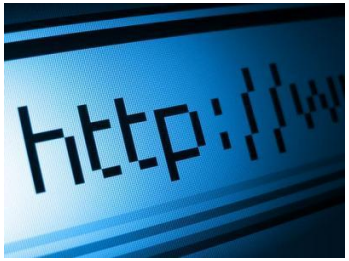
Chapitre 4
Interactions serveur

Plan

- Protocole HTTP
- REST
- Restful API
- AJAX

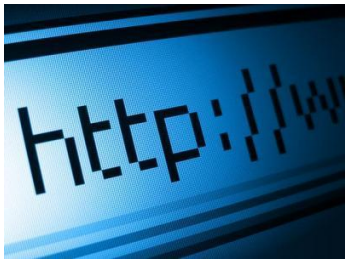
Protocole HTTP

- Hyper Text Transfer Protocol : protocole de la couche Application , créé en 1990



- Standard(s) actuel(s) : **HTTP 1.1, 2.0**
- Variante : HTTPS : Utilise le SSL ou TLS pour la communication sécurisée (nous en reparlerons après la relâche)
- Par défaut, utilise le port **80**, ou **443** pour HTTPS
- Les clients HTTP se connectent à des serveurs HTTP tels que Apache ou IIS.

Protocole HTTP



Pourquoi étudier le protocole **HTTP**?

⇒ Protocole **par défaut** utilisé sur les internet.

⇒ Toute communication impliquant un client/serveur passe par le HTTP.

Donc important de connaître et comprendre ce que ça fait!

HTTP **2.0** en cours de développement:

Voir http://http2.github.io/

Requête HTTP

Ligne de commande (Commande, URL, Version de protocole)
- En-tête de requête
- [Ligne vide]
- Corps de requête

Exemple typique :

Requête HTTP

Host : Site web concerné par la requête.

Referer : Indique l'adresse du document qui a donné lien vers la ressource demandée.

User-Agent : Logiciel utilisé pour la requête. Peut être un navigateur ou un robot.

Autres entêtes: Connection, Accept, Accept-Charset, Accept-Language, Transfer-Encoding, Trailer….

Requête HTTP

Un header très important : le **Content-Type**.

Sert à spécifier le **MIME type** du contenu de votre requête.

Exemple :

Content-Type : application/json

Le client doit indiquer quel genre de contenu il envoie au serveur afin que les données se rendent sous un format acceptable.

Réponse HTTP

Ligne de statut (Version, Code-réponse, Texte-réponse)
- En-tête de réponse
- [Ligne vide]
- Corps de réponse

Exemple typique :

Date: Fri, 31 Dec 1999 23:59:59 GMT
Server: Apache/0.8.4
Content-Type: text/html
Content-Length: 59
Expires: Sat, 01 Jan 2000 00:59:59 GMT
Last-modified: Fri, 09 Aug 1996 14:21:40 GMT

Réponse HTTP

Date : Moment auquel le message est généré.

Server : Modèle du serveur répondant à la requête.

Content-Type : Type MIME de la ressource obtenue

Ex : text/html, application/xml, application/json, etc.

Content-Length : Taille en octets de la ressource

Expires : Moment où la ressource devient obsolète : pratique pour la mémoire cache

Last-Modified : Date de dernière modification

et autres...

Primitives HTTP

- **GET** :

  Méthode permettant d'**obtenir** une ressource. Requête sans effet sur la ressource : doit pouvoir être répétée, **idempotente**.

- **HEAD** :

  Semblable a GET, mais demande des **informations** sur la ressource au lieu de demander la ressource elle-même.

- **POST** :

  Permet typiquement de **soumettre** une nouvelle ressource. Sert également de primitive pour tout ce qui *action* sur une ressource.

Primitives HTTP

- **OPTIONS** :
  Obtient les **options** de communication d'une ressource ou du serveur
- **CONNECT** :
  Permet d'utiliser un **proxy** ou tunnel.
- **TRACE** :
  Demande au serveur de retourner ce qu'il a reçu, à des fins de **débogage**.
- **PUT** :
  Permet de **remplacer** ou **d'ajouter** une ressource sur le serveur.

Primitives HTTP

- **PATCH** :

  Permet la **modification partielle** d'une ressource.

- **DELETE** :

  Permet de **supprimer** une ressource du serveur.

