

JavaScript

JavaScript

A yellow square containing the letters 'JS' in a bold, black, sans-serif font.

- Historiquement permet de programmer des interactions au sein des navigateurs
 - **Interagir** : savoir qu'un bouton a été cliqué
 - **Afficher** : manipuler la page web pour rendre visible des nouvelles parties
 - **Communiquer** : envoyer ou recevoir des requêtes
- **C'est le seul langage disponible côté navigateur**
- Mais est aussi disponible côté serveur via NodeJS

Exemple

```
<!doctype html>
<html lang="fr">
  <head>
    <title>Exemple</title>
    <script>
      var msg = "hello";
      alert(msg);
    </script>
  </head>
  <body>
  </body>
</html>
```



Best practice

```
<!doctype html>  
<html lang="fr">  
  <head>  
    <title>Example</title>  
    <script src="script.js"></script>  
  </head>  
  <body>  
  </body>  
</html>
```



Messages de log

L'instruction qui permet d'afficher des messages de log en JavaScript est :

script.js

```
console.log('Hello, world!');
```

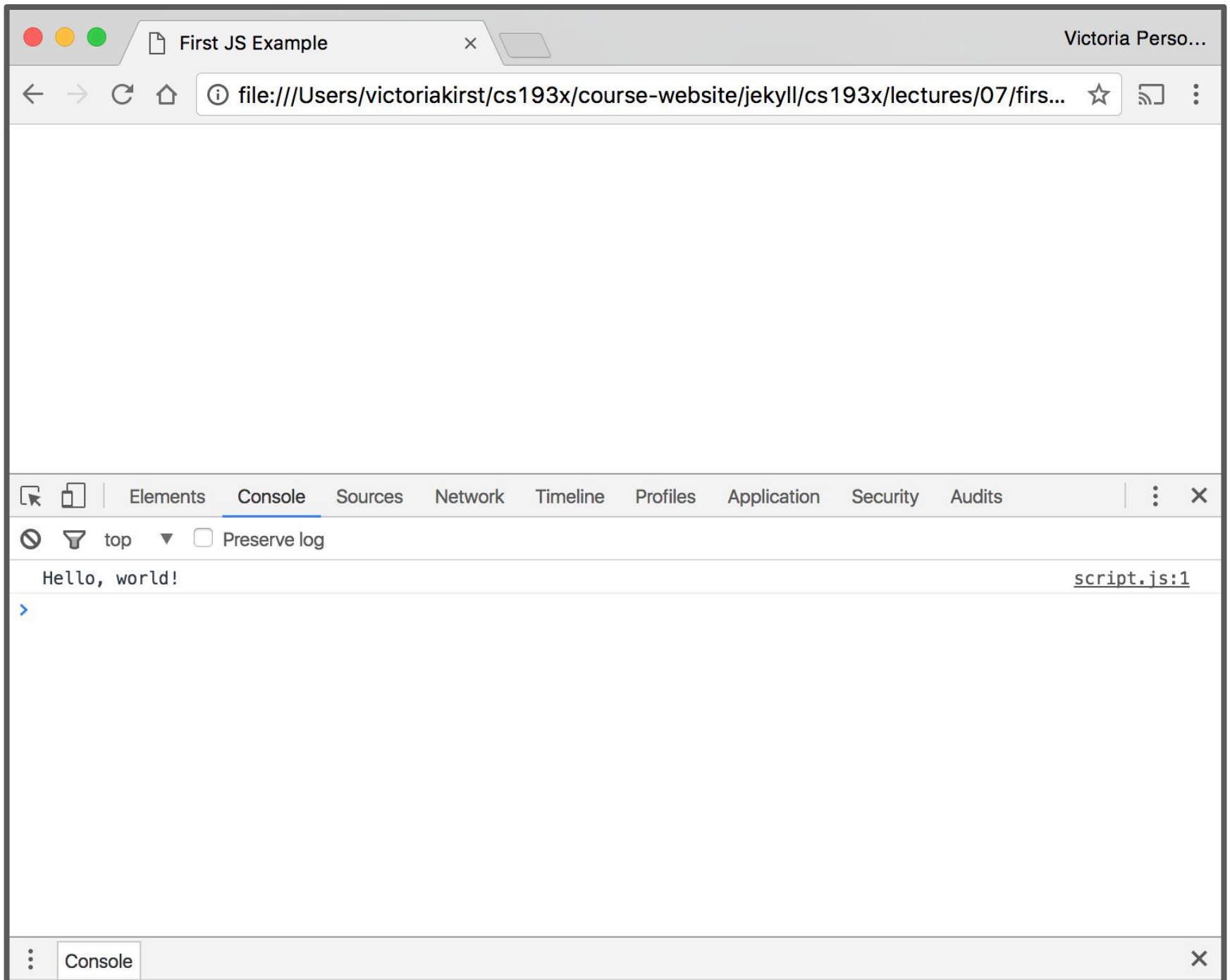
Exécution de JavaScript

Il n'y a pas de "**main method**"

- Le script est exécuté de haut en bas

Il n'y a pas de **compilation** par le développeur

- JavaScript est compilé et exécuté à la volée par le navigateur



Similarités avec Java, C, ...

for-loops:

```
for (let i = 0; i < 5; i++) { ... }
```

while-loops:

```
while (notFinished) { ... }
```

comments:

```
// comment or /* comment */
```

conditionals (if statements):

```
if (...) {  
    ...  
} else {  
    ...  
}
```


Fonctions

La syntaxe suivante est une des manières de définir une fonction en JavaScript

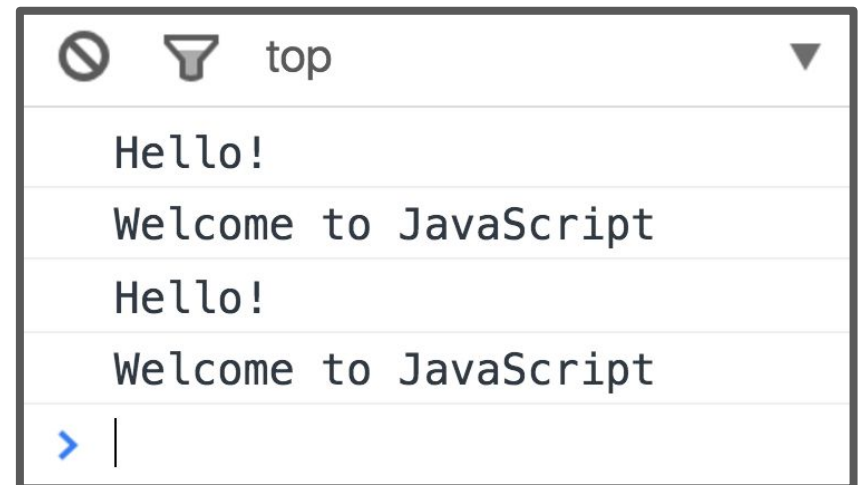
```
function name() {  
    statement;  
    statement;  
    ...  
}
```

Exemple

```
hello();  
hello();  
  
function hello() {  
  console.log('Hello!');  
  console.log('Welcome to JavaScript');  
}
```

Cela fonctionne car les déclarations de fonctions sont ***hoisted*** : déplacées au sommet du scope dans lesquelles elles sont définies

il faut éviter de se reposer sur ce mécanisme



Variables

Trois façons de déclarer des variables en JS

```
// Function scope variable  
var x = 15;  
// Block scope variable  
let fruit = 'banana';  
// Block scope constant; cannot be reassigned  
const isHungry = true;
```

Le langage est dynamiquement typé

Paramètres de fonctions

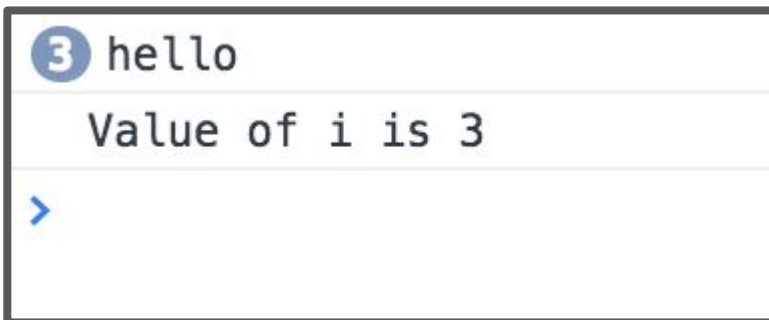
```
function printMessage(message, times) {  
  for (var i = 0; i < times; i++) {  
    console.log(message);  
  }  
}
```

Les paramètres de fonctions ne sont pas déclarés à l'aide de `let`, `const` ou `var`

Comprendre var

```
function printMessage(message, times) {  
  for (var i = 0; i < times; i++) {  
    console.log(message);  
  }  
  console.log('Value of i is ' + i);  
}
```

```
printMessage('hello', 3);
```



A screenshot of a JavaScript console window. It shows three lines of output: a blue circle with the number 3 followed by the text 'hello', the text 'Value of i is 3', and a blue prompt character '>' on the third line.

```
3 hello  
Value of i is 3  
>
```

La valeur de `i` est accessible hors de la boucle `for` car `var` déclare des variables avec un scope *fonction*

Comprendre let

```
function printMessage(message, times) {  
  for (let i = 0; i < times; i++) {  
    console.log(message);  
  }  
  console.log('Value of i is ' + i);  
}
```

```
printMessage('hello', 3);
```

3 hello

✖ ▶ Uncaught ReferenceError: i is not defined
at printMessage (script.js:5)
at script.js:8

>

La valeur de `i` n'est pas accessible hors de la boucle `for` car `let` déclare des variables avec un scope *block*

Comprendre const

```
const y = 10;  
y = 0;           // error!  
y++;            // error!  
const list = [1, 2, 3];  
list.push(4);    // OK
```

Les variables déclarées avec `const` ne peuvent pas être réaffectées

Cependant il reste possible de modifier l'objet sous jacent

- Ressemble au `final` de Java

Bonnes pratiques

- Utiliser `const` partout où vous pouvez
- Si vous avez besoin d'une variable réaffectable, utilisez `let`
- **N'utilisez pas `var`.**
 - `const` et `let` sont maintenant bien supportés par les browsers

Types

Les **variables** JS n'ont pas de types, mais leurs **valeurs** si

Il y a plusieurs types primitifs:

- **Boolean** : true et false
- **Number** : tout est de type double (pas d'entiers)
- **String**: avec 'single' ou "double-quotes"
- **Null**: null une valeur qui signifie “ceci n’a pas de valeur”
- **Undefined**: la valeur d’une variable non affectée

Il y a aussi les types Object, comme Array, Date, et même Function!

