Les ressources

Les ressources sont vraiment **le coeur** d'une API ! Considérez-les comme une réduction de vos données aux éléments les plus centraux. Comme une API REST est un système d'URI, les ressources déterminent la structure des URI et, du coup, la manière dont une application trouve vos données. "Ressource", c'est même le deuxième mot dans l'acronyme "URI." Prenons donc le temps de bien comprendre ce qu'est une ressource.

Comme nous l'avons vu précédemment, des services externes comme Instagram, Gmail, etc. rendent accessible certaines de leurs données à travers une API REST. Les choses auxquelles vous pouvez accéder avec ces API peuvent être des utilisateurs, des photos, des températures… Ce sont des ressources !

Une ressource est un objet ou plusieurs objets auquel les utilisateurs de votre API peuvent vouloir accéder.

Accéder aux ressources

Prenons le temps de bien assimiler le concept de ressource. La première partie de l'URI est toujours la ressource :

`/users`

Si vous demandez la forme **plurielle** d'une ressource, la réponse contiendra tous les objets de cette catégorie. Ici vous recevriez une liste des utilisateurs et leurs attributs comme leur âge, leur adresse, leur taille, etc. (selon ce que l'API a rendu disponible aux clients externes).

Si vous ajoutez des éléments supplémentaires à l'URI, vous recevrez une réponse plus précise. Par exemple, si vous cherchez un utilisateur en particulier, ajoutez l'ID :

`/users/238`

Vous recevrez les données de l'utilisateur avec l'ID 238, comme son nom, son adresse, etc. selon ce que l'API a mis à votre disponibilité.

Imbrication

Selon la structure des données, une API peut rendre disponibles des ressources associées à une autre ressource, selon un schéma d'imbrication. Supposons par exemple qu'un utilisateur a des adresses associées à son compte.

`/users/238/addresses`

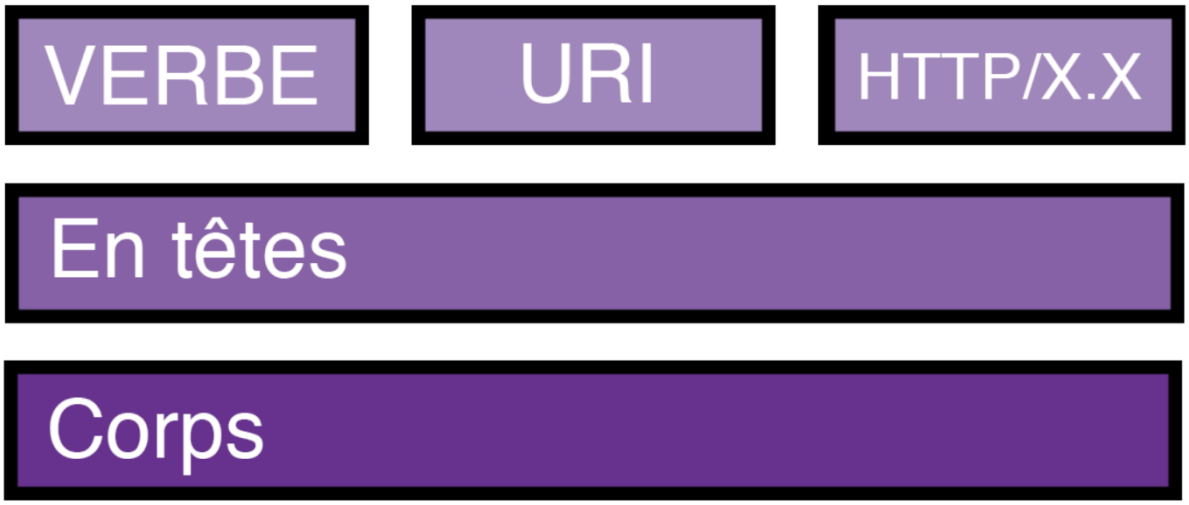
Néanmoins, cela peut devenir compliqué s'il y a plusieurs associations. Il faut toujours essayer de trouver la manière la plus simple et directe d'accéder aux données. Pour chercher des notes associées à une adresse de livraison par exemple, ce qui suit n'est pas idéal :

