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.

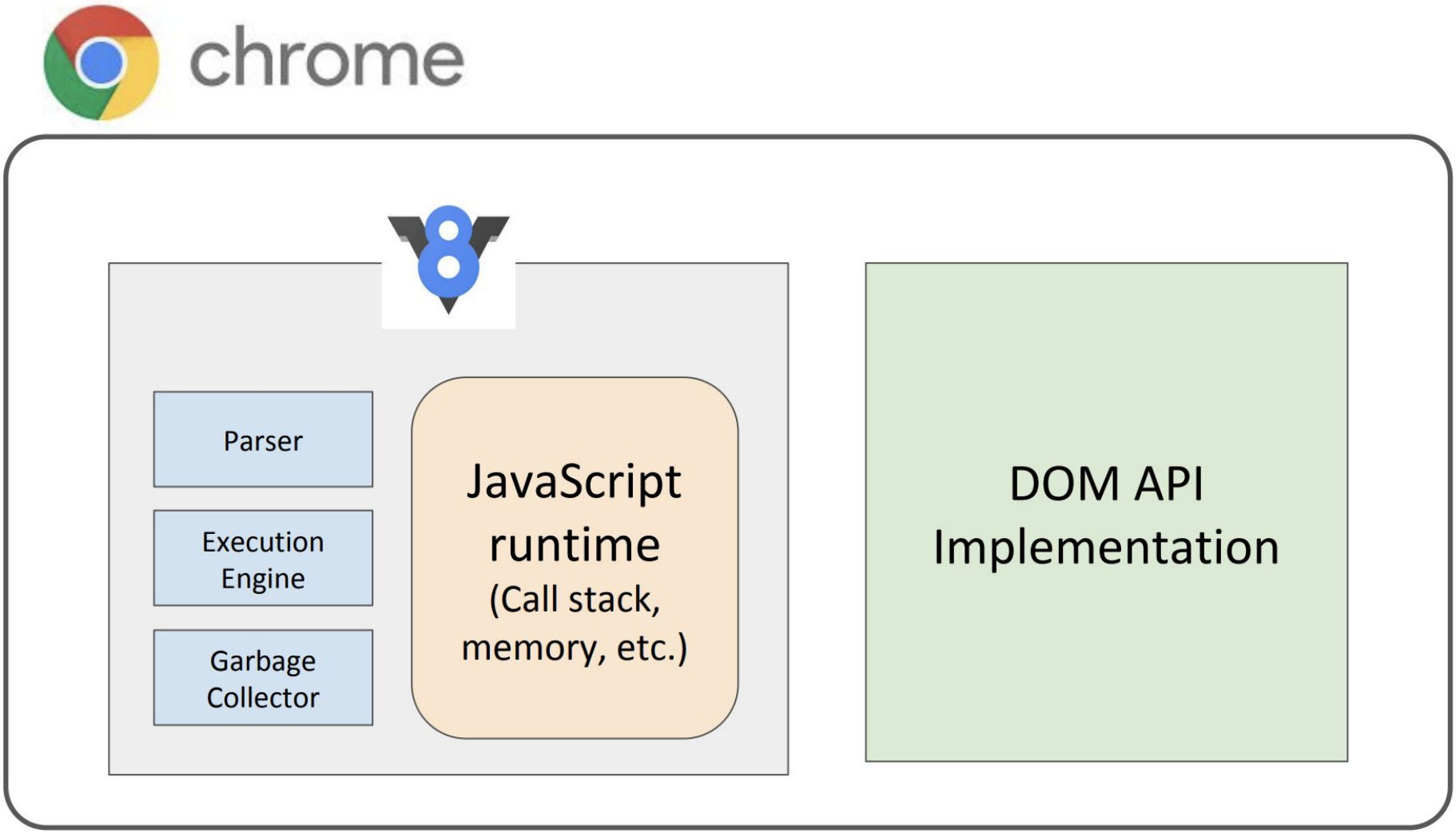
NodeJS API

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

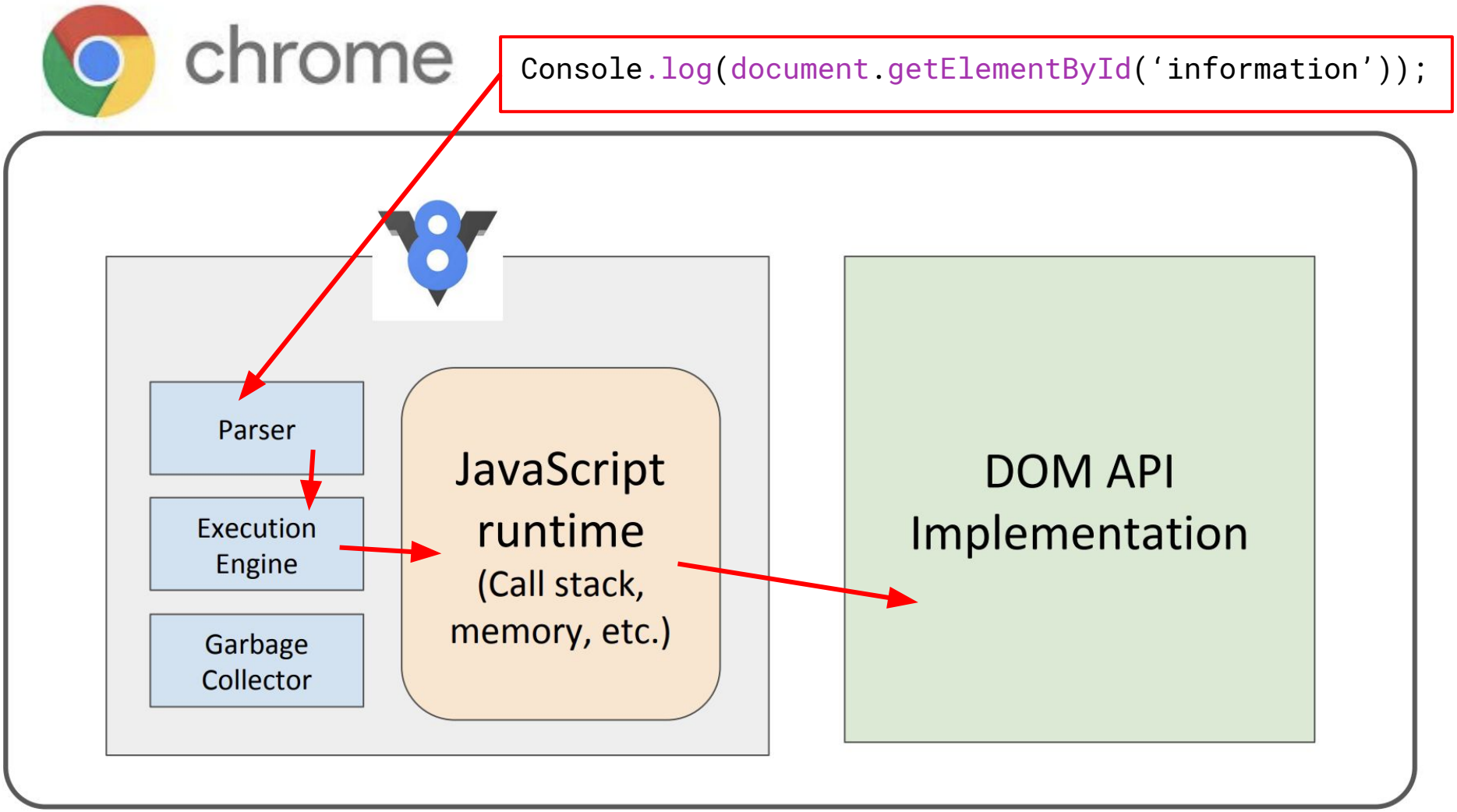
V8 (chrome)