Transtypage booléen

Les valeurs non booléennes peuvent être utilisées dans les instructions de contrôle, elles sont converties en "truthy" ou "falsy"

- `null`, `undefined`, `0`, `NaN`, `' '` évaluent à `false`
- Les autres valeurs évaluent à `true`

```
if (username) {  
    // username is defined  
}
```

Egalité

`==` et `!=` font une conversion implicite de type avant comparaison

```
' ' == '0' // false
' ' == 0   // true
0  == '0'  // true
NaN == NaN // false
[ ' ' ] == ' ' // true
false == undefined // false
false == null // false
null == undefined // true
```

Opérateurs === et !==

=== et !== sont les véritables opérateurs de comparaison,
toujours les utiliser!

```
' ' === '0'    // false
' ' === 0      // false
0 === '0'     // false
NaN === NaN    // still weirdly false
[ ' ' ] === ' ' // false
false === undefined // false
false === null  // false
null === undefined // false
```

null et undefined

Quelle différence?

- `null` est la valeur représentant l'absence de valeur (comme `null` en Java)
- `undefined` est la valeur d'une variable n'ayant pas reçu de valeur

```
let x = null;  
let y;  
console.log(x);  
console.log(y);
```

`null`

`undefined`



Tableaux

Les tableaux sont les objets pour définir des listes

```
// Creates an empty list
const list = [];
const groceries = ['milk', 'cocoa puffs'];
groceries[1] = 'kix';

// For each loop
for (let item of groceries) {
  console.log(item);
}
```

Objets

Collection de paires clé - valeur, peut être utilisé comme une hashmap

```
const prices = {};  
const scores = {  
  peach: 100,  
  mario: 88,  
  luigi: 91  
};  
console.log(scores['peach']);    // 100  
console.log(scores.peach);      // 100  
scores.peach = 20;  
console.log(scores.peach);      // 20
```

Itérer les propriétés d'un objet

Il est possible d'itérer sur les propriétés d'un objet

```
const scores = {  
  peach: 100,  
  mario: 88,  
  luigi: 91  
};  
  
for (let name in scores) {  
  console.log(name + ':' + scores[name]);  
}
```


Fonctions de premier ordre

En JavaScript, les fonctions sont des objets comme les autres

```
var add = function(a, b) {  
    return a + b;  
}
```

```
add(2, 2); // 4
```

Ajout d'une fonction sur un objet

On peut ajouter une fonction dans un objet, `this` désigne l'objet receveur

```
const scores = {  
  peach: 100,  
  mario: 88,  
  luigi: 91,  
  total: function() {  
    return this.peach + this.mario + this.luigi;  
  }  
};
```

Classes ES6

```
class Person {  
  constructor(name, age) {  
    this.name = name;  
    this.age = age;  
  }  
  
  displayAge() {  
    console.log(this.age);  
  }  
  displayName() {  
    console.log(this.age);  
  }  
  displayAgeAndName() {  
    this.displayAge();  
    this.displayName();  
  }  
}  
  
const p = new Person("Joe", 1);
```

- Le constructeur est optionnel
- Tous les attributs et les méthodes sont publics
- Tous les appels de méthodes et les accès d'attributs se font via `this`

Callbacks

Les fonctions peuvent être passées en paramètres d'autres fonctions

```
const groceries = ['milk', 'cocoa puffs'];
```

```
// For each loop  
for (let item of groceries)  
  console.log(item);
```

```
// Callback style  
groceries.forEach(function(item) {  
  console.log(item);  
});
```

```
// Nerdy callback style (use that!)  
groceries.forEach(item => {  
  console.log(item);  
});
```

Mais que fait forEach?

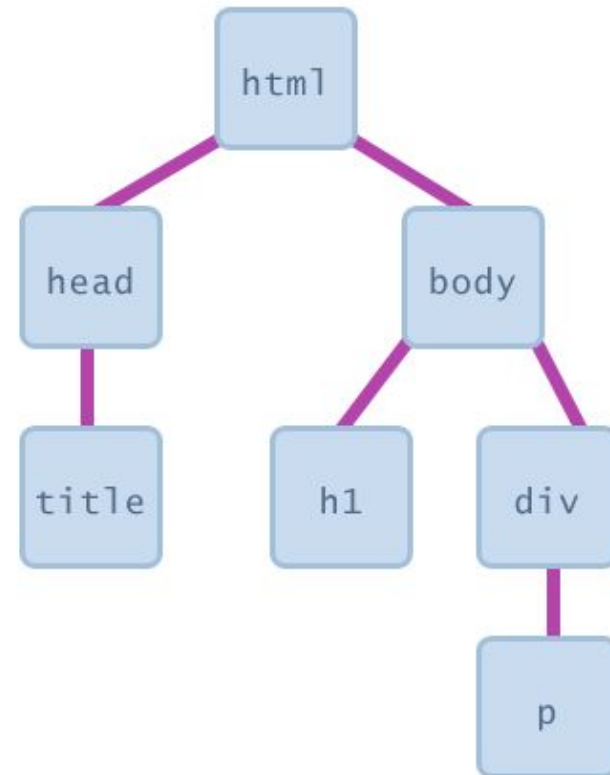
```
class Array {  
  forEach(callback) {  
    for (let i = 0; i < this.length; i++) {  
      callback(this[i], i, this);  
    }  
  }  
}
```

forEach itère sur tous les éléments du tableau, pour chaque élément, elle applique la fonction callback (passée en paramètre de forEach) en lui fournissant en paramètre l'élément courant **forEach calls callback back!**

DOM

On accède aux éléments composant la page web en JavaScript au travers du **Document Object Model**

```
<html>
  <head>
    <title></title>
  </head>
  <body>
    <h1></h1>
    <div>
      <p></p>
    </div>
  </body>
</html>
```



DOM et JavaScript

JavaScript peut :

- **examiner** les noeuds du DOM pour en inspecter l'état (ex. voir le texte saisi par un utilisateur)
- **editer** les attributs des noeuds du DOM (ex. changer le style d'un élément <h1>)
- **ajouter ou supprimer** des noeuds du DOM (ex. ajouter un texte de statut quelque part)

Accéder aux noeuds du DOM

L'accès aux noeuds du DOM se fait via la fonction `querySelector`:

```
document.querySelector('css selector');
```

- Retourne le **premier** noeud qui matche la règle CSS

```
document.querySelectorAll('css selector');
```

- Retourne **tous** les éléments qui matchent la règle CSS

Quelques propriétés des noeuds du DOM

Property	Description
id	la valeur de l'attribut id de l'élément
innerHTML	le HTML entre le noeud ouvrant et fermant de l'élément vu comme une chaîne de caractères
textContent	Le contenu texte d'un noeud ainsi que celui de ses descendants
classList	Un objet contenant toute la liste des classes dans lesquelles l'élément se situe

Créer des éléments dans le DOM

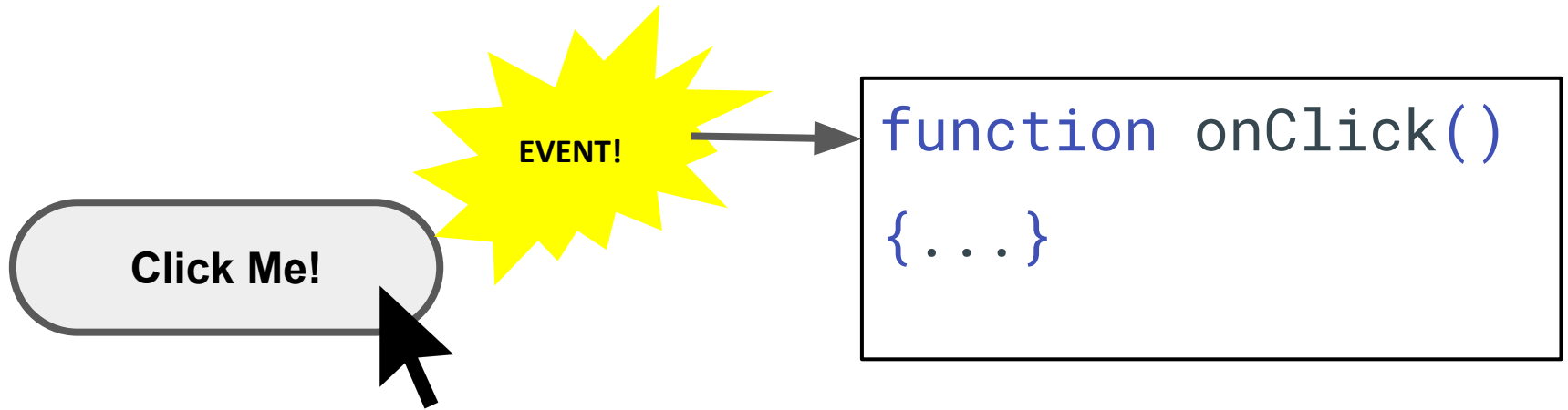
Il est possible de créer des éléments dynamiquement via `createElement` et `appendChild`:

```
document.createElement(tag string)  
    element.appendChild(element);
```

On peut supprimer des éléments via `remove`

```
element.remove();
```

Évènements



Exemple : un élément de page avec lequel on peut interagir. Quand on clique sur le bouton, un événement est déclenché. L'utilisateur peut enregistrer des callbacks sur les événements de son choix

```
<html>
  ▼<head>
    <meta charset="utf-8">
    <title>First JS Example</title>
    <script src="script.js" defer</script>
  </head>
  ▼<body>
    <button>Click Me!</button>
  </body>
</html>
```

```
function onClick() {
  console.log('clicked');
}

const button = document.querySelector('button');
button.addEventListener('click', onClick);
```

Click Me!