Primitives HTTP

POST - PUT - PATCH

POST: Création de ressource

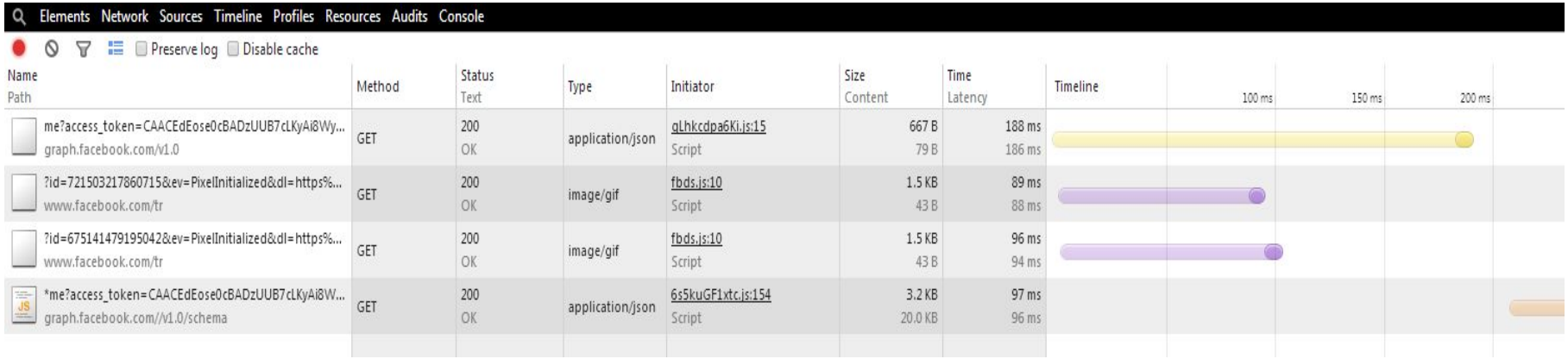
PUT: Modification complète d'une ressource existante ou création

PATCH: Modification partielle d'une ressource existante

Un bon API utilise les primitives HTTP de manière appropriée: GET, PUT, POST, DELETE etc.

En pratique

Onglet Network de la console Chrome



En pratique

Postman:

https://www.getpostman.com/

Outil de référence pour développement web. Importance de tester vos appels HTTP (REST) *avant* de coder quoique ce soit...

Status HTTP

- Les **codes d'erreur** sont très pratiques pour diagnostiquer le bobo lors d'une requête qui n'aboutit pas ou qui retourne le mauvais résultat.
- Les codes d'erreur sont répartis dans des **intervalles**, qui elles sont divisées en catégories

1xx : Information 

2xx : Succès 

3xx : Redirection 

4xx : Erreur Client 

5xx : Erreur Serveur 

Une bonne API doit se conformer au standard et retourner des statuts cohérents.

Status HTTP

200 OK : La requête s'est effectuée avec **succès**

GET : La page désirée a bien été retournée
POST : La requête contient le résultat de l'action postée, peut-être du contenu HTML, une ressource, ou rien du tout (204 dans ce cas)

302 FOUND : La ressource a été trouvée, mais après une redirection.

Dans la réponse, l'entête **'Location'** doit normalement indiquer le nouvel URL de la ressource.

400 BAD REQUEST :

Requête mal formée, rejetée par le serveur.

Status HTTP

401 : UNAUTHORIZED

L'accès à la ressource nécessite une **authentification** qui n'a pas réussie. Par exemple, l'usager n'est pas valide ou a fourni un mauvais mot de passe.

403 : FORBIDDEN

Le serveur a bien reçu la requête, mais refuse de l'interpréter car l'usager n'a pas les permissions nécessaires.

404 : NOT FOUND

Le serveur ne trouve rien correspondant à l'URL de la requête.

Status HTTP

500 : INTERNAL SERVER ERROR

Le serveur a fait une erreur en essayant de remplir la requête (typiquement une exception)

503 : SERVER UNAVAILABLE

Le serveur ne peut répondre à la requête, par exemple suite à un volume trop lourd ou une maintenance planifiée...

Pour les autres erreurs, il y a la documentation...
http://www.restapitutorial.com/httpstatuscodes.html
418 !!

REST (Representational State Transfer)

Le **REST** est devenu le style architectural de référence pour toute application web moderne.