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

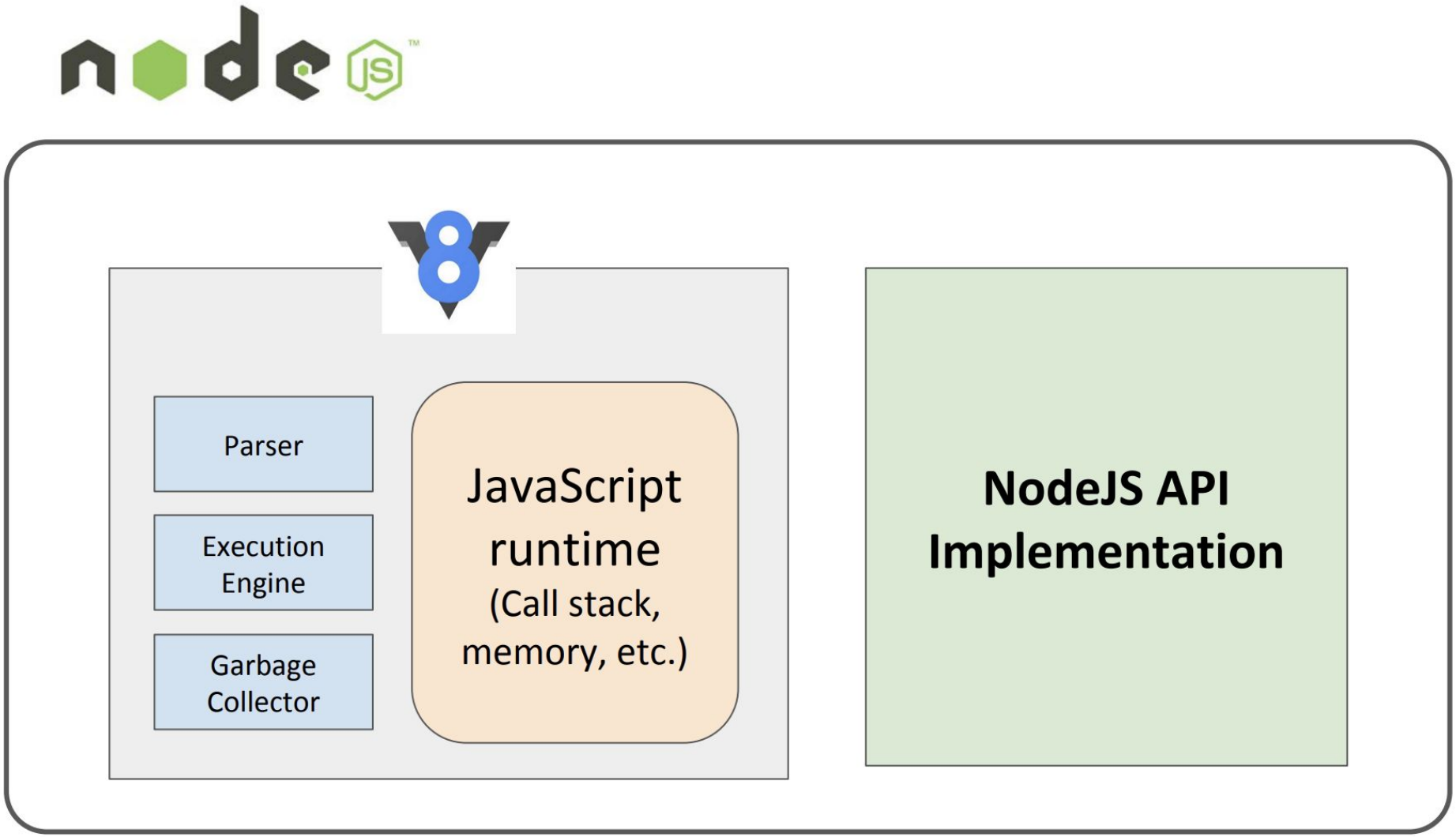
Chrome


chrome
Parser
Execution Engine
Garbage Collector
JavaScript runtime
(Call stack, memory, etc.)
DOM API Implementation

