*. [Livre blanc NoSQL]

Références

[RFC 1738 #2. General URL Syntax], p.3
T. BERNERS-LEE, L. MASINTER, M. MCCAHILL, *"RFC 1738 - Uniform Resource Locators (URL)",* IETF (Internet Engineering Task Force), décembre 1994, p. 2.
En ligne, disponible sur https://tools.ietf.org/html/rfc1738#section-2
[consulté le 7 nov. 2015]

[RFC 2396 #3. URI Syntactic Components], p.4
T. BERNERS-LEE, R. FIELDING, U.C. IRVINE, L. MASINTER, *"RFC 2396 - URI Generic Syntax",* IETF (Internet Engineering Task Force), août 1998, p. 11.
En ligne, disponible sur https://tools.ietf.org/html/rfc2396#section-3
[consulté le 8 nov. 2015]

[RFC 3986 #4.2 Relative references], p.5
T. BERNERS-LEE, R. FIELDING, L. MASINTER, *"RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax",* IETF (Internet Engineering Task Force), janvier 2005, p. 26.
En ligne, disponible sur https://tools.ietf.org/html/rfc3986#section-4.2
[consulté le 9 nov. 2015]

[Spécifications HTTP/0.9], p.6
T. BERNERS-LEE, *"HTTP/0.9, The Original HTTP as defined in 1991",* World Wide Web Consortium, 1991.
En ligne, disponible sur http://www.w3.org/Protocols/HTTP/AsImplemented.html
[consulté le 12 nov. 2015]

[HTTP de 1991 à 1998], p.7
H. FRYSTYK NIELSEN, *"Classic HTTP Documents",* World Wide Web Consortium, 1998.
En ligne, disponible sur http://www.w3.org/Protocols/Classic.html
[consulté le 12 nov. 2015]

[RFC 1945 #9. Status Code Definitions], p.8
T. BERNERS-LEE, R. FIELDING, H. FRYSTYK , *"RFC 1945 - Hypertext Transfer Protocol -- HTTP/1.0",* IETF (Internet Engineering Task Force), mai 1996, p. 32.
En ligne, disponible sur https://tools.ietf.org/html/rfc1945#section-9
[consulté le 12 nov. 2015]

[HTTP Server Push], p.9
WIKIPÉDIA (ÉD.), *"Push Technology",* .
En ligne, disponible sur https://en.wikipedia.org/wiki/Push_technology#HTTP_server_push
[consulté le 12 nov. 2015]

[RFC 2068 - HTTP/1.1], p.9
R. FIELDING, J. GETTYS, J. MOGUL, H. FRYSTYK, T. BERNERS-LEE, *"RFC 2068 - Hypertext Transfer Protocol -- HTTP/1.1",* IETF (Internet Engineering Task Force) Network Working Group, janvier 1997, 162 p.
En ligne, disponible sur https://tools.ietf.org/html/rfc2068
[consulté le 12 nov. 2015]

[RFC 2145 - HTTP/1.1 Version Numbers], p.9
J. MOGUL, R. FIELDING, J. GETTYS, H. FRYSTYK, *"RFC 2145 - Use and interpretation of HTTP Version Numbers",* IETF (Internet Engineering Task Force) Network Working Group, mai 1997, 7 p.
En ligne, disponible sur https://tools.ietf.org/html/rfc2145
[consulté le 12 nov. 2015]

[RFC 2616 - HTTP/1.1], p.9
R. FIELDING, J. GETTYS, J. MOGUL, H. FRYSTYK, L. MASINTER, P.LEACH, T. BERNERS-LEE, *"RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1",* IETF (Internet Engineering Task Force) Network Working Group, juin 1999, 176 p.
En ligne, disponible sur https://tools.ietf.org/html/rfc2616
[consulté le 12 nov. 2015]

[RFC 7230 - HTTP/1.1 Message Syntax and Routing], p.9
R. FIELDING, J. RESCHKE, *"RFC 7230 - Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing",* IETF (Internet Engineering Task Force), juin 2014, 89 p.
En ligne, disponible sur https://tools.ietf.org/html/rfc7230
[consulté le 12 nov. 2015]

[RFC 7231 - HTTP/1.1 Semantics and Content], p.9
R. FIELDING, J. RESCHKE, *"RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content",* IETF (Internet Engineering Task Force), juin 2014, 101 p.
En ligne, disponible sur https://tools.ietf.org/html/rfc7231
[consulté le 12 nov. 2015]