- Ce n'est pas un **protocole**, mais bien **un style architectural**.
- Il s'agit simplement d'une façon de structurer son API en suivant un certain standard.
- Utilise toutes les notions d'HTTP vues précédemment.
- Très, très, très flexible...

REST

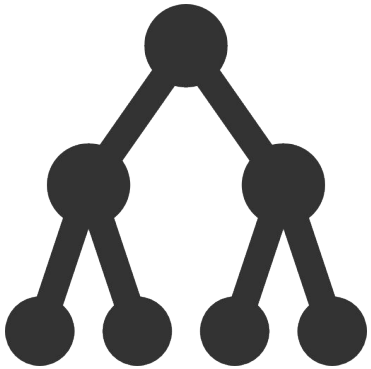
CONCEPT :

Le REST repose sur l'image qu'on se fait d'une application Web : l'usager progresse à travers des **liens** qui pointent sur des ***ressources***. L'accès à un lien se traduit à une **requête** HTTP:

GET, POST, DELETE, etc .

→ Le REST est donc plutôt **invisible**!

→ Le REST est **hiérarchique**!



REST

CONCEPT :

REST repose sur l'existence de ***ressources***.

→ Chaque ressource est référencée par un identifiant unique, soit son URI.

Une ressource est souvent un simple bout de code s'exécutant lors de l'appel de l'URI en question.

Les données échangées peuvent prendre plusieurs formes: JSON, HTML, images etc.

REST

Exemple :

L'accès à http://ubeat.herokuapp.com/unsecure/users

Exécute du **code** et retourne une **ressource** reliée à cet URL.

Il est important de comprendre que le code peut être littéralement n'importe quoi, et retourner un format complètement arbitraire.

Ceci dit, UBeat est un **JSON** Rest API - donc retourne strictement ce format (plus certains fichiers).

RESTful API

Lorsqu'on parle de RESTful API, on parle donc du style architectural REST, appliqué à une API.

- Respecte les primitives HTTP vues.
- Les données échangées seront traditionnellement sous la forme de JSON. Puisqu'il s'agit d'une API, pas de HTML échangé…
- Destinée à être consommée par d'autres applications ou par un navigateur.