Elements

Console



top

clicked



Envoyer des requêtes

```
// FAKE HYPOTHETICAL API.  
// This is not real a JavaScript function!  
const content = loadFromFile('images.txt');
```

Quelques problèmes avec cette API fictive:

- Ce serait mieux de charger les données de manière **asynchrone** sinon l'appel va être **bloquant**
- Il n'est pas possible de vérifier le statut de la requête.
Quid si la ressource n'existe pas? Si l'on a pas les droits?

L'API Fetch

L'API Fetch comporte une seule fonction, concise et facile à utiliser

```
fetch( 'images.txt' );
```

- fetch prend en paramètre l'URL de la ressource que vous voulez consulter
- Elle retourne une Promise
- **What???**

Les Promise

```
const promise = fetch('images.txt');  
promise.then(onSuccess, onFail);
```

Une promesse peut être dans un des états suivants

- **pending**: état initial, promesse non-exécutée
- **fulfilled**: la promesse a été exécutée correctement
- **rejected**: la promesse a rencontrée une erreur

On attache un gestionnaire à une promesse via `.then()`

```
fetch('images.txt').then(response =>  
console.log(response.status));
```

REST

Restful API

Une API web est dite “**Restful**” si elle a les caractéristiques suivantes:

- Les requêtes sont envoyées sous forme de requêtes HTTP:
 - Méthodes HTTP: GET, POST, PUT, PATCH, DELETE..
- Un **Endpoint**: <http://example.com/api/ressources>
- Les requêtes doivent être envoyées avec un MIME*/Content-type spécifique tel: JSON, XML, HTML, etc.

* Multipurpose Internet Mail Extensions

Contraintes d'une API REST

Une API Restful doit respecter certaines contraintes d'architecture:

1. **Archi client-serveur:** chacun évolue indépendamment.
2. **Serveur sans états (stateless):** pas de sessions.
3. **Utilisation du cache:** le client sait combien de temps il peut garder les données qu'il reçoit avant expiration.
4. **Une interface uniforme:** un seul id, contient l'URL des ressources suivantes..
5. **Archi en couches:** Réduire la complexité de l'architecture globale de l'API.
6. **Code à la demande:** étendre les fonctionnalités du client.

Ressources

En REST tout est resources, on interagit avec les ressources à travers différents endpoints (URLs):

- <http://example.com/products>
- <http://example.com/products/1>
- <http://example.com/products/1/description>

Action sur les ressources

Pour interagir avec une ressource, il suffit d'utiliser la méthode HTTP (verbe) adéquate :

- GET <http://example.com/products/1> (Récupérer)
- POST <http://example.com/products> (Créer)
- PUT <http://example.com/products/1> (Mettre à jour)
- DELETE <http://example.com/products/1> (Effacer)

API Endpoints

Tweets	Retweets	Likes (formerly favorites)
<ul style="list-style-type: none">• POST statuses/update• POST statuses/destroy/:id• GET statuses/show/:id• GET statuses/oembed• GET statuses/lookup	<ul style="list-style-type: none">• POST statuses/retweet/:id• POST statuses/unretweet/:id• GET statuses/retweets/:id• GET statuses/retweets_of_me• GET statuses/retweeters/ids	<ul style="list-style-type: none">• POST favorites/create/:id• POST favorites/destroy/:id• GET favorites/list

API de Twitter

API Endpoints

Resource URL

<https://api.twitter.com/1.1/statuses/update.json>

Resource Information

Response formats	JSON
Requires authentication?	Yes (user context only)
Rate limited?	Yes

Parameters

Name	Required	Description	Default Value	Example
status	required	The text of the status update. URL encode as necessary. t.co link wrapping will affect character counts.		

POST statuses/update

Paramètres des Endpoints

Un endpoint peut accepter un ensemble de paramètres pour effectuer sa tâche. Ces paramètres peuvent être passés de **trois manières** différentes:

- Paramètres dans l'URL.
- Headers dans la requête.
- Corps (body) de la requête.

Paramètres des Endpoints - paramètres dans l'URL

The diagram illustrates the components of a URL using the example `http://www.mywebsite.com/apparel/skirt.php?sku=123&lang=en§=silk`. Brackets above the URL identify five parts: protocol (red), hostname (black), directory (blue), filename (green), and query parameters (magenta). Brackets below the URL identify two parts: domain name (under the hostname) and URI (under the directory, filename, and query parameters).

protocol	hostname	directory	filename	query parameters
http://	www.mywebsite.com	/apparel/	skirt.php	?sku=123&lang=en§=silk

domain name

URI

Paramètres des Endpoints - headers de la requête

▼ Request Headers

```
:authority: www.google.fr
:method: GET
:path: /
:scheme: https
accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
accept-encoding: gzip, deflate, br
accept-language: fr-FR,fr;q=0.9,en-US;q=0.8,en;q=0.7
cache-control: no-cache
pragma: no-cache
upgrade-insecure-requests: 1
user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.162 Safari/537.36
```

Paramètres des Endpoints - corps de la requête

```
1  POST /login HTTP/1.1
2  Host: foo.com
3  Content-Type: application/x-www-form-urlencoded
4  Content-Length: 37
5
6  username=foo@foo.com&password=123_foo
```

Codes d'état HTTP (Status code)

Il existe plus de 40 codes HTTP, chacun ayant une signification bien précise. Voici les grandes catégories:

- **1xx**: Information (e.g. 100 attente de la suite (continue))
- **2xx**: Succès (e.g. 200 tout s'est bien passé (OK))
- **3xx**: Redirection (e.g. 301 document déplacé (redirection))
- **4xx**: Erreur du client web (e.g. 404 document introuvable)
- **5xx**: Erreur du serveur (e.g. 500 Erreur interne)

API Restful et cache (E-tags)

Les E-tags permettent d'implémenter la contrainte de cache pour une API Restful. Il existe également d'autres méthodes qui sont l'utilisation des headers **Expires** et **cache-control**.

```
HTTP/1.1 200 OK
Date: Sat, 09 Feb 2013 16:09:50 GMT
Server: Apache/2.2.22 (Ubuntu)
Last-Modified: Sat, 02 Feb 2013 12:02:47 GMT
ETag: "c0947-b1-4d0258df1f625"
Content-Type: application/json

{
  id: 4,
  item: "take out the trash",
  created: "Sat, 02 Feb 2013 08:29:53 GMT",
  updated: "Sat, 02 Feb 2013 12:02:47 GMT",
}
```

API Restful et cache (E-tags)

Une fois que je connais l'E-tag, je peux à l'avenir demander s'il a changé ou non en envoyant une requête avec comme header:

If-None-Match: c0947-b1-4d0258df1f625

```
HTTP/1.1 304 Not Modified
Date: Sat, 09 Feb 2013 16:09:50 GMT
Server: Apache/2.2.22 (Ubuntu)
Last-Modified: Sat, 02 Feb 2013 12:02:47 GMT
ETag: "c0947-b1-4d0258df1f625"
```

API Restful et cache (E-tags)

```
HTTP/1.1 200 OK
```

```
Date: Sat, 09 Feb 2013 16:29:24 GMT
```

```
Server: Apache/2.2.22 (Ubuntu)
```

```
Last-Modified: Sat, 02 Feb 2013 14:33:21 GMT
```

```
ETag: "c7493-d7-a6b64d37f6cc3" # New ETag!
```

```
Content-Type: application/json
```

```
{
```

```
  id: 4,
```

```
  item: "Take out the trash, TODAY!",
```

```
  created: "Sat, 02 Feb 2013 08:29:53 GMT",
```

```
  updated: "Sat, 02 Feb 2013 14:33:21 GMT",
```

```
}
```

OpenAPI

C'est un ensemble de spécifications permettant de décrire une API REST.

Cette description peut être comprise aisément à la fois par les développeurs ainsi que par les ordinateurs.

L'API est décrite sous forme d'un fichier JSON ou YAML.

Un ensemble d'outils permet de facilement convertir ces fichiers de description en documentation web ou encore en bibliothèques pour directement utiliser l'API décrite sans avoir à l'implémenter.

OpenAPI

```
1  swagger: '2.0'
2  schemes:
3    - https
4  host: api.twitter.com
5  basePath: /1.1
6  info:
7    contact:
8      email: support@twitter.com
9      name: Twitter support
10     url: 'https://dev.twitter.com'
11     x-twitter: twitter
12   title: Twitter
13   version: '1.1'
```


OpenAPI

```
29  paths:
30    /account/settings.json:
31      get:
32        description: |-
33          Returns settings (including
34          current trend, geo and sleep time information) for the authenticating user.
35        externalDocs:
36          url: 'https://dev.twitter.com/docs/api/1.1/get/account/settings'
37        operationId: account.settings.get
38        responses:
39          '200':
40            description: Successful Response
41        parameters:
42          - description: |-
43            The Yahoo! Where On Earth ID to use as the user's default trend location. Global informat
44            available by using 1 as the WOEID. The woeid must be one of the locations returned by GET
45            trends/available.
46
47            Example Values: 1
48            in: query
49            name: trend_location_woeid
50            required: false
51            type: string
52          - description: |-
53            When set to true, t or 1, will enable sleep time for the user. Sleep time is the time when
54            SMS notifications should not be sent to the user.
55
56            Example Values: true
57            in: query
58            name: sleep_time_enabled
59            required: false
60            type: string
```

NodeJS

NodeJS

- C'est un runtime Javascript écrit en C++.
- Il permet d'interpréter et exécuter du code Javascript.
- Il fournit par défaut un ensemble de bibliothèques pour interagir avec le système d'exploitation et concevoir des serveurs Web.