`/users/238/addresses/195/delivery_notes`
Ce qui suit est mieux :

`/addresses/195/delivery_notes`
Vous voyez à présent les adresses comme une ressource du premier niveau au lieu de passer par user chaque fois. Si vous utilisez une API externe, essayez toujours de construire votre application pour qu'elle envoie des requêtes d'accès aux ressources les plus directes comme dans cet exemple. Si un jour vous développez votre propre API, gardez ces principes en tête pour être sympa avec les développeurs qui utiliseront votre API !

Les requêtes

Une requête est un message qui va du client au serveur. Elle est structurée de la manière suivante :



La structure d'une requête

Première ligne (HTTP)

La première ligne d'une requête HTTP comportera toujours les éléments suivants si vous utilisez une API ou vous visitez tout simplement un site web dans votre explorateur.

`VERBE HTTP /file_path.html HTTP/X.X`
Verbe HTTP : Vous verrez une liste des verbes et leurs usages dans le chapitre Méthodes. Le verbe que vous utiliserez va déterminer l'interaction avec le serveur : est-ce que vous allez récupérer un fichier, envoyer des données avec, modifier un objet existant, ou autre chose ? Choisissez grâce à votre verbe !

Fichier : Le fichier que vous indiquez ici, précédé de son arborescence, sera récupéré par le serveur.

HTTP : la version d'HTTP que vous utilisez est toujours envoyée dans la première ligne de la requête. La version que vous verrez le plus souvent est **HTTP/1.1.**

Les en-têtes

Sous la première ligne de la requête, vous trouverez l'en-tête de la requête. Les en-têtes sont là pour donner des informations sur l'échange, souvent sur **le client** qui fait la requête. Bien sûr, vous pourriez n'envoyer que la première ligne, vous recevriez quand même une réponse du serveur, mais il vaut mieux utiliser l'en-tête pour être sûr de recevoir un contenu approprié.

Ici quelques exemples du contenu d'un en-tête :

Date

Elle indique quand le client a effectué sa demande. Cette date sera au format de date HTTP (jour de la semaine, jour, mois, an, heure en Temps Moyen de Greenwhich). Par exemple :

```
Date: Tue, 19 Jan 2016 18:15:41 GMT
```

Referer

Cela indique la précédente localisation du client. Par exemple :

```
Referer: https://www.google.fr/
```

User agent

Indique l'application utilisée par le client, ainsi que sa version. Par exemple :

```
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5)
```

User language

Renseigne sur la langue utilisée par le client dans son application. Le contenu sera envoyé dans cette langue s'il existe. Par exemple :

```
Accept-Language: en-us,en;q=0.8
```

Cookies

Est-ce que le client a déjà des cookies des requêtes avant ? Si oui ils font partie des en-têtes. Par exemple :

```
Cookie: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX (chaîne alphanumérique)
```

Les champs des en-têtes HTTP suivent toujours le motif Key: Value.
En général, il n'est pas nécessaire d'envoyer beaucoup d'en-têtes avec une requête HTTP. Renseigner tous les champs serait trop lourd ! C'est vrai que certains en-têtes sont utiles pour des clients qui visitent des sites dans leurs explorateurs, mais les autres sont plus importants pour des requêtes API comme des formats fichier acceptés et des infos sur la date de la requête.

Voici la liste complète des en-têtes HTTP possibles !

Accept
Accept-Charset
Accept-Encoding
Accept-Language
Accept-Datetime
Authorization
Cache-Control
Connection
Cookie
Content-Length
Content-MD5
Content-Type
Date
Expect
Forwarded
From
Host
If-Match
If-Modified-Since
If-None-Match
If-Range
If-Unmodified-Since
Max-Forwards
Origin
Pragma
Proxy-Authorization
Range
Referer [sic]
TE
User-Agent
Upgrade
Via
Warning

Vous pouvez trouver plus d'infos sur les en-têtes sur le site de W3C.