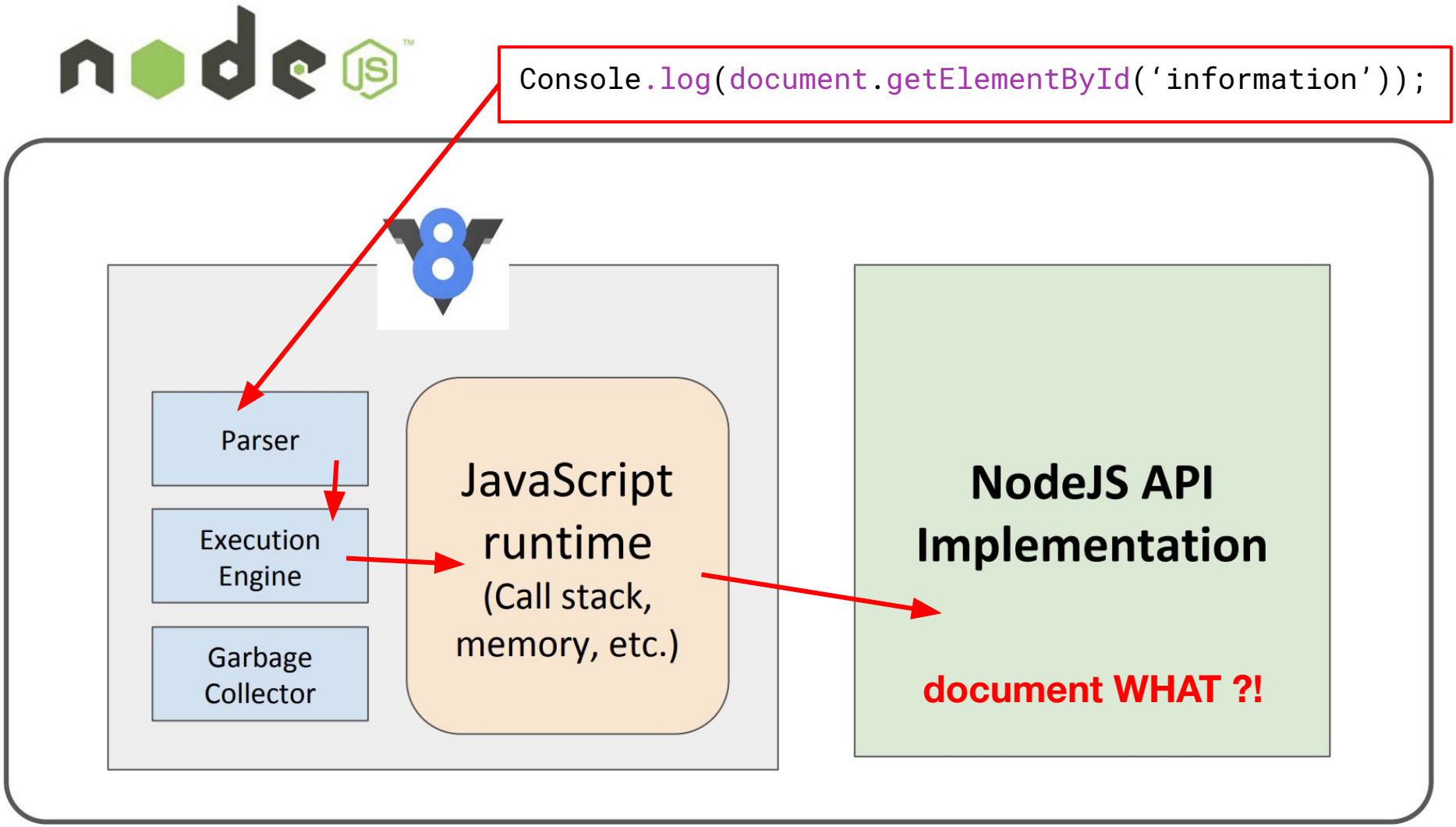
Chrome


chrome
Console.log(document.getElementById('information'));
Parser
Execution Engine
Garbage Collector
JavaScript runtime
(Call stack, memory, etc.)
DOM API Implementation

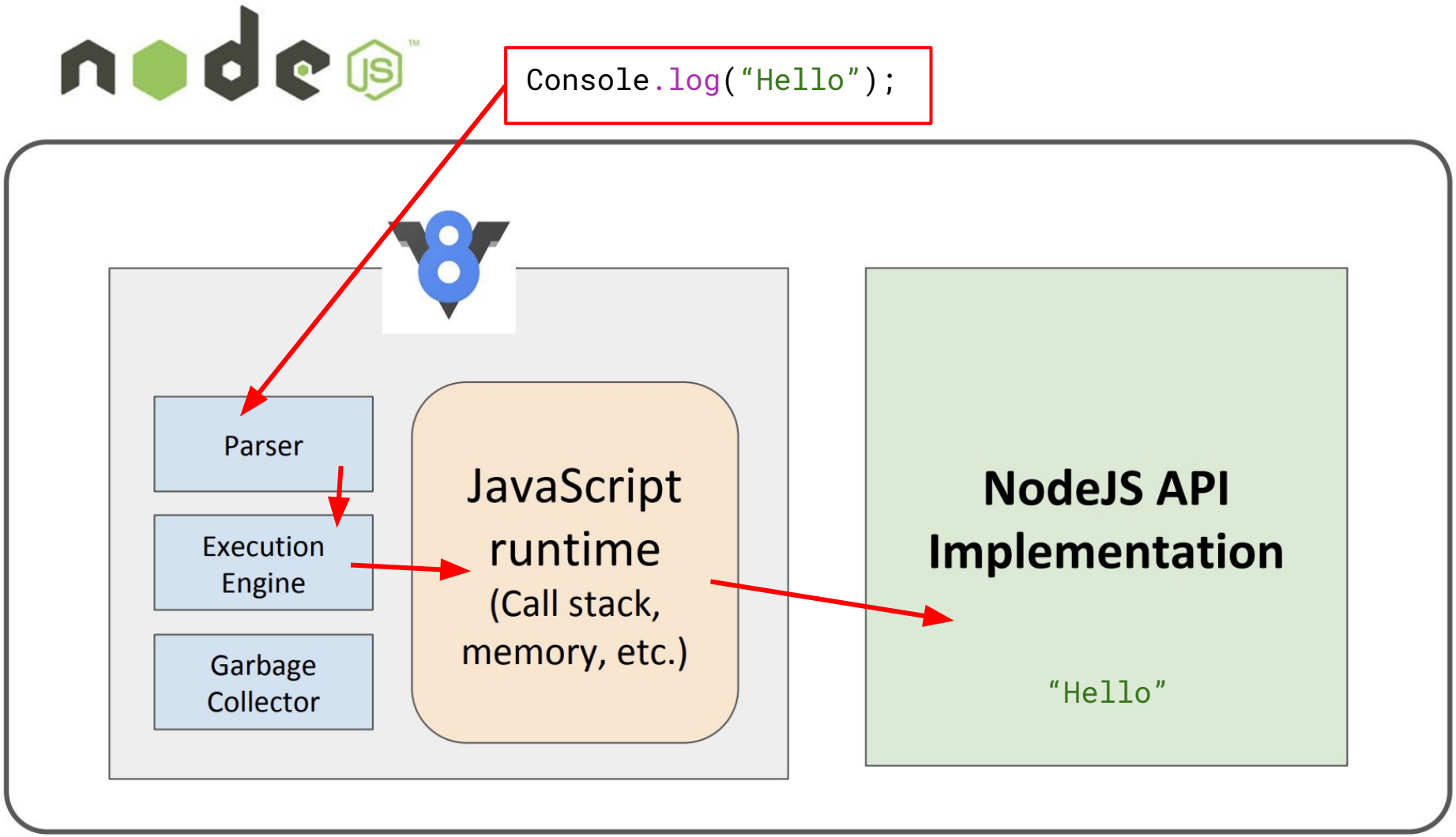
NodeJS


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 ?!