RESTful API

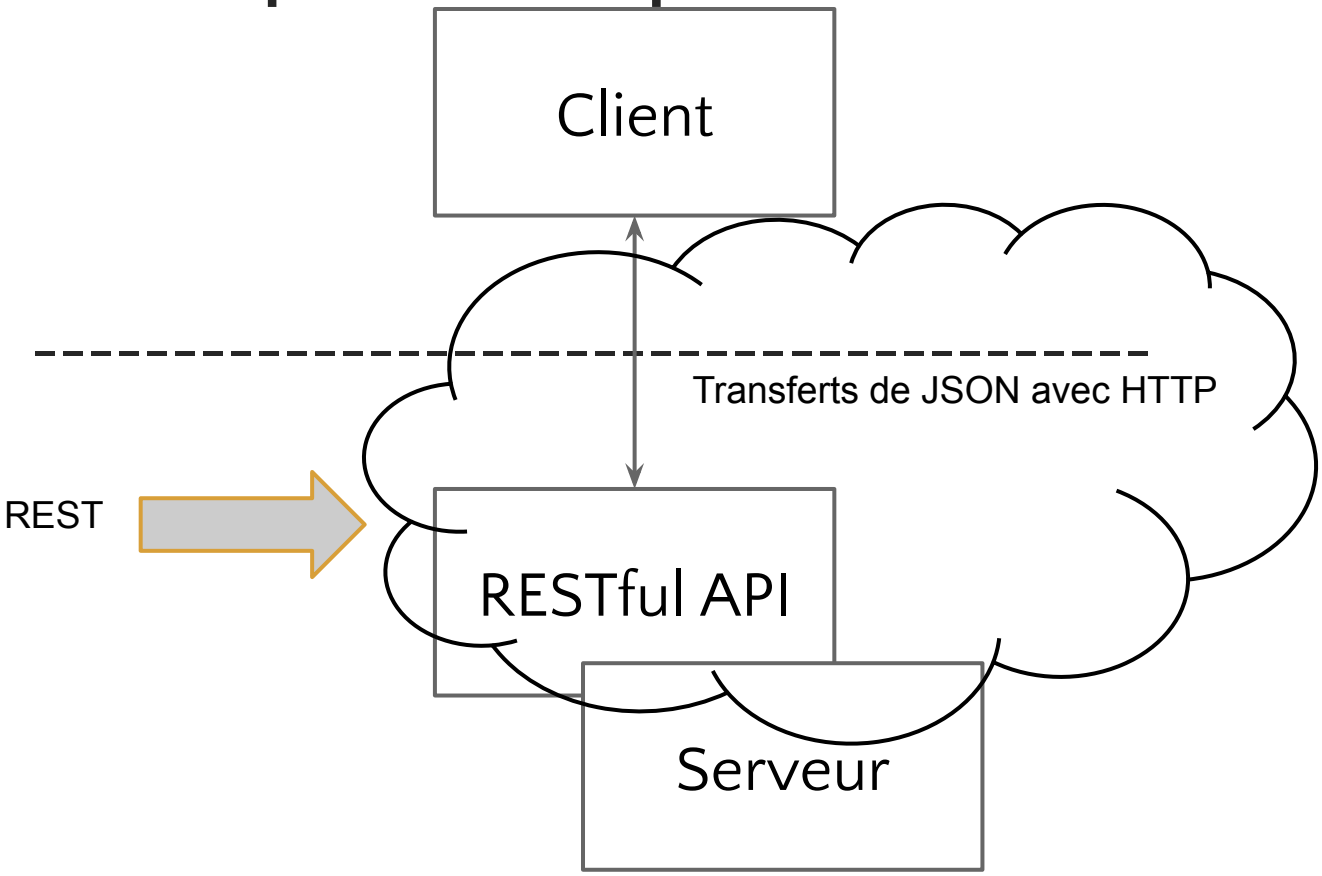
IMPORTANT

- **Destinée à être consommée par d'autres applications ou par un navigateur.**

Le design d'un API RESTful est aussi important que le code. Les méthodes doivent être claires, respecter les standards, exposer des statuts HTTP cohérents etc. Puisqu'il ne s'agit que de JSON, vous ne pouvez prévoir qui sera votre utilisateur!

RESTful API

! Très important à comprendre !



RESTful API

URL	**GET**	**POST**
Collection : http://www.google.com/resources/	**Liste** les URI des différents éléments dans la collection.	**Crée** une nouvelle entrée dans la collection. Le nouvel URI est automatiquement assigné.
Élément: http://www.google.com/resources/item1	**Retourne** une certaine représentation de l'élément.	Généralement **non utilisé.**

RESTful API

URL	**PUT**	**DELETE**
 **Collection** : http://www.google.com/resources/	**Remplace** la collection par une nouvelle collection.	**Supprime** la collection.
 **Élément**: http://www.google.com/resources/item1	**Remplace** un élément de la collection ou le **crée** s'il n'existe pas.	**Supprime** l'élément de la collection.

RESTful API

Exemple d'une **bonne** ressource:

http://ubeat.herokuapp.com/unsecure/users/userId

Exemple d'une **mauvaise** ressource:

http://ubeat.herokuapp.com/unsecure/users?id=userId

Les identifiants de ressource doivent former un ensemble **hiérarchique cohérent.** Les ***query strings*** ne devraient servir qu'à filtrer, et non pas impacter quel appel est fait sur le serveur.

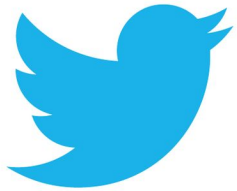
Exemple typique:

http://ubeat.herokuapp.com/unsecure/users?name=Vincent

Exemples

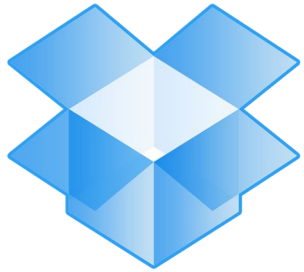
Twitter REST API:

https://dev.twitter.com/docs/api



Dropbox Core API:

https://www.dropbox.com/developers/core/docs



Slack API:

https://api.slack.com/web

Exemples

Any-Api

https://any-api.com/

Public API

https://github.com/public-apis/public-apis

Pourrait être pratique pour votre projet de session...

CORS (Cross-Origin Resource Sharing)

Le **CORS** est un mécanisme permettant d'obtenir/envoyer des ressources entre des **domaines différents**.

Pourquoi s'en soucier?