Corps

Après les en-têtes, vient le corps de la requête. Là-dedans vous aurez des détails selon la nature de la requête. Par exemple, si une requête envoie des données à une API avec POST (que vous allez voir dans le chapitre Méthodes) les attributs seront envoyés dans ce corps. Il n'y a pas toujours de corps dans une requête ! Cela dépend de la méthode HTTP que vous utilisez.

Ici un exemple GET pour openclassrooms.com/paths. Prenez quelques minutes pour comprendre chaque ligne en considérant les explications dans ce chapitre !

```
GET /paths HTTP/1.1
```

```
Host: openclassrooms.com:443

Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8

Accept-Encoding: gzip, deflate, sdch

Accept-Language: en-US,en;q=0.8

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2526.111 Safari/537.36
```

Les méthodes

Les méthodes HTTP sont toujours des verbes en **majuscules**. Apprenons-les !

GET

GET est la méthode la **plus utilisée** pour les requêtes HTTP. Comme le suggère le mot “GET” en anglais, cette méthode existe pour récupérer des données d'une ressource. Ce type de requête comporte la première ligne importante que vous avez vue dans le chapitre précédent et les en-têtes pour communiquer les infos du client mais n’a pas de corps, car elle ne fait que chercher des informations externes (rien à envoyer avec). Par contre, vous pouvez envoyer des paramètres à l’URI en forme d’un query string pour pouvoir recevoir les données spécifiques que vous cherchez.

```
GET http://site.com/users?parametre=exemple&amp;autre_parametre=exemple2
```

Une requête GET peut aussi être conditionnelle, par exemple GET la ressource si une condition est vraie (sinon laisse tomber).

POST

On utilise POST pour envoyer des données dans une requête et souvent pour l’ajouter à la ressource précisée dans la partie URI de la première ligne de la requête. Le type de contenu dans le corps de la requête peut être défini avec un en-tête `Content-type` pour que le serveur sache comment traiter les données dans la requête. Par exemple si vous utilisez une API pour laisser un commentaire sur un article sur un autre site la requête POST peut ressembler à la suivante :

```
POST /page.php HTTP/1.1

Host: site.com

Content-Type: application/x-www-form-urlencoded

Content-Length: 36

titre=titreici&corps=icimoncommentaire
```

Dans cette requête vous voyez deux nouveaux en-têtes. `Content-type` définit le type d’encodage utilisé pour les informations. `Content-`

`length` définit le nombre des caractères dans le contenu : avec cette information le serveur sera certain d'accepter tout ce que vous avez envoyé.

PUT

Vous utiliserez PUT beaucoup moins souvent que POST mais regardons tout de même cette méthode. POST, comme vous avez pu le constater, envoie des données au serveur et le serveur lance une action après la requête même si cette action ne se passe pas forcément au niveau de l'URI précisé par le client : c'est le serveur qui interprète la requête et qui décide la suite. Contrairement à POST, PUT indique que seul l'URI précisé par le client doit être affecté, point final.

Par exemple, disons que vous voulez créer un nouvel utilisateur qui n'existe pas dans une base de données d'un site. Normalement vous le feriez avec POST :

```
POST /users HTTP/1.1
name=jessica
```

Cette requête permettrait au serveur de décider de créer un nouvel utilisateur et de lui donner un ID correct. Avec PUT, cela ressemblerait à la requête suivante :

```
PUT /users/10 HTTP.1.1
name=jessica
```