NodeJS API

Un ensemble riche de bibliothèques Javascript pour concevoir des serveurs Web.

V8 (chrome)

C'est le “moteur” qui permet à NodeJS d'interpréter et d'exécuter du code Javascript

Chrome



Parser

Execution
Engine

Garbage
Collector

JavaScript
runtime
(Call stack,
memory, etc.)

DOM API
Implementation

Chrome



chrome

```
Console.log(document.getElementById('information'));
```



Parser

Execution
Engine

Garbage
Collector

JavaScript
runtime
(Call stack,
memory, etc.)

DOM API
Implementation

NodeJS



Parser

Execution
Engine

Garbage
Collector

JavaScript
runtime
(Call stack,
memory, etc.)

**NodeJS API
Implementation**

NodeJS



```
Console.log(document.getElementById('information'));
```



Parser

Execution
Engine

Garbage
Collector

JavaScript
runtime
(Call stack,
memory, etc.)

**NodeJS API
Implementation**

document WHAT ?!

NodeJS



```
Console.log("Hello");
```



Parser

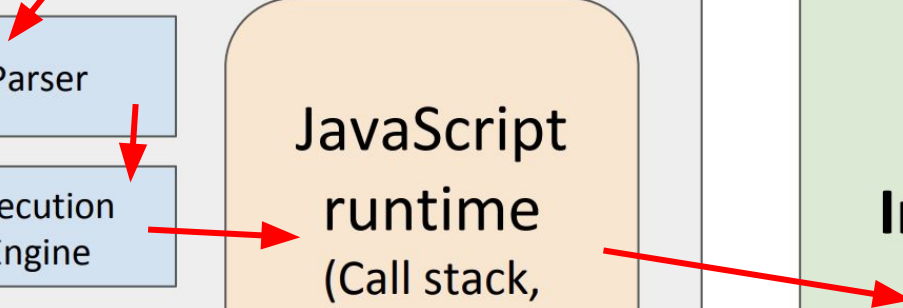
Execution
Engine

Garbage
Collector

JavaScript
runtime
(Call stack,
memory, etc.)

**NodeJS API
Implementation**

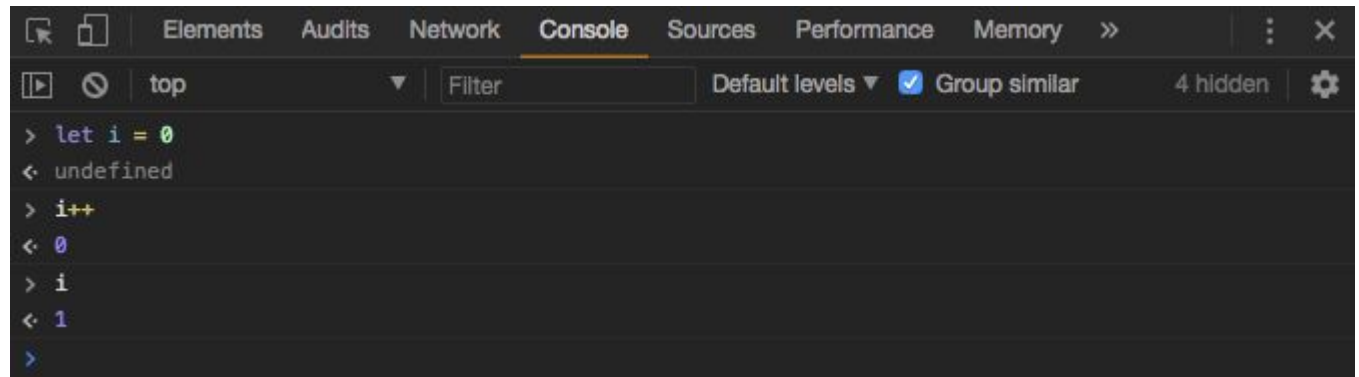
"Hello"



NodeJS en ligne de commande

Lancer node sans spécifier de fichier en argument le démarre une boucle REPL (Read-Eval-Print-Loop).

```
$ node  
> let i = 0  
undefined  
> i++  
0  
> i  
1
```



NodeJS avec des scripts

NodeJS peut également être utilisé à travers des scripts.

Pour le lancer, il suffit simplement d'exécuter node en spécifiant le fichier à exécuter en argument.

```
$ node monscript.js
```

```
function hello() {  
    console.log("Hello World!");  
}  
  
hello();
```

monscript.js

NPM: Node Package Manager

Lorsque vous installez node, vous installez également npm.

C'est un outil en ligne de commande qui vous permet de facilement installer des packages écrits en Javascript et compatibles avec NodeJS.

Pour rechercher des packages rendez-vous sur <https://npmjs.com>



NPM: Pour les nuls...

```
$ npm init
$ npm install express [--save] [-g]
$ npm uninstall express
$ npm start
$ npm install
```

package.json



```
1  {
2    "name": "web",
3    "version": "1.0.0",
4    "description": "",
5    "main": "server/server.js",
6    "scripts": {
7      "start": "node server/server.js",
8      "test": "echo \"Error: no test specified\" && exit 1"
9    },
10   "author": "",
11   "license": "ISC",
12   "dependencies": {
13     "express": "^4.16.3"
14   }
15 }
```

NPM: Pour les nuls...

```
1  {
2    "name": "web",
3    "version": "1.0.0",
4    "description": "",
5    "main": "server/server.js",
6    "scripts": {
7      "start": "node server/server.js",
8      "test": "echo \"Error: no test specified\" && exit 1"
9    },
10   "author": "",
11   "license": "ISC",
12   "dependencies": {
13     "express": "^4.16.3"
14   }
15 }
```

\$ npm start

\$ npm test

NPM: Pour les nuls...

```
1  {
2    "name": "web",
3    "version": "1.0.0",
4    "description": "",
5    "main": "server/server.js",
6    "scripts": {
7      "start": "node server/server.js",
8      "test": "echo \"Error: no test specified\" && exit 1",
9      "installDb": "node install_db.js"
10   },
11   "author": "",
12   "license": "ISC",
13   "dependencies": {
14     "express": "^4.16.3"
15   }
16 }
```

\$ npm **installDb**

ExpressJS

ExpressJS

```
const express = require('express');  
const app = express();
```

```
// répondre avec hello world quand on reçoit une  
requête GET
```

```
app.get('/', (req, res) => {  
  res.send('hello world');  
});
```

```
app.listen(3000, () => {  
  console.log("Serveur démarré");  
});
```

app est une instance d'ExpressJS.

Routes

Une route est définie comme suit:

```
app.method(path, handler)
```

- **method**: permet de définir la méthode HTTP de la requête.
- **path**: permet de définir le chemin de la ressource demandée.
- **handler**: représente la fonction qui va gérer la requête lors de sa réception.

Handler

Un handler reçoit toujours deux objets en paramètres. Ces objets sont créés par express et sont spécifiques à chaque requête reçue.

```
app.get('/', (req, res) => {  
  res.send('hello world');  
});
```

`res.send()` envoie la réponse avec un MIME/Content-type par défaut à “text/html”

Chaîner les Handler

Il est également possible de chaîner les Handlers, pour ce faire il suffit de spécifier le paramètre “next” et d’y faire appel.

```
app.get('/example', (req, res, next) => {  
    console.log('La réponse sera envoyée par la  
fonction suivante...');  
    next();  
}, (req, res) => {  
    res.send('Hello from B!');  
});
```

Ordre de déclaration des routes

L'ordre de déclaration des routes est **important**. Toujours mettre le chemin racine en dernier.

```
app.get('/products', (req, res) => {  
  res.send('products list');  
});
```

```
app.get('/', (req, res) => {  
  res.send('hello world');  
});
```

Méthodes de l'objet réponse

```
res.send('hello world');
```

```
res.status(404).end();
```

```
res.status(404).send('product not found.');
```

```
res.json(json_object);
```

```
res.redirect(301, 'http://example.com');
```

Paramètres d'une requête HTTP

Il existe plusieurs méthodes pour récupérer les paramètres d'une requête HTTP:

```
// http://localhost:3000/?prenom=john&nom=doe
app.get('/', (req, res) => {
  res.send(req.query.prenom);
});
```

Paramètres d'une requête HTTP

Il existe plusieurs méthodes pour récupérer les paramètres d'une requête HTTP:

```
// http://localhost:3000/john/doe
app.get('/:prenom/:nom', (req, res) => {
  var prenom = req.params.prenom
  res.send('Salut ' + prenom + ' !');
});
```

Headers d'une requête HTTP

Pour récupérer des headers depuis la requête entrante, il vous suffit de faire appel à la méthode `get()`.

```
req.get('user-agent');  
console.log(req.headers);
```

```
▼ Request Headers  
:authority: www.google.fr  
:method: GET  
:path: /  
:scheme: https  
accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8  
accept-encoding: gzip, deflate, br  
accept-language: fr-FR,fr;q=0.9,en-US;q=0.8,en;q=0.7  
cache-control: no-cache  
pragma: no-cache  
upgrade-insecure-requests: 1  
user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.162 Safari/537.36
```


Body d'une requête HTTP

Pour récupérer le body de la requête entrante, il vous suffit d'utiliser l'attribut **body** de l'objet **req**.

```
<form action="login" method="post">  
  <input type="text" id="email" name="email">  
  <input type="password" name="password">  
  <input type="submit" value="Submit">  
</form>
```

```
app.post('/login', function (req, res) {  
  res.json(req.body);  
});
```

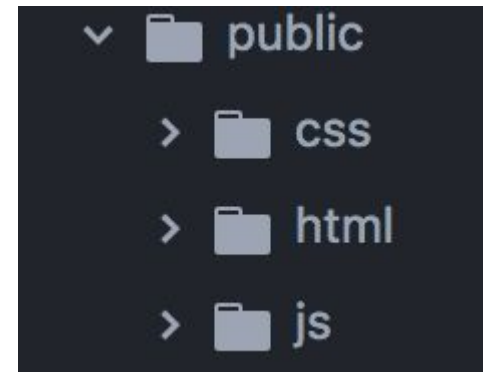
Données statiques

ExpressJS permet également de transmettre des fichiers **statiques** tels des fichiers html, css, js, jpg...

```
const express = require('express');  
const app = express();
```

```
app.use(express.static("public"));
```

```
app.get('/', (req, res) => {  
  res.send('hello world');  
});
```



Outils: Supervision

Redémarrer automatiquement le serveur lorsqu'un changement a été effectué sur un des fichiers du projet.