[RFC 7232 - HTTP/1.1 Conditional Requests], p.9
R. FIELDING, J. RESCHKE, *"RFC 7232 - Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests",* IETF (Internet Engineering Task Force), juin 2014, 28 p.
En ligne, disponible sur https://tools.ietf.org/html/rfc7232
[consulté le 12 nov. 2015]

[RFC 7233 - HTTP/1.1 Range Requests], p.9
R. FIELDING, Y. LAFON, J. RESCHKE, *"RFC 7233 - Hypertext Transfer Protocol (HTTP/1.1): Range Requests",* IETF (Internet Engineering Task Force), juin 2014, 25 p.
En ligne, disponible sur https://tools.ietf.org/html/rfc7233
[consulté le 12 nov. 2015]

[RFC 7234 - HTTP/1.1 Caching], p.9
R. FIELDING, M. NOTTINGHAM, J. RESCHKE, *"RFC 7234 - Hypertext Transfer Protocol (HTTP/1.1): Caching",* IETF (Internet Engineering Task Force), juin 2014, 43 p.
En ligne, disponible sur https://tools.ietf.org/html/rfc7234
[consulté le 12 nov. 2015]

[RFC 7235 - HTTP/1.1 Authentication], p.9
R. FIELDING, J. RESCHKE, *"RFC 7235 - Hypertext Transfer Protocol (HTTP/1.1): Authentication",* IETF (Internet Engineering Task Force), juin 2014, 176 p.
En ligne, disponible sur https://tools.ietf.org/html/rfc7235
[consulté le 12 nov. 2015]

[base 64], p.14
WIKIPÉDIA (ÉD.), *"Base64",* .
En ligne, disponible sur https://fr.wikipedia.org/wiki/Base64
[consulté le 13 nov. 2015]

[RFC 1321], p.15
R. RIVEST, *"RFC 1321 - The MD5 Message-Digest Algorithm",* IETF (Internet Engineering Task Force) Network Working Group, Avril 1992, 21 p.
En ligne, disponible sur https://tools.ietf.org/html/rfc1321
[consulté le 13 nov. 2015]

[RFC 2617 - Basic and Digest], p.15
J. FRANKS, P. HALLAM-BAKER, J. HOSTETLER, S. LAWRENCE, P. LEACH, A. LUOTONEN, L. STEWART, *"RFC 2617 - HTTP Authentication: Basic and Digest Access Authentication",* IETF (Internet Engineering Task Force) Network Working Group, Juin 1999, 34 p.
En ligne, disponible sur https://tools.ietf.org/html/rfc2617
[consulté le 13 nov. 2015]

[Doc. SimpleHTTPRequestHandler], p.16
PYTHON SOFTWARE FOUNDATION (ÉD.), *"HTTP Servers - Class http.server.SimpleHTTPRequestHandler",* 2 nov. 2015.
En ligne, disponible sur https://docs.python.org/3/library/http.server.html#http.server.SimpleHTTPRequestHandler
[consulté le 14 nov. 2015]

[Percent Encoding], p.19
WIKIPÉDIA (ÉD.), *"Percent-encoding",* .
En ligne, disponible sur https://en.wikipedia.org/wiki/Percent-encoding
[consulté le 14 nov. 2015]

[XMLHttpRequest2], p.20
E. BIDELMAN, *"New Tricks in XMLHttpRequest2",* 27 mai 2011.
En ligne, disponible sur http://www.html5rocks.com/en/tutorials/file/xhr2/
[consulté le 14 nov. 2015]

[Doc. urllib.parse], p.22
PYTHON SOFTWARE FOUNDATION (ÉD.), *"urllib.parse — Parse URLs into components",* 14 fév. 2009.
En ligne, disponible sur https://docs.python.org/3.0/library/urllib.parse.html#module-urllib.parse
[consulté le 15 nov. 2015]

[CSV sur Wikipédia], p.22
WIKIPÉDIA (ÉD.), *"Comma-separated values",* 16 juil. 2015.
En ligne, disponible sur https://fr.wikipedia.org/wiki/Comma-separated_values
[consulté le 15 nov. 2015]