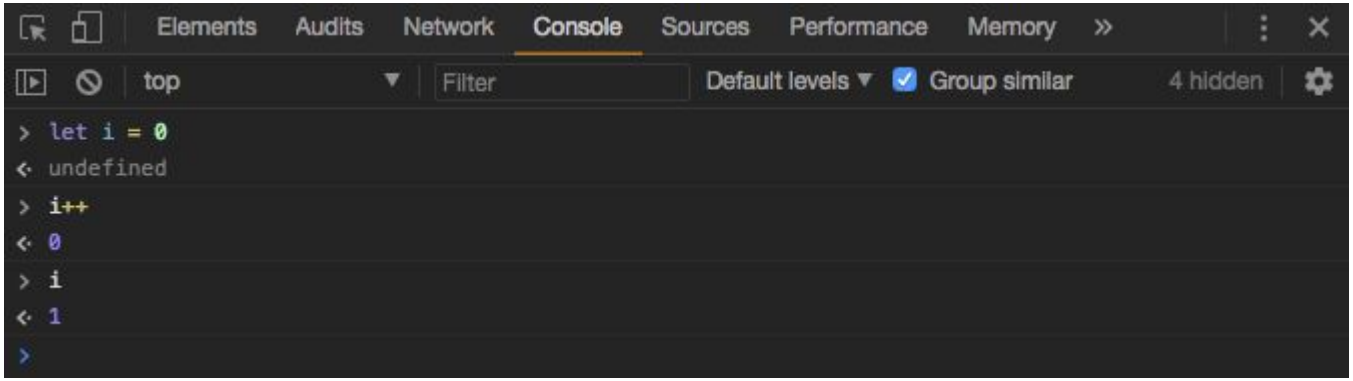
NodeJS


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 →

```json
{
  "name": "web",
  "version": "1.0.0",
  "description": "",
  "main": "server/server.js",
  "scripts": {
    "start": "node server/server.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.16.3"
  }
}
```

NPM: Pour les nuls...

```
{
  "name": "web",
  "version": "1.0.0",
  "description": "",
  "main": "server/server.js",
  "scripts": {
    "start": "node server/server.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.16.3"
  }
}
```

```
$ npm start
$ npm test
```

NPM: Pour les nuls...

```
{
  "name": "web",
  "version": "1.0.0",
  "description": "",
  "main": "server/server.js",
  "scripts": {
    "start": "node server/server.js",
    "test": "echo \"Error: no test specified\" && exit 1",
    "installDb": "node install_db.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.16.3"
  }
}
```

```
$ 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 est 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

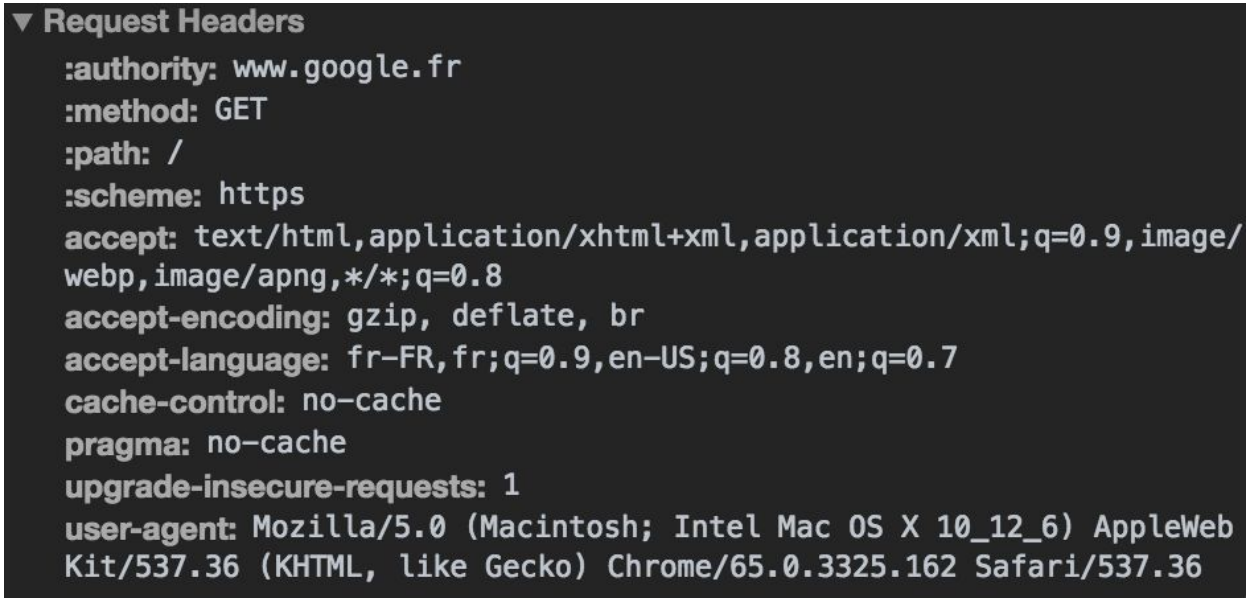
Il existe plusieurs méthodes pour récupérer les paramètres d’une requête HTTP:

```
// 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);
```



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
```