Différents outils :

- Forever
- nodemon
- pm2
- supervisor

Outils: cURL

C'est un outil qui va vous permettre de faire des requêtes depuis votre terminal avec les méthodes HTTP que vous voulez.

```
$ curl -X GET http://localhost:3000/john/doe
```

```
$ curl -X POST http://localhost:3000/john/doe
```

```
$ curl -X PUT http://localhost:3000/john/doe
```

```
$ curl -X DELETE http://localhost:3000/john/doe
```

Outils: Postman, insomnia

Si vous préférez utiliser plutôt une interface graphique riche en fonctionnalités, vous pouvez également utiliser postman ou encore insomnia.

getpostman.com

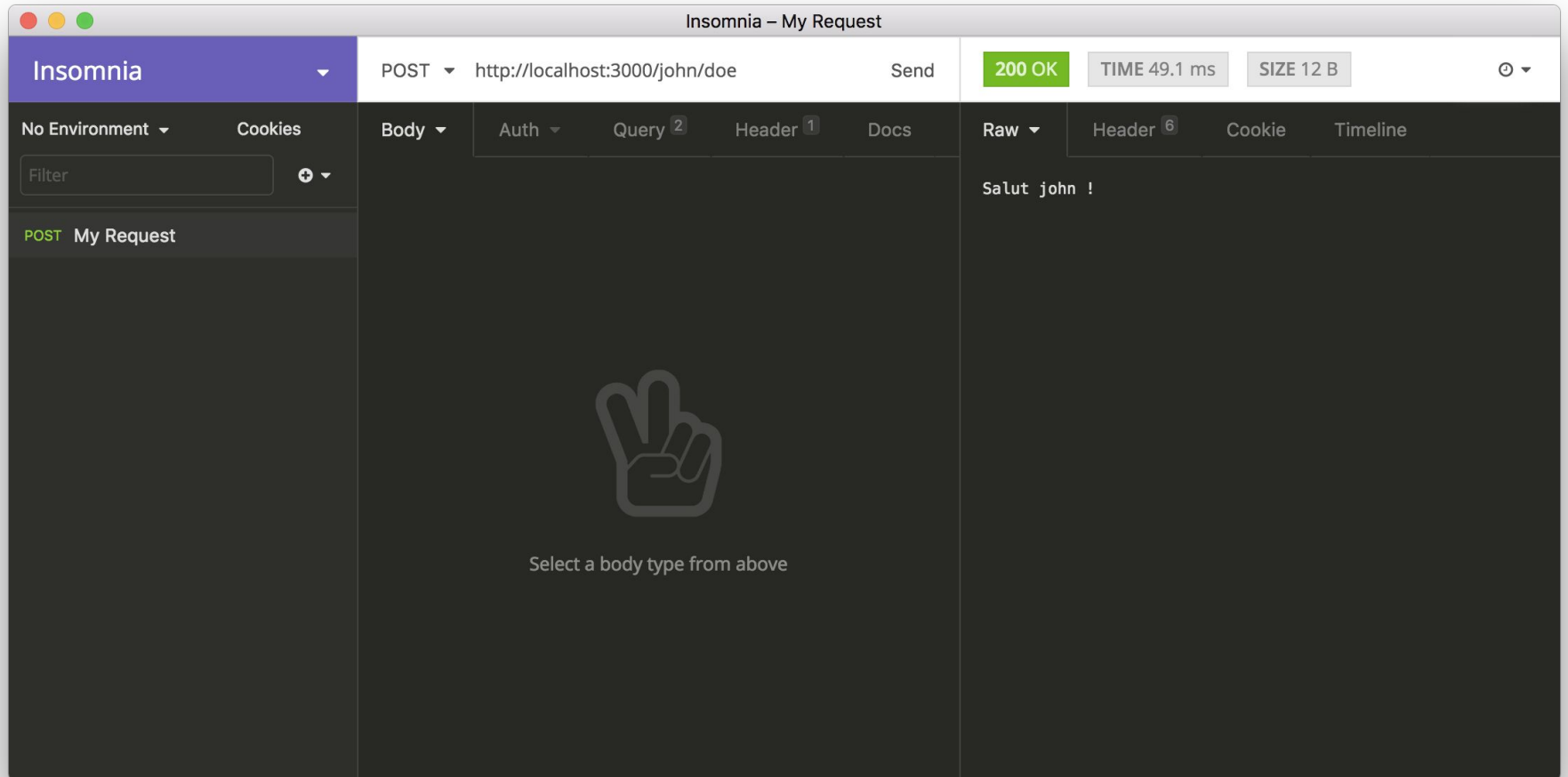
insomnia.rest



P O S T M A N



Outils: Insomnia



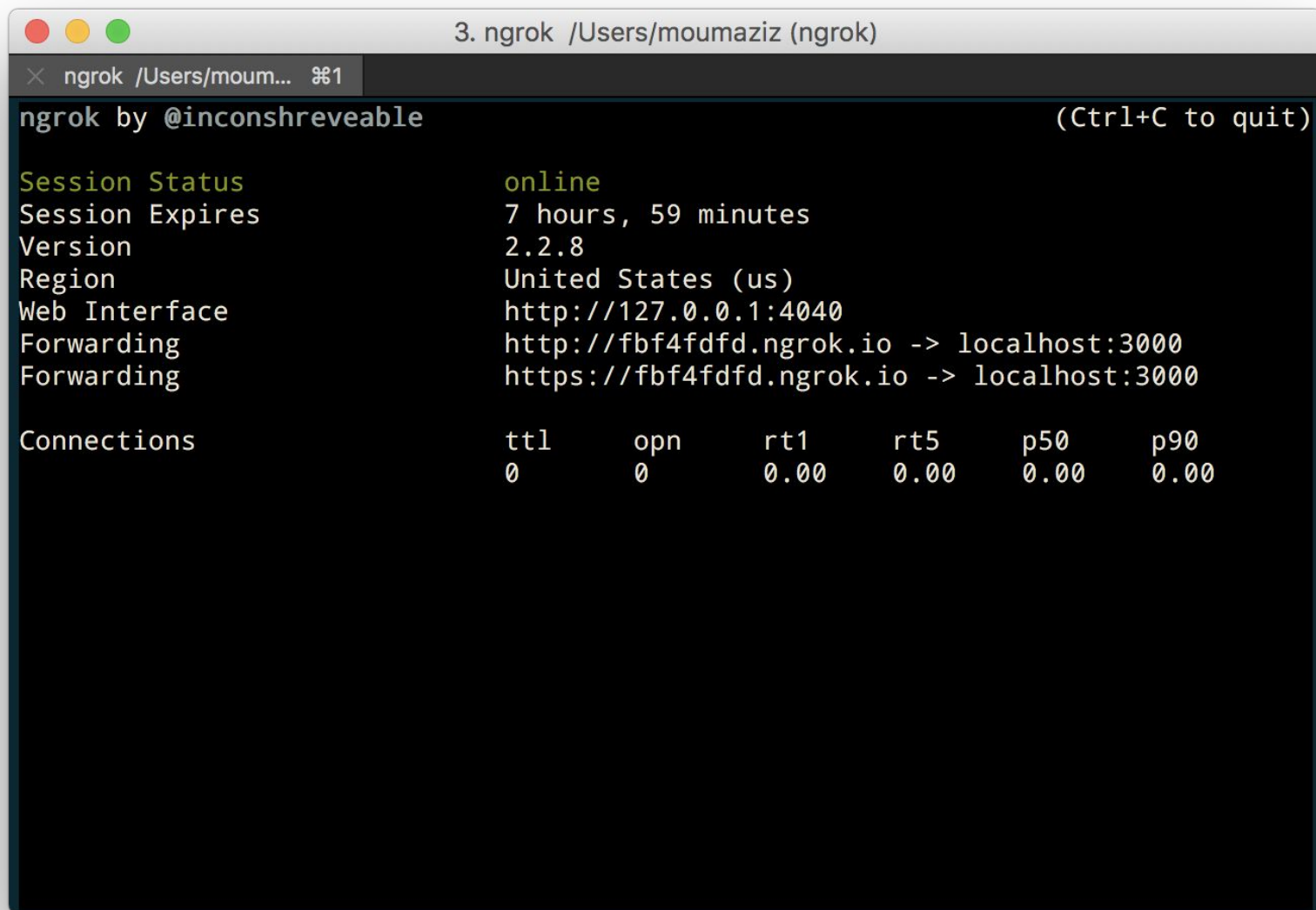
Outils: Ngrok

Si vous voulez partager votre localhost avec le reste du monde, Ngrok vous permet d'avoir une URL publique.