[RFC 4180 - CSV], p.22
Y. SHAFRANOVICH, *"RFC 4180 - Common Format and MIME Type for Comma-Separated Values (CSV) Files",* IETF (Internet Engineering Task Force) Network Working Group, Oct. 2005, 8 p.
En ligne, disponible sur https://tools.ietf.org/html/rfc4180
[consulté le 15 nov. 2015]

[Python Module csv], p.22
PYTHON SOFTWARE FOUNDATION (ÉD.), *"csv — CSV File Reading and Writing",* 11 sept. 2015.
En ligne, disponible sur https://docs.python.org/3.4/library/csv.html
[consulté le 15 nov. 2015]

[Syntaxe JSON], p.23
(ÉD. ?), *"Présentation de JSON",* .
En ligne, disponible sur http://www.json.org/json-fr.html
[consulté le 15 nov. 2015]

[JSON sur Wikipédia], p.23
WIKIPÉDIA (ÉD.), *"JavaScript Object Notation",* 12 nov. 2015.
En ligne, disponible sur https://fr.wikipedia.org/wiki/JavaScript_Object_Notation
[consulté le 15 nov. 2015]

[Python Module json], p.23
PYTHON SOFTWARE FOUNDATION (ÉD.), *"json — JSON encoder and decoder",* 10 oct. 2015.
En ligne, disponible sur https://docs.python.org/3.4/library/json.html
[consulté le 15 nov. 2015]

[XML sur Wikipédia], p.24
WIKIPÉDIA (ÉD.), *"Extensible Markup Language",* 8 nov. 2015.
En ligne, disponible sur https://fr.wikipedia.org/wiki/Extensible_Markup_Language
[consulté le 16 nov. 2015]

[Python Module lxml], p.24
S. BEHNEL, *"lxml - XML and HTML with Python",* 13 nov. 2015.
En ligne, disponible sur http://lxml.de/index.html
[consulté le 16 nov. 2015]

[SQLite], p.25
D.R. HIPP, D. KENNEDY, J. MISTACHKIN, *"SQLite Home Page",* SQLite Consortium.
En ligne, disponible sur https://www.sqlite.org/
[consulté le 16 nov. 2015]

[Python Module sqlite3], p.25
PYTHON SOFTWARE FOUNDATION (ÉD.), *"sqlite3 — DB-API 2.0 interface for SQLite databases",* 2 nov. 2015.
En ligne, disponible sur https://docs.python.org/3.4/library/sqlite3.html
[consulté le 16 nov. 2015]

[MySQL sur Wikipédia], p.26
WIKIPÉDIA (ÉD.), *"MySQL",* 16 oct. 2015.
En ligne, disponible sur https://fr.wikipedia.org/wiki/MySQL
[consulté le 16 nov. 2015]

[MariaDB sur Wikipédia], p.26
WIKIPÉDIA (ÉD.), *"MariaDB",* 19 oct. 2015.
En ligne, disponible sur https://fr.wikipedia.org/wiki/MariaDB
[consulté le 16 nov. 2015]

[PostgreSQL sur Wikipédia], p.26
WIKIPÉDIA (ÉD.), *"PostgreSQL",* 13 nov. 2015.
En ligne, disponible sur https://fr.wikipedia.org/wiki/PostgreSQL
[consulté le 16 nov. 2015]

[Oracle sur Wikipédia], p.26
WIKIPÉDIA (ÉD.), *"Oracle Database",* 16 nov. 2015.
En ligne, disponible sur https://fr.wikipedia.org/wiki/Oracle_Database
[consulté le 16 nov. 2015]

[ACID], p.26
WIKIPÉDIA (ÉD.), *"Propriétés ACID",* 11 juin 2015.
En ligne, disponible sur https://fr.wikipedia.org/wiki/Propri%C3%A9t%C3%A9s_ACID
[consulté le 16 nov. 2015]

[Théorème CAP], p.26
WIKIPÉDIA (ÉD.), *"Théorème CAP",* 26 mai 2015.
En ligne, disponible sur https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_CAP
[consulté le 17 nov. 2015]

[Livre blanc NoSQL], p.27
A. FOUCRET, *"NoSQL - Une nouvelle approche du stockage et de la manipulation de données",*
Smile - Open source solutions, 2011, p. 12, 55 p.
En ligne, disponible sur http://www.smile.fr/Livres-blancs/Culture-du-web/Nosql
[consulté le 17 nov. 2015]