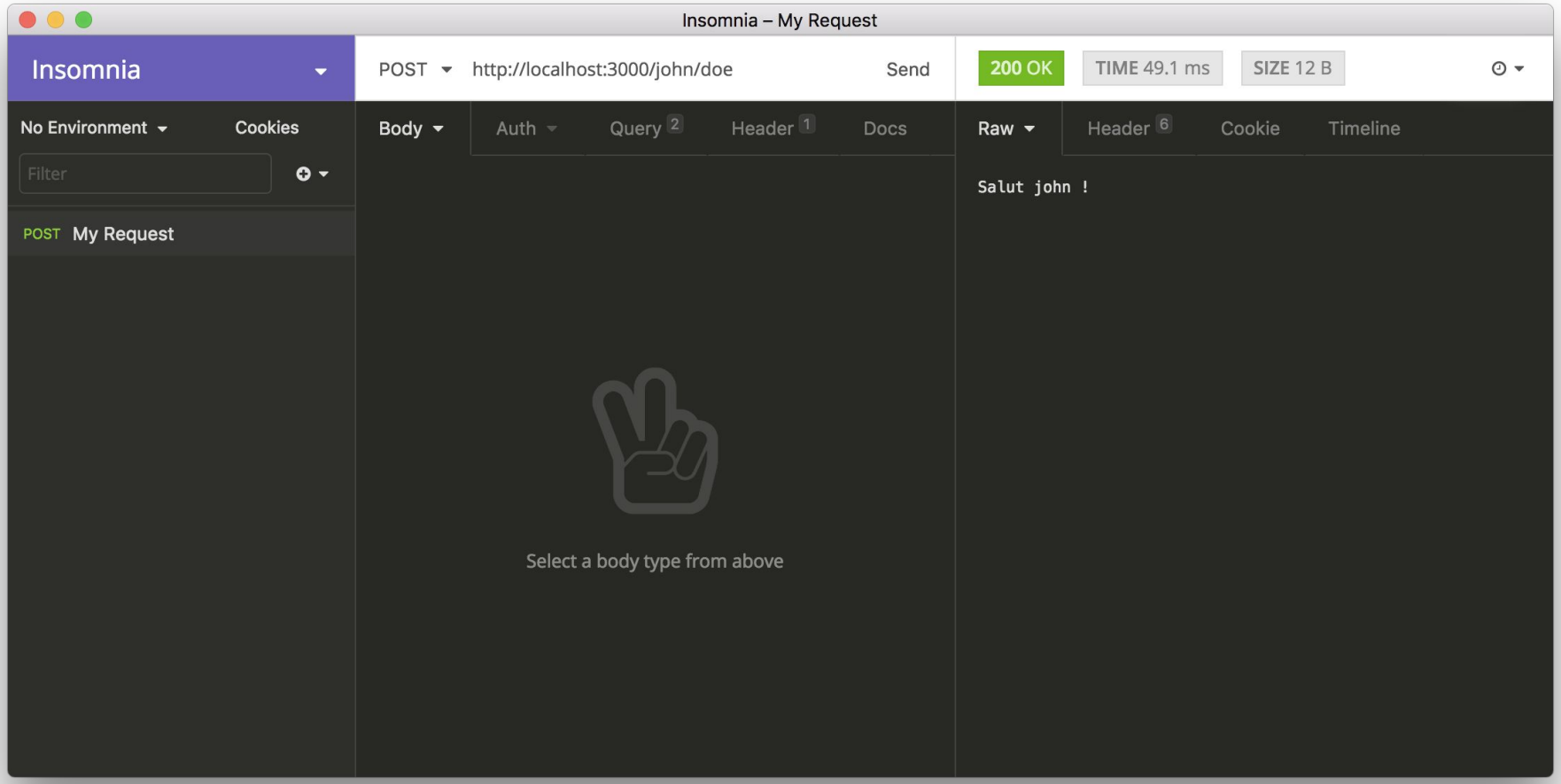
Outils: Postman, insomnia

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

getpostman.com

insomnia.rest

Outils: Insomnia

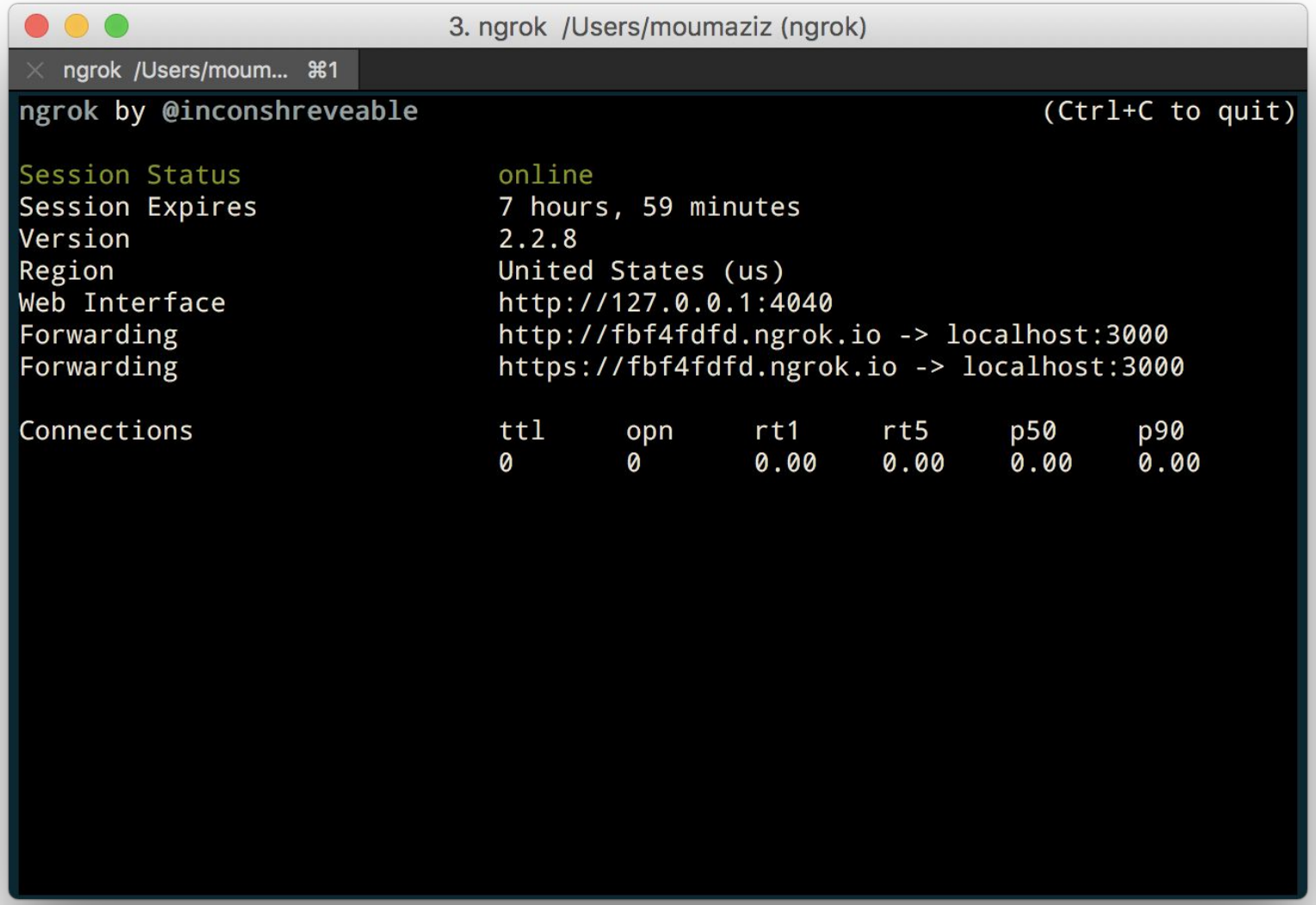


Outils: Ngrok

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

https://ngrok.com

Outils: Ngrok



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 avions vu ça précedemment 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: 4afe3fe836115021358477 59
  "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 appel 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** créé 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()`