<https://ngrok.com>

ngrok

Outils: Ngrok



The screenshot shows a macOS terminal window with a title bar containing three colored window control buttons (red, yellow, green) and the text "3. ngrok /Users/moumaziz (ngrok)". The terminal has a single tab titled "ngrok /Users/moum... 1". The prompt is "ngrok by @inconsreveable" followed by "(Ctrl+C to quit)". The output displays the status of the ngrok session, including session status, expiration time, version, region, web interface, forwarding URLs, and a table of connection statistics.

```
ngrok by @inconsreveable (Ctrl+C to quit)

Session Status      online
Session Expires    7 hours, 59 minutes
Version            2.2.8
Region             United States (us)
Web Interface       http://127.0.0.1:4040
Forwarding          http://fbf4fdfd.ngrok.io -> localhost:3000
Forwarding          https://fbf4fdfd.ngrok.io -> localhost:3000

Connections
  ttl    opn    rt1    rt5    p50    p90
    0      0    0.00    0.00    0.00    0.00
```


ExpressJS (suite)

Rappel

```
const express = require('express');  
const app = express();
```

```
// répondre avec hello world quand on reçoit une  
requête GET
```

```
app.get('/', (req, res) => {  
  res.send('hello world');  
});
```

```
app.listen(3000, () => {  
  console.log("Serveur démarré");  
});
```

Rappel

```
const express = require('express');
```

```
const app = express();
```

```
// répondre avec hello world quand on reçoit une  
requête GET
```

```
app.get('/', (req, res) => {  
  res.send('hello world');  
});
```

```
app.listen(3000, () => {  
  console.log("Serveur démarré");  
});
```

Modules

NodeJS permet de charger des modules à travers la commande **require()**.

Pour créer votre propre module, il faut obligatoirement créer un fichier javascript (un module = un fichier).

Par défaut, tout est privé dans un module (variable, fonctions..). Pour qu'une variable ou fonction ne soit pas privée, il faudra clairement le spécifier.

Modules

Chaque fichier JS possède un objet qui lui est propre nommé **module**.

Lorsqu'on fait appel à la fonction **require()**, c'est l'attribut **exports** de l'objet **module** du fichier JS qu'on import qui sera retourné. Il est vide par défaut.

```
module.exports = "Hello World"
module.exports = (req, res) => {
  res.send("Hello World")
}
```

Modules

```
function printHello() {  
    console.log("Hello")  
}
```

```
function printWorld() {  
    console.log("World!")  
}
```

```
module.exports.printHello = printHello
```

```
module.exports.printWorld = printWorld
```

Modules

Pour **require()** un module présent dans notre projet, il faudra contrairement à ce qu'on a vu précédemment, clairement spécifier le chemin relatif du fichier JS sans son extension.

```
require("./module1/fichier")
```

Si on ne spécifie pas le chemin, la fonction **require()** ira chercher le module dans le dossier **node_modules**.

Middleware

Avec ExpressJS, toutes les fonctions qui ont comme argument la fonction **next ()** ou **non** sont appelés **middleware**.

Nous avons vu ça précédemment avec les **handlers** pour gérer les routes, on avait dit qu'on pouvait chaîner les handlers. Les handlers sont donc des **middlewares**.

```
function checkAuth(req, res, next) {  
  if (req.get("API-KEY")) next()  
  else res.send("Error: Auth missing")  
}  
app.get( " /", checkAuth, ...)
```


Middleware: `app.use()`

Il est également possible de définir des Middleware qui seront exécutés au début de chaque nouvelle requête entrante.

Ceci peut être utile pour par exemple définir des variables dans les objets `req` et `res` qui pourront être accessibles à tout le reste de l'application.

Il suffit simplement d'utiliser la fonction **`use()`** de l'objet **`app`**.

```
app.use(checkAuth)
```

```
app.use("/user/:id", checkAuth)
```

Requêtes avec données dans le corps

```
1  POST /login HTTP/1.1
2  Host: foo.com
3  Content-Type: application/x-www-form-urlencoded
4  Content-Length: 37
5
6  username=foo@foo.com&password=123_foo
```

Requêtes avec données dans le corps

```
1  POST /login HTTP/1.1
2  Host: foo.com
3  Content-Type: application/x-www-form-urlencoded
4  Content-Length: 37
5
6  username=foo@foo.com&password=123_foo
```

BodyParser

C'est une bibliothèque vous permettant de directement parser le corps d'une requête. Le résultat sera directement disponible dans l'objet **request**.

BodyParser est **middleware**.

```
$ npm install body-parser
```

BodyParser

```
const bodyParser = require("body-parser")
```

```
// Content-type: application/json
```

```
app.use(bodyParser.json())
```

```
// Content-type: application/x-www-form-urlencoded
```

```
app.use(bodyParser.urlencoded({ extended: false })))
```

BodyParser

```
app.post("/products", (req, res) => {  
  product = {  
    name: req.body.name,  
    price: req.body.price  
  }  
  res.json(product)  
})
```

MongoDB

MongoDB

C'est une base de données **orientée documents**. Contrairement à une base de données relationnelle, une **BDOD** garde un ensemble de **collections** (tables) composées d'un ensemble de **documents** (lignes).

Un document n'est rien d'autre qu'un **objet JSON**. En réalité cet objet JSON sera stocké en tant qu'objet **BSON** (Binaire).

```
{  
  _id: 5abe492a8cbadb22dc80ab54  
  "nom": "iPhone X",  
  "prix": 1159  
}
```


MongoDB: Schéma

Contrairement à une base de données relationnelle le **schéma** n'est pas fixé à l'avance.

Une même collection peut contenir différents documents (objets) de structure différentes.

```
{
  _id: 5abe492a8cbadb22dc80ab54
  "nom": "iPhone X",
  "prix": 1159
}
{
  _id: 4afe3fe83611502135847759
  "nom": "iPhone X",
  "description": "Dernier iPhone"
}
```

MongoDB vs DB relationnelle

Une des plus grandes différences est que dans MongoDB il n'y a pas de **clés étrangères** ou de **jointures**.

Cependant, il est possible à un document JSON **d'inclure** un autre document JSON mais il ne peut pas le référencer.

Évidemment, il est toujours possible de référencer un autre document à travers son **_id**, mais il faudra le faire **manuellement**.

MongoDB est ce qu'on appelle une base de données **NoSQL**.

MongoDB

Une fois mongoDB installé, il faudra le lancer depuis le terminal:

```
$ mongod
```

Le serveur sera donc lancé et écoutera sur le port **27017**.

Afin d'interagir avec le serveur, il est possible d'utiliser le client mongo depuis le terminal:

```
$ mongo
```

Commandes shell MongoDB

> show dbs

Affiche toutes les bases de données.

> use NomBD

Passer de la BD courante à la BD NomBD.

> show collections

Affiche les collections de la BD courante.

MongoDB et NodeJS

Afin de pouvoir interagir avec la base de données MongoDB depuis NodeJS, il nous faudra récupérer un nouveau module qu'on appelle driver.

Il existe plusieurs drivers MongoDB pour NodeJS, nous allons dans cours utiliser le module officiel "MongoDB":

```
$ npm install mongodb --save
```

Objets du module MongoDB

Le module propose plusieurs objets permettant de manipuler les bases de données, collections et documents:

- L'objet **db** qui nous permet de récupérer les collections d'une base de donnée précise.
- L'objet **Collection** permet de récupérer, insérer, modifier et supprimer des documents.
- Les documents sont simplement des objets JavaScript.

MongoDB: récupérer l'objet Db

Pour récupérer la référence de l'objet db, il faudra utiliser la fonction suivante:

```
MongoClient.connect(url, callback)
```

Où:

- **url**: est la chaîne de caractère utilisée pour se connecter à mongodb.
- **callback**: Fonction appelée une fois connecté avec comme argument la référence à l'objet **db**.

MongoDB: récupérer l'objet Db

```
const MongoClient = require('mongodb').MongoClient;  
const MONGO_URL = 'mongodb://localhost:27017/maDb';
```

```
// Avec callback
```

```
MongoClient.connect(MONGO_URL, (err, database) => {  
    db = database;  
})
```


MongoDB: récupérer l'objet Db

```
const MongoClient = require('mongodb').MongoClient;
const MONGO_URL = 'mongodb://localhost:27017/maDb';

let db = null;

function onConnected(err, database) {
  db = database;
}

// Avec promesse
MongoClient.connect(MONGO_URL)
  .then(onConnected)
```

MongoDB: récupérer l'objet Collection

Une fois l'objet db récupéré, il est possible de récupérer l'objet collection à travers une fonction que possède l'objet **db**:

```
const coll = db.collection("maCollection")
```

La fonction collection est **synchrone**.

Elle nous retourne un objet que l'on peut utiliser pour ajouter/rechercher/modifier/supprimer des documents dans notre collection.

Si la collection n'existe pas, elle sera **automatiquement** créée au moment de l'écriture.

```
collection.insertOne()
```

```
collection.insertOne(doc, callback);
```

Permet d'insérer un document dans une collection.

- **doc**: n'est rien d'autre qu'un objet Javascript contenant les données qui seront sauvegardées dans notre collection en tant que document.
- **callback**: fonction qui sera appelée à la fin de la sauvegarde, elle a deux arguments: err, result. où result.**insertedId** représente le **_id** du document.

```
collection.insertOne()
```

```
function insertProduct(name, price) {  
  const product = {  
    "name": name,  
    "price": price  
  }  
}
```

```
collection.insertOne(product, (err, result) => {  
  console.log(result.insertedId)  
})  
}
```

```
collection.findOne()
```

```
collection.findOne(query [, options], callback);
```

Permet de rechercher un document ayant les caractéristiques spécifiées dans la **query**.