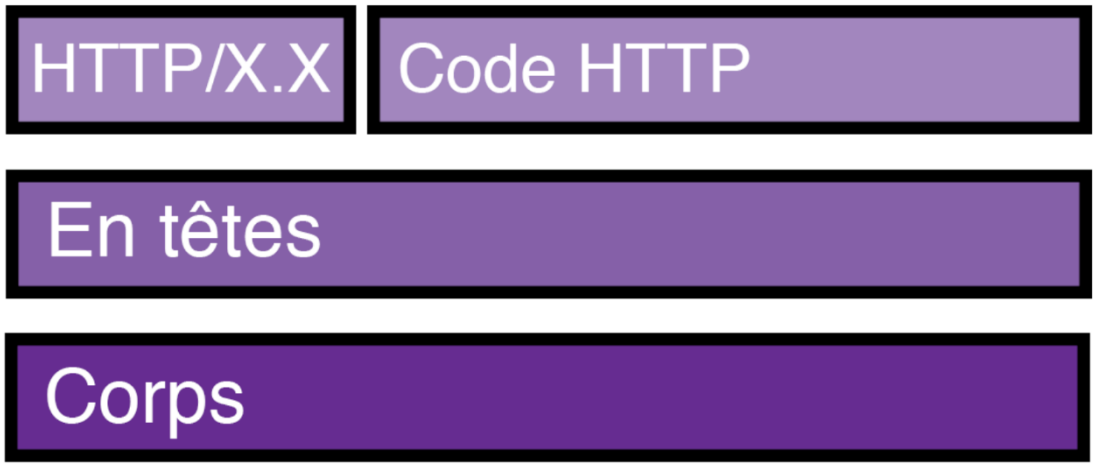
Cette requête dit deux choses : elle demande de créer un nouvel utilisateur à l'ID 10 s'il n'y en a pas ou de remplacer l'utilisateur à cet ID ! Ce n'est pas une bonne idée. Il vaut mieux en général laisser le serveur décider quel ID il faut donner à un nouvel objet et quoi faire avec. Comme vous êtes le client dans ce cas, ce n'est pas vraiment vous qui savez mieux, surtout si vous travaillez avec une API REST externe.

DELETE

Cette méthode est assez claire ! En utilisant DELETE vous dites que vous voulez supprimer la ressource donnée dans l'URI. Par contre, vous ne pourrez pas être sûr que l'action s'est vraiment passée, car la réponse du serveur, quelle qu'elle soit, n'est pas une véritable confirmation de suppression.

Les réponses

La communication HTTP ne se déroule pas que dans un seul sens ! Les réponses d'une API sont d'ailleurs la principale raison qui vous pousse à utiliser une API. Les réponses HTTP sont au format suivant :



Format d'une réponse HTTP
Vous remarquez peut-être des **similarités** entre ce format de réponse HTTP et le format de requête HTTP vu dans le chapitre Les requêtes. Bien joué ! Il y a des éléments spécifiques pour ces deux éléments de communication HTTP, mais les deux formats ne sont pas très différents.

La version HTTP

Indique la version d'HTTP utilisée.

Le code réponse HTTP

Indique l'état de la réponse. La ressource a-t-elle bien été trouvée ? Si non, pourquoi ? Ce code de 3 chiffres vous dira tout ce qu'il faut savoir. Il y a à peu près 30 codes possibles qui couvrent plein de situations de "Tout va bien !" à "Mince, y'a un truc qui va pas !" Dans le prochain chapitre, vous allez apprendre les codes que vous verrez le plus souvent, mais pour l'instant gardez simplement en tête que vous en connaissez déjà quelques-uns (comme la fameuse erreur "404 page non trouvée").

Les en-têtes

Ici vous trouverez des infos classiques (la date, etc.) comme vous l'avez vu dans le chapitre sur les requêtes. Dans les en-têtes réponses spécifiquement, vous trouverez aussi les infos sur le serveur lui-même comme son type ou sa localisation !

Le corps

Le corps de la réponse contient des données cruciales pour les utilisateurs des API REST. Quand vous regardez un site web sur votre ordinateur, le corps d'une réponse est normalement en HTML pour que vous puissiez voir le site mis en forme. Par contre, dans une situation API, quand deux applications communiquent entre elles, le corps de la réponse est souvent en forme de JSON avec des paires key:value. Eh oui, les applications n'ont pas besoin de la représentation visuelle ! Du coup, l'application client peut

récupérer les paires key:value et décider quoi faire avec : soit les assigner aux variables, soit les traiter d'une autre manière.