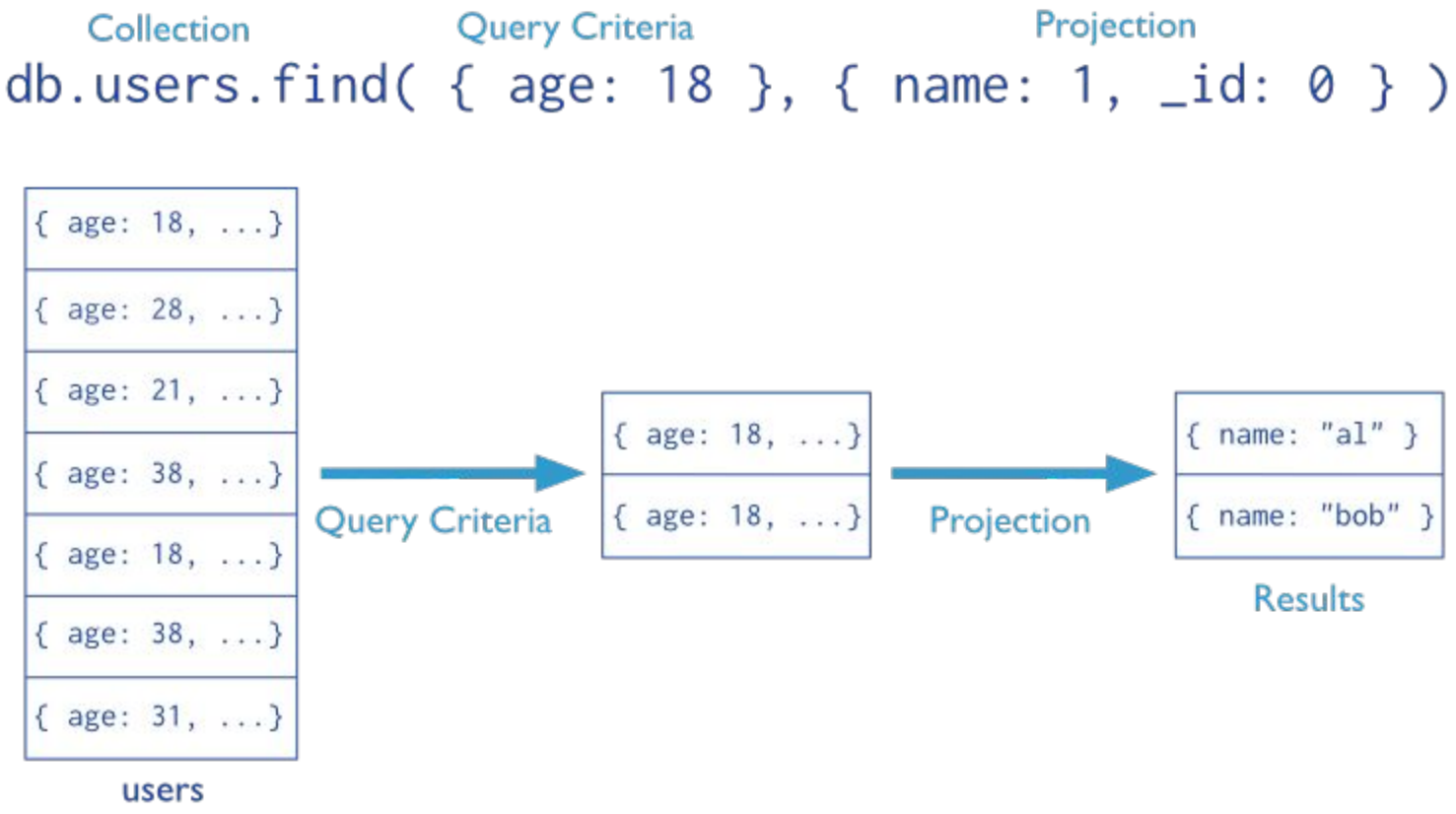
Fonctionne de la même manière que **`findOne()`** à l'exception qu'elle nous retourne non pas un document mais un **curseur** qui pointe sur le **premier** document.

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

```
db.users.find(                          <-- collection
   { age: { $gt: 18 } },                <-- query criteria
   { name: 1, address: 1 }              <-- projection
).limit(5)                              <-- cursor modifier
```

```
SELECT _id, name, address   <-- projection
FROM   users                <-- table
WHERE  age > 18             <-- select criteria
LIMIT  5                    <-- cursor modifier
```