⇒ Par défaut, vous ne pouvez pas envoyer de requêtes à partir de votre navigateur sur un serveur **sur un différent port** et/ou **url**.

Votre serveur, qui roule par exemple sur **localhost:8080/**, n'a pas la même origine que votre client!

Ce mécanisme est imposé par votre **navigateur afin** d'éviter des **failles de sécurité.**

CORS (Cross-Origin Resource Sharing)

Comment s'en sortir?

⇒ Ajouter **un header** sur les requêtes HTTP afin de modifier les restrictions d'origines!

```
Access-Control-Allow-Origin = "*"
```

Peut-être que mettre * n'est pas la technique la plus **sécuritaire**…

```
Autres headers pertinents :
Access-Control-Allow-Methods = GET, POST
Access-Control-Allow-Credentials = true
```

Voir : https://developer.mozilla.org/fr/docs/HTTP/Access_control_CORS

AJAX

Qu'est-ce que l'**AJAX**?

- Simplement, il s’agit du nom donné à la façon d’interagir avec le serveur en JavaScript.

- X stands for XML, ce qui n’est évidemment plus d’actualité… mais le nom demeure.

AJAX

L'**AJAX** représente l'intégration pratique des différents concepts vus jusqu'à présent.

- Une façon simple d'interagir avec un API en JavaScript de manière asynchrone.

- Toute application web moderne est basée sur l'AJAX. C'est ce qui permet de rafraîchir différentes parties d'une page de manière indépendante...

AJAX

Traditionnellement, l'**AJAX** était basé sur l'objet **XMLHttpRequest** en JavaScript. Malgré son nom, il s'agit encore de la référence utilisée en ES5.

C'est ce qui permet de faire des requêtes au serveur et d'handler la réponse de manière asynchrone.

https://developer.mozilla.org/en/docs/Web/API/XMLHttpRequest

Promise

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Promise

- Un objet dont la valeur n'est pas connue _encore_ et sera déterminée de manière asynchrone.
- On dit une Promise comme **resolved** lorsque l'appel serveur est exécuté avec succès, ou **rejected** si l'appel échoue.
- Peu importe la syntaxe, il est toujours important de traiter les 2 cas possibles
  - Toujours assumer que vos requêtes serveur peuvent **échouer!**

Promise

```
const promise = new Promise((resolve, reject) => {
  const request = new XMLHttpRequest();

  request.open('GET', 'https://api.icndb.com/jokes/random');
  request.onload = () => {
    if (request.status === 200) {
      resolve(request.response);
    } else {
      reject(Error(request.statusText));
    }
  };

  request.onerror = () => {
    reject(Error('Error fetching data.'));
  };

  request.send();
});

promise.then((data) => {
  document.body.textContent = JSON.parse(data).value.joke;
}, (error) => {
  console.log(error.message);
});
```

Async/Await

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/async_function

- La syntaxe **async/await** permet de faire des promises de manière plus élégante.
- Introduite dans la même veine que les changements ES6.
- Permet de ne pas tomber dans ce qui est souvent référencé comme le *callback hell*.

Async/Await

```
function doRequest(options) {
  return new Promise ((resolve, reject) => {
    const req = http.request(options);
    req.on('response', res => { resolve(res); });
    req.on('error', err => { reject(err); });
  });
}

try {
   const res = await doRequest(options);
} catch (err) {
   console.log('some error occurred...');
}
```

AJAX

Fetch API

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

Enfin, XMLHttpRequest remplacé par **Request** et **Response**, en plus d'ajouter quelques nouvelles fonctionnalités.

N'est évidemment compatible qu'avec les plus récents navigateurs.

AJAX

Fetch API

```
var myHeaders = new Headers();

var myInit = { method: 'GET',
               headers: myHeaders,
               mode: 'cors',
               cache: 'default' };

fetch('flowers.jpg', myInit).then(function(response) {
  return response.blob();
}).then(function(myBlob) {
  var objectURL = URL.createObjectURL(myBlob);
  myImage.src = objectURL;
});
```

Exemples complets:

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch
https://scotch.io/tutorials/how-to-use-the-javascript-fetch-api-to-get-data

AJAX

Axios

https://github.com/axios/axios

Librairie largement utilisée pour toute ce qui est requête HTTP - souvent reconnue comme plus simple et avec une interface plus pratique que fetch.

Fonctionne aussi bien en Promise qu'avec async await.