La query n'est autre qu'un objet Javascript ayant les associations clés/valeur que l'on recherche.

```
collection.findOne()
```

```
function findProduct(name) {  
  collection.findOne(  
    {"name": name},  
    (err, product) => {  
      return product;  
    }  
  )  
}
```

collection.findOne() avec ObjectId

Et si on souhaitait retrouver un document à partir de son **_id**?

```
function findProduct(id) {  
  collection.findOne(  
    { "_id": id },  
    (err, product) => {  
      return product;  
    })  
}
```

collection.findOne() avec ObjectId

Et si on souhaitait retrouver un document à partir de son **_id**?

```
function findProduct(id) {  
  collection.findOne(  
    { "_id": id },  
    (err, product) => {  
      return product;  
    }  
  )  
}
```

Ne marche pas !

collection.findOne() avec ObjectID

Avant de rechercher un document avec son `_id`, il nous faut convertir la chaîne de caractère que l'on a qui correspond à son `_id` en un **ObjectID**.

```
const ObjectID = require('mongodb').ObjectID
```

```
function findProduct(id) {  
  collection.findOne( { "_id": ObjectID(id) },  
    (err, product) => {  
      return product;  
    })  
}
```

collection.find()

Fonctionne de la même manière que **findOne()** à l'exception qu'elle nous retourne non pas un document mais un **curseur** qui pointe sur le **premier** document.

On peut utiliser **hasNext** et **next** pour avancer le **curseur**.

```
db.users.find(  
  { age: { $gt: 18 } },  
  { name: 1, address: 1 }  
) .limit(5)
```

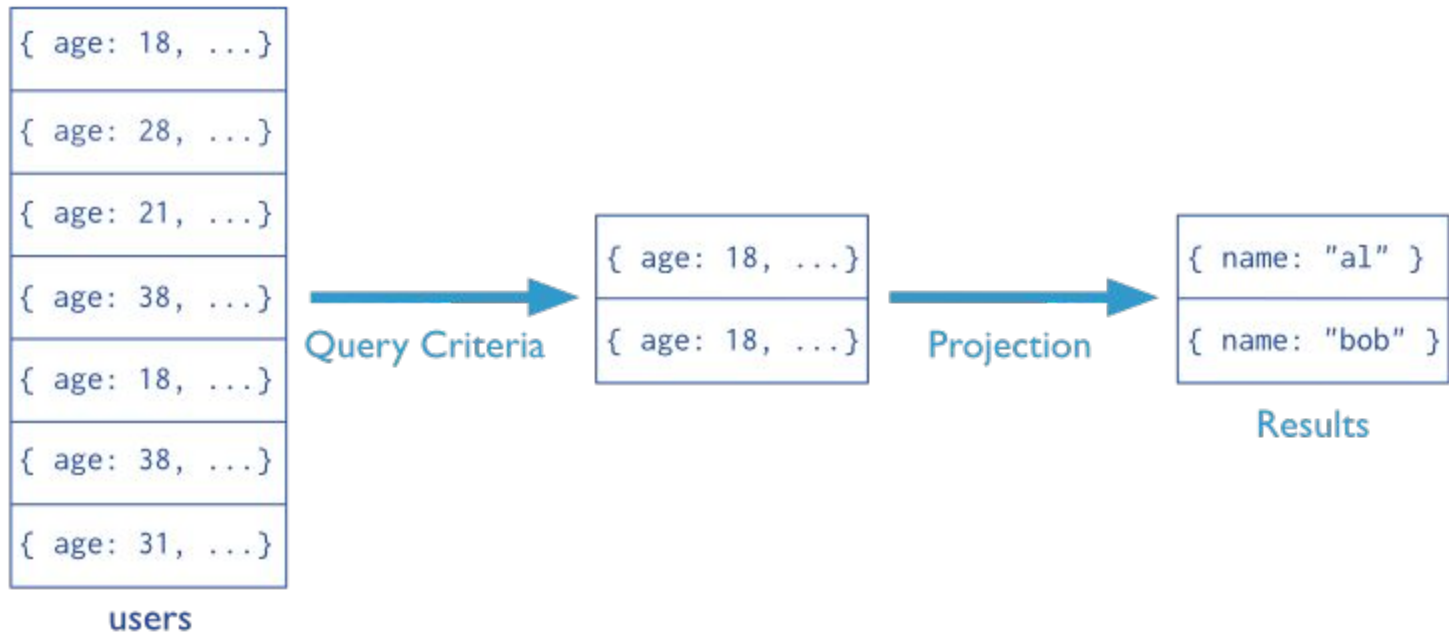
← collection
← query criteria
← projection
← cursor modifier

```
SELECT _id, name, address  
FROM   users  
WHERE  age > 18  
LIMIT 5
```

← projection
← table
← select criteria
← cursor modifier

Requêtes et projections

Collection Query Criteria Projection
`db.users.find({ age: 18 }, { name: 1, _id: 0 })`



Curseur et `.toArray()`

Chaque curseur possède une fonction permettant de le convertir en un tableau possédant tous les documents que le curseur pointe dans les limites de la mémoire disponible.

```
collection.findOne(...).toArray((err, items) => {  
    return items;  
})
```

```
collection.update()
```

```
collection.update(query, newDocument);
```

C'est la version la plus basique pour mettre à jour des documents. Elle permet de directement remplacer les documents qui correspondent à la **query** avec le contenu de **newDocument**.

collection.update()

```
function updateProduct(name, price) {  
  const old_product = {  
    "name": name  
  }  
  const new_product = {  
    "name": name,  
    "price": price  
  }  
  
  collection.update(old_product, new_product)  
}
```

collection.update() et upsert

```
collection.update(query, newDocument, params);
```

En plus des arguments vus précédemment, la fonction update supporte aussi d'autres paramètres en argument, tel l'argument **upsert** qui permet à la fonction en plus de mettre à jour le document, **d'automatiquement** créer l'entrée si la query ne retourne **aucun** résultat.

collection.update() et upsert

```
function updateProduct(name, price) {  
  const old_product = {  
    "name": name  
  }  
  
  const new_product = {  
    "name": name,  
    "price": price  
  }  
  
  const params = { upsert: true }  
  
  collection.update(old_product, new_product, params)  
}
```



```
collection.deleteOne()
```

```
collection.deleteOne(query, callback);
```

Permet de supprimer le premier document qui correspond à la **query**.

Le callback reçoit en paramètre: **err** et **result**, ou **result** possède une variable **result.deletedCount** qui indique le nombre de document supprimés, dans ce cas-ci un seul.

`collection.deleteMany()`

```
collection.deleteMany(query, callback);
```

Permet de supprimer tous les document qui correspondent à la **query**.

Le callback reçoit en paramètre: **err** et **result**, ou result possède une variable **result.deletedCount** qui indique le nombre de document supprimés, dans ce cas-ci un seul.

```
collection.deleteMany()
```

Permet de vider toute la collection en une fois.

Opérateurs de Requêtes sur les documents

MongoDB possède une syntaxe particulière pour les requêtes permettant de faire des recherches plus sophistiquées.

```
db.users.find(  
  { age: { $gt: 18 } },  
  { name: 1, address: 1 }  
)
```

← collection
← query criteria
← projection
← cursor modifier

```
db.inventory.find( { ratings: { $gt: 5, $lt: 9 } } )
```

copy

```
db.products.update(  
  { _id: 100 },  
  { $set: { "details.make": "zzz" } }  
)
```

copy

Optimisation: indexes

```
// On crée notre index
```

```
db.records.createIndex( { userid: 1 } )
```

```
// On crée notre index multiple de produits
```

```
db.products.createIndex( { item: 1, category: 1,  
price: 1 } )
```

```
// Query exécutée très rapidement
```

```
db.records.find( { userid: { $gt: 10 } } )
```

Optimisation: connexions multiples (pooling)

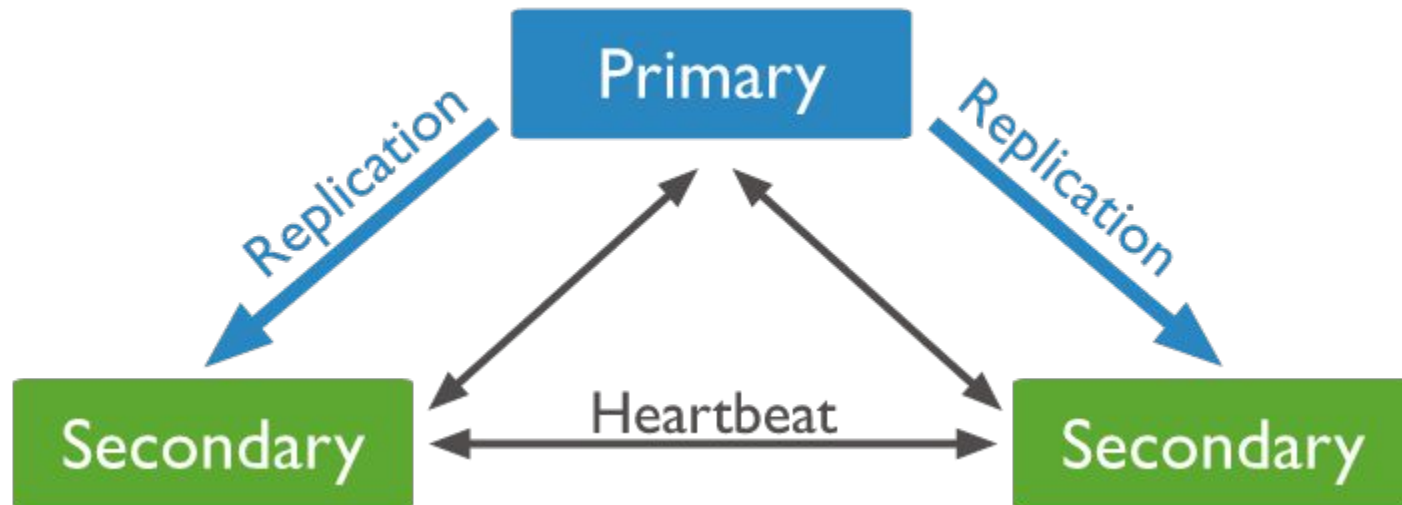
```
const MongoClient = require('mongodb').MongoClient;  
const MONGO_URL = 'mongodb://localhost:27017/maDb';
```

```
let db = null;  
function onConnected(err, database) {  
  db = database;  
}
```