Requêtes et projections



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

copy

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

copy

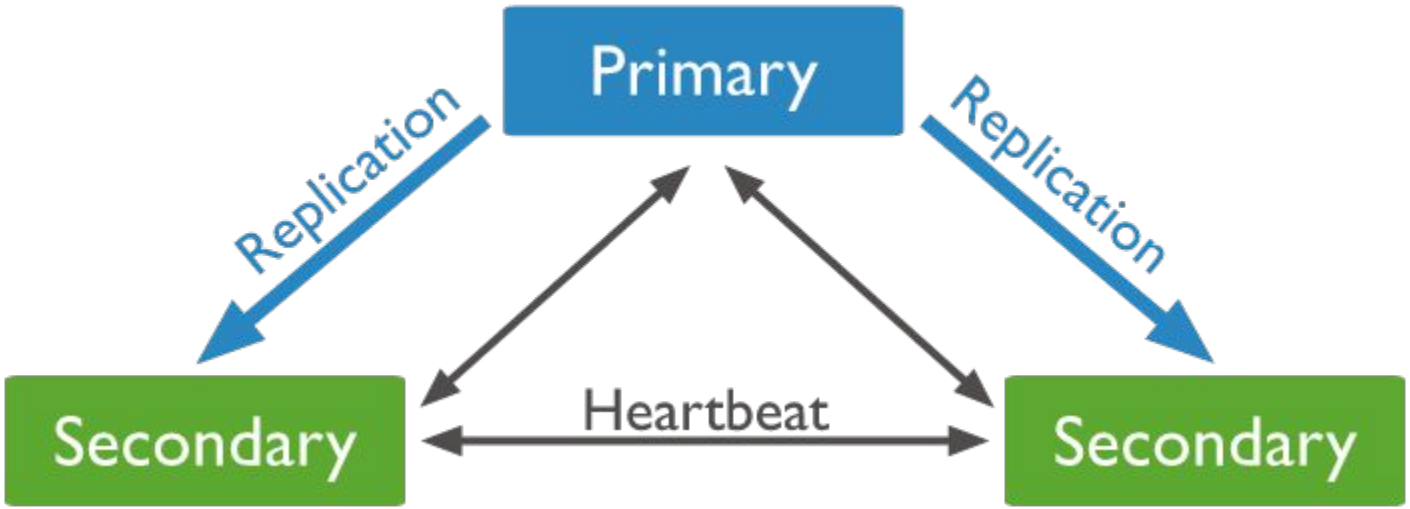
Optimisation: indexes

```
// 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 executée 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



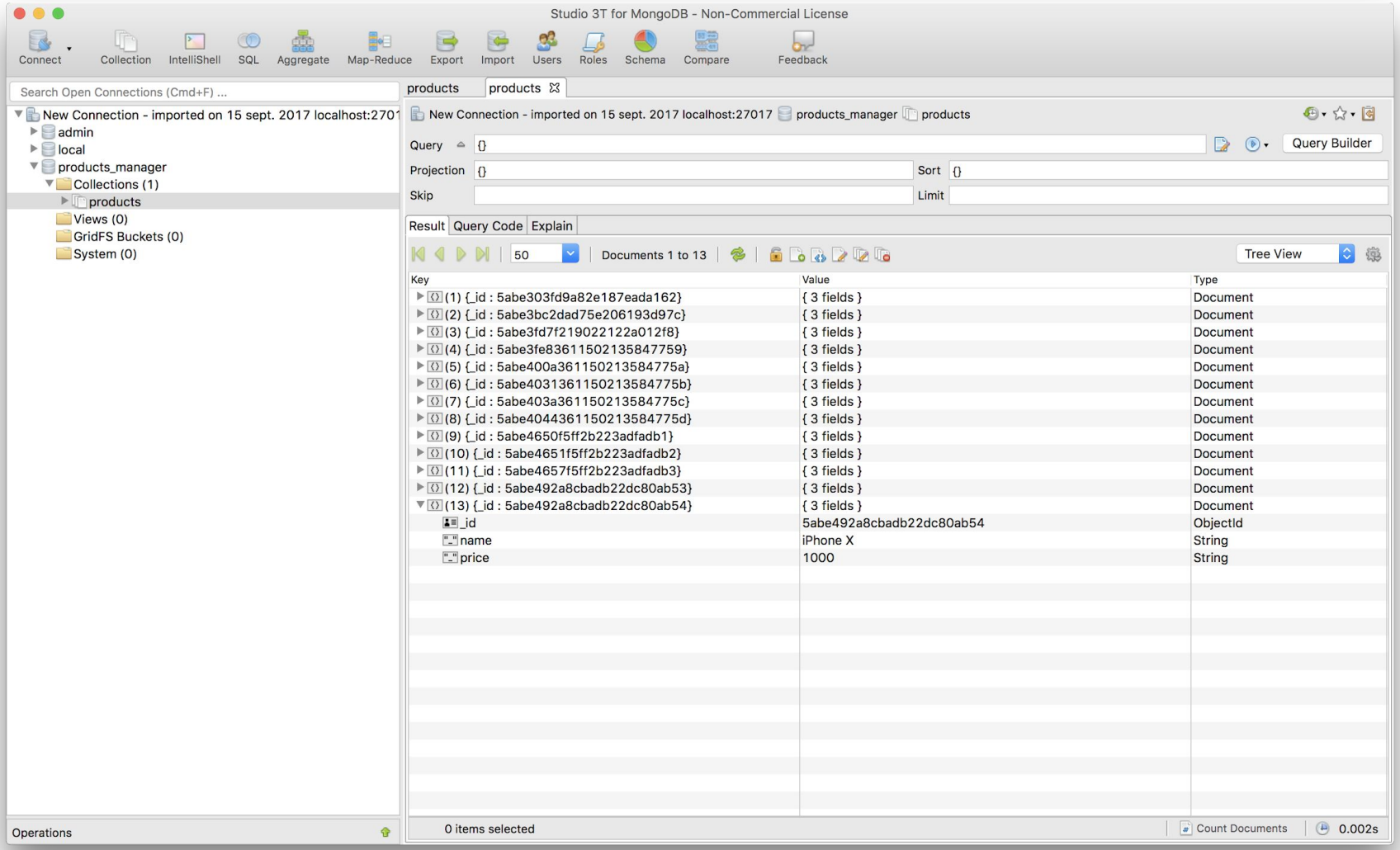
Utiliser MongoDB avec ExpressJS