Ici un exemple de réponse HTTP. Prenez quelques minutes pour comprendre chaque ligne en considérant les explications dans ce chapitre !

```
HTTP/1.1 200 OK

Content-Language: en

X-Ratelimit-Limit: 5000

Vary: Cookie, Accept-Language

Date: Thu, 21 Jan 2016 14:19:31 GMT

Content-Length: 67

Expires: Sat, 01 Jan 2000 00:00:00 GMT

X-Ratelimit-Remaining: 4998

Connection: keep-alive

Content-Type: application/json; charset=utf-8

Cache-Control: private, no-cache, no-store, must-revalidate

Pragma: no-cache

{



  "nom" : "Fluffy",

  "animal" : "chat",

  "couleur" : "noir",

  "goûts" : {

     "jouets" : [

        {

           "nom" : "balle",
```

```
        }

    ]

}
```

}

Les codes HTTP

Quand vous accédez une **ressource** (soit par API soit dans un explorateur normal) vous recevez un code numérique avec un message. Vous ne le voyez pas d'une façon évidente tout le temps dans votre navigateur quand vous naviguez sur le web mais vous le verrez toujours quand vous utilisez une API ! Comme vous avez vu dans le chapitre précédent, le serveur envoie ce code pour dire oui, non, ou autre chose à une requête. Dans un monde idéal, chaque requête serait parfaite et aurait une réponse parfaite du serveur mais ce n'est pas le cas ! Le code HTTP permet le client d'interpréter la réponse du serveur et de réagir de façon appropriée.

Les codes HTTP sont toujours trois chiffres et sont catégorisés en fonction du chiffre des centaines. Un code de la forme :

- **2xx** indique le succès.
- **3xx** redirige le client ailleurs.
- **4xx** indique une faute de la part du client.
- **5xx** indique une erreur de la part du serveur.

Vous verrez certains codes HTTP plus que les autres en travaillant avec les API REST en particulier. Ici les plus fréquents.

2xx :

200 OK

Votre requête a bien été comprise avec une bonne réponse du serveur. Pas de soucis ! 

201 Created

Une ressource a bien été créée. Comme les ressources sont créées avec POST ou PUT un code 201 est l'idéal après avoir envoyé une requête avec une de ces méthodes.

3xx :

301 Moved Permanently

La ressource désirée a déménagé. Pour la trouver vous pouvez peut-être lire la documentation de l'API ou regarder si le nouvel endroit est précisé dans la réponse du serveur.

4xx :

400 Bad Request

La requête n'était pas correcte d'une manière ou d'une autre, souvent à cause des données mal structurées dans les corps des requêtes POST et PUT (des requêtes qui ont souvent des informations dans leurs corps).

401 Unauthorized

Le client n'est pas autorisé à avoir une réponse à la requête qu'il a envoyé. C'est une erreur que vous allez voir tout le temps quand vous travaillerez avec les API qui ont des strictes règles d'autorisation (par exemple, il faut être connecté avec un compte pour accéder au service).

404 Not Found

La ressource n'a pas été trouvée. Vous verrez cette page dans votre navigateur quand vous tentez d'aller sur une page web qui n'existe pas. Une application reçoit le même code réponse quand elle visite une ressource qui n'existe pas (et ça reste aussi frustrant).

5xx :

500 Internal Server Error

Il y a un problème avec le serveur et la requête a planté. Zut ! 😕

Selon le code HTTP reçu, l'application client peut décider quoi faire après. Par exemple si un serveur répond avec un code 500, le client pourra pas assigner les données anticipées à une variable. Par contre avec un code 200, le client saura que la réponse était bonne et qu'il pourra procéder à l'interprétation des informations reçues !

Vous pouvez en apprendre plus sur le site de W3C.