// Avec promesse

```
MongoClient.connect(MONGO_URL, {poolSize: 10})  
  .then(onConnected)
```

MongoDB: réplication



Utiliser MongoDB avec ExpressJS

```
MongoClient.connect(MONGO_URL, (err, database) => {  
  db = database;  
  app.get("/", (req, res) => {  
    db.collection("products").find({}, (err, items)  
=> { res.json(items) })  
  })  
  app.listen(3000, () => {  
    console.log("En attente de requêtes...")  
  })  
})
```

Utiliser MongoDB avec ExpressJS

```
let db = null;

function onConnected(err, database) {
  db = database;
  app.get("/", (req, res) => {
    db.collection("products").find({}, (err, items)
=> { res.json(items) })
  })
}
```

```
MongoClient.connect(MONGO_URL)
  .then(onConnected)
```


MongoDB: Studio 3T

Studio 3T for MongoDB - Non-Commercial License

Connect Collection IntelliShell SQL Aggregate Map-Reduce Export Import Users Roles Schema Compare Feedback

Search Open Connections (Cmd+F) ...

New Connection - imported on 15 sept. 2017 localhost:27017

- admin
- local
- products_manager
 - Collections (1)
 - products
 - Views (0)
 - GridFS Buckets (0)
 - System (0)

products products

New Connection - imported on 15 sept. 2017 localhost:27017 products_manager products

Query {} Query Builder

Projection {} Sort {}

Skip Limit

Result Query Code Explain

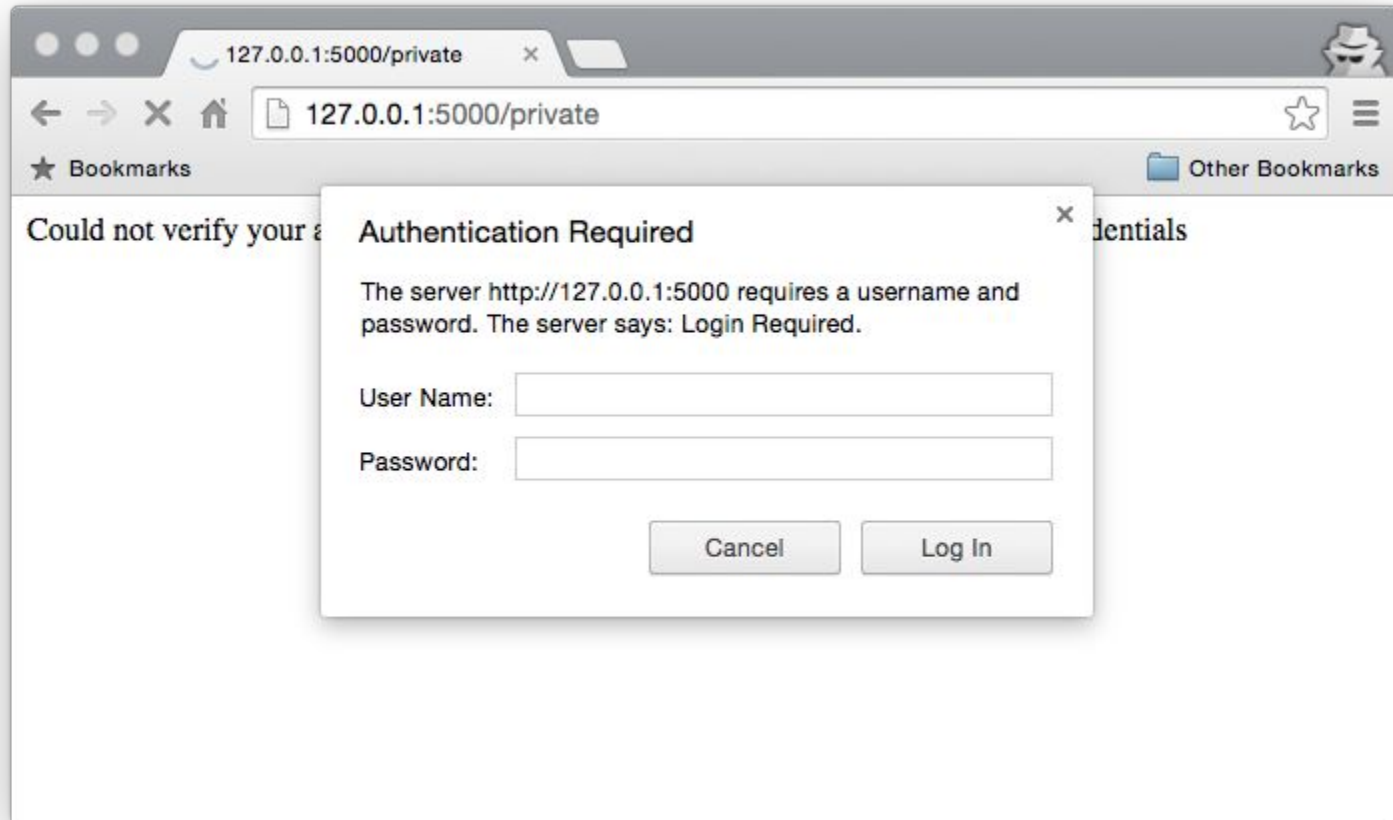
50 Documents 1 to 13 Tree View

Key	Value	Type
(1) {_id : 5abe303fd9a82e187eada162}	{ 3 fields }	Document
(2) {_id : 5abe3bc2dad75e206193d97c}	{ 3 fields }	Document
(3) {_id : 5abe3fd7f219022122a012f8}	{ 3 fields }	Document
(4) {_id : 5abe3fe83611502135847759}	{ 3 fields }	Document
(5) {_id : 5abe400a361150213584775a}	{ 3 fields }	Document
(6) {_id : 5abe4031361150213584775b}	{ 3 fields }	Document
(7) {_id : 5abe403a361150213584775c}	{ 3 fields }	Document
(8) {_id : 5abe4044361150213584775d}	{ 3 fields }	Document
(9) {_id : 5abe4650f5ff2b223adfadb1}	{ 3 fields }	Document
(10) {_id : 5abe4651f5ff2b223adfadb2}	{ 3 fields }	Document
(11) {_id : 5abe4657f5ff2b223adfadb3}	{ 3 fields }	Document
(12) {_id : 5abe492a8cbadb22dc80ab53}	{ 3 fields }	Document
(13) {_id : 5abe492a8cbadb22dc80ab54}	{ 3 fields }	Document
_id	5abe492a8cbadb22dc80ab54	Objectid
name	iPhone X	String
price	1000	String

Operations 0 items selected Count Documents 0.002s

Authentication

Types d'authentification: Basic Auth



Types d'authentification: Basic Auth

L'authentification peut se faire deux manières différentes:

- Directement dans l'URL:

<http://toto:password@example.com>

- En tant que header:

Authorization: Basic AZIBAFJRFjpPcGVuU84JFN1I

Types d'authentification: API Key

La clé peut être transmise dans l'URL ou l'en-tête.

Curl

```
curl -X GET "https://nightly.apinf.io:3002/kithuppi/emojis" -H "accept: application/json" -H "X-API-Key: W284fJGLGzBCPyuNrKs2rAe1fnwz6AY6HZLD3F6"
```

Server response

Code

Details

Undocumented

TypeError: Failed to fetch

Response headers

Types d'authentification: JSON Web Token

C'est un standard définissant une méthode légère et sécurisée pour transmettre une information à travers un objet JSON.

L'information étant transmise, on pourra aisément vérifier sa validité.

header.payload.signature

JWT: Header

Le header contient généralement deux informations:

- Le **type du token**, dans notre cas **JWT**.
- L'**algorithme de hashage** utilisé, tels HMAC SHA256 ou RSA.

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

Par la suite, il sera encodé en **Base64Url**.

JWT: Payload

Il est composé de 3 types de “réclamations” (claims):

1. *réclamations inscrites*: Non obligatoire, représente des informations utiles: **iss** (issuer), **exp** (expiration time), **sub** (subject), **aud** (audience), ...
2. *réclamations publiques*: Libre, définies par le développeur.
3. *réclamations privées*: réclamations spécifiques qui ne sont ni des réclamations inscrites ou publiques.

JWT: Payload

Voici un exemple:

```
{  
  "sub" : "1234567890",  
  "name" : "John Doe",  
  "admin" : true  
}
```

Par la suite, il sera encodé en **Base64Url**.

JWT: Signature

La signature est utilisée pour s'assurer que les informations n'ont pas été modifiées en chemin.

Il est également possible si on utilise un algorithme asymétrique de s'assurer à travers elle que la personne qui a transmis l'information est bien celle qu'elle dit qu'elle est.

```
HMACSHA256(base64UrlEncode(header) + "." +  
            base64UrlEncode(payload), secret)
```

OAuth1a, OAuth2

OAuth (protocol d'autorisation) permet a une application d'avoir une autorisation pour accéder à certaines informations.