```
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

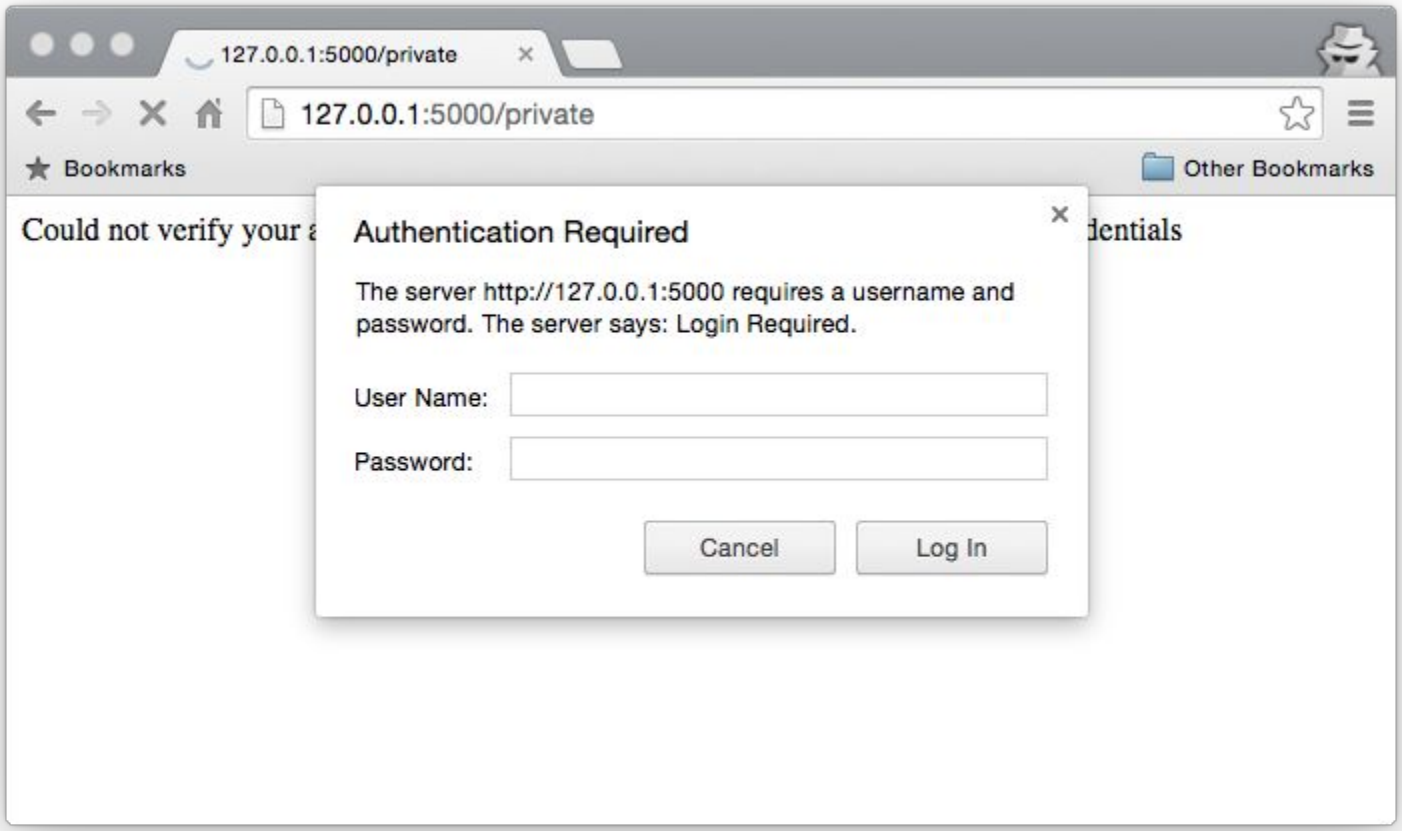
```
let db = null;
function onConnected(err, database) {
 db = database;
 app.get(“/”, (req, res) => {
   db.collection(“products”).find({}, (err, items)
=> { res.json(items) })
 })
}

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

MongoDB: Studio 3T



Authentification

Types d’authentification: Basic Auth



Types d’authentification: Basic Auth

L’authentification peut se faire deux manières différentes:

- Directement dans l’URL:

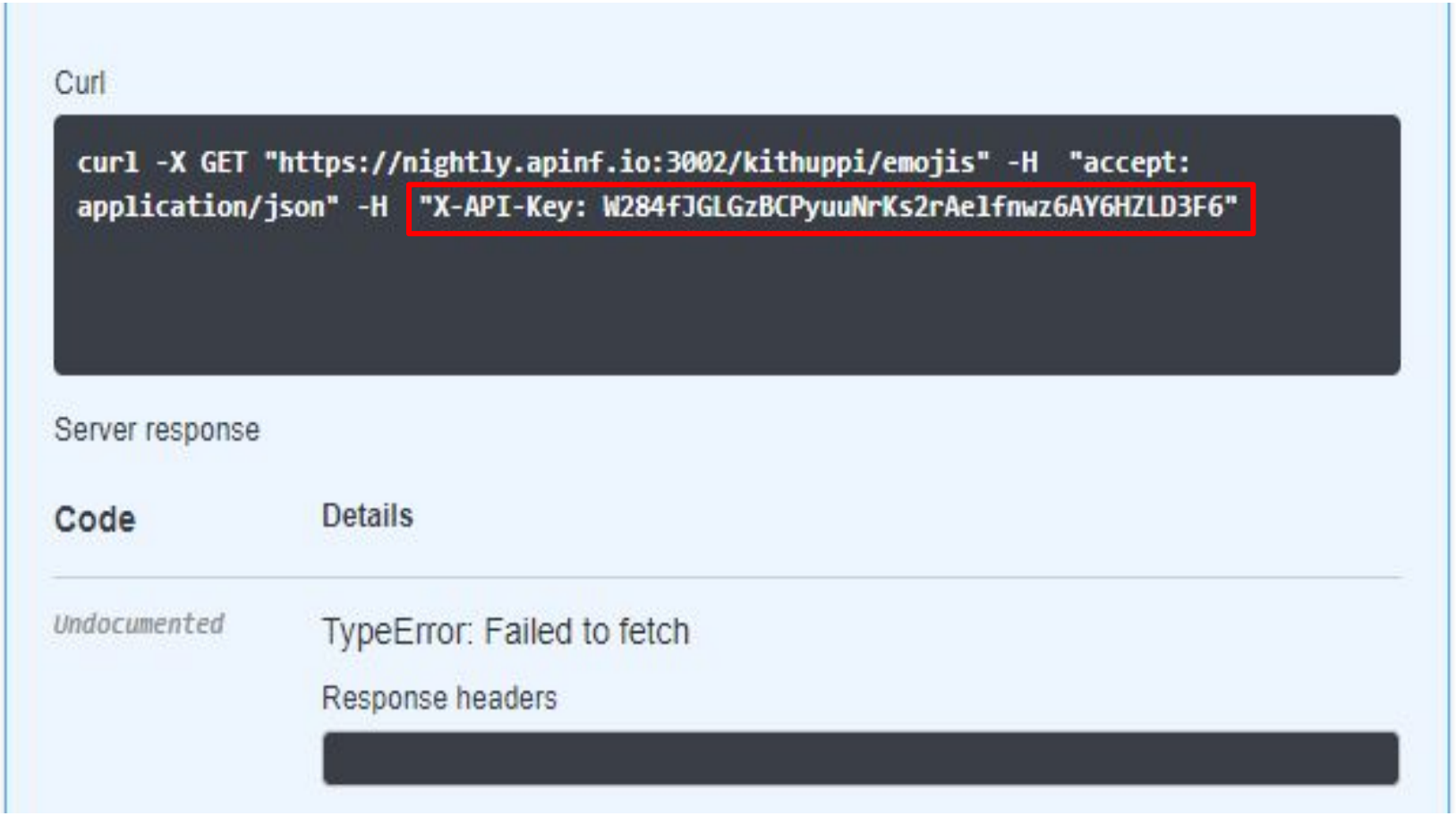
  http://toto:password@example.com

- En tant que header:

  Authorization: Basic AZIBAFJRFjpPcGVuU84JFN1I

Types d’authentification: API Key

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



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'**agorithme 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.