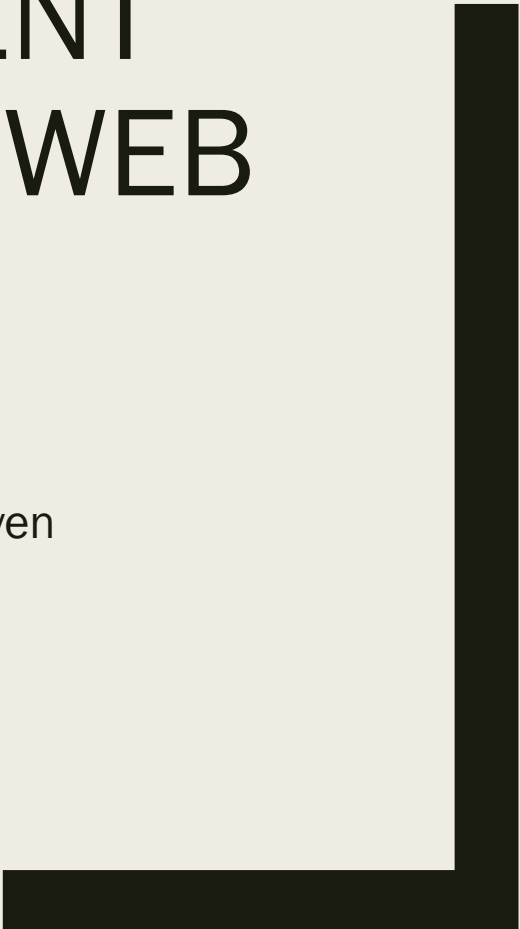




DÉVELOPPEMENT D'APPLICATIONS WEB

COURS 9: SERVEUR

Enseignante: NGUYEN Thi Minh Tuyen



Plan du cours

1. Serveur
2. NodeJS
3. ExpressJS
4. package.json
5. Paramètres de la route
6. Renvoi de JSON du serveur

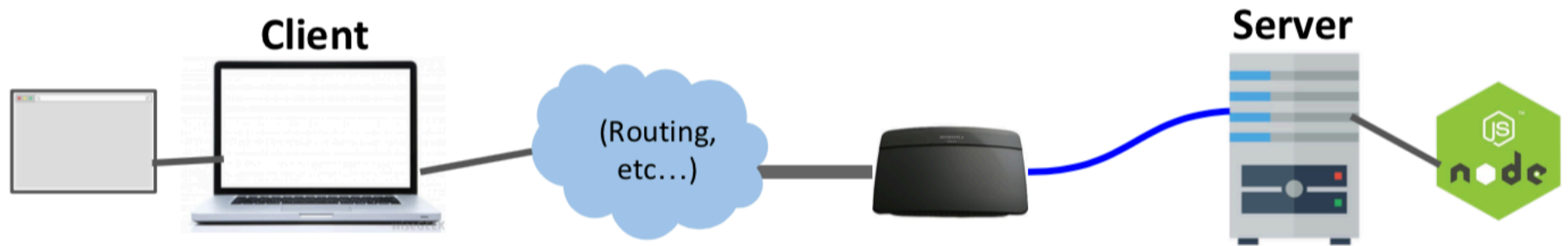
Plan du cours

1. **Serveur**
2. NodeJS
3. ExpressJS
4. package.json
5. Paramètres de la route
6. Renvoi de JSON du serveur

Programmation côté serveur

- Le code que nous écrivons est exécuté sur un serveur.
- Les serveurs:
 - *Sont des ordinateurs,*
 - *Exécutent des programmes pour générer des pages Web et autres ressources Web.*

Comment fonctionnent les clients et les serveurs ?



Coté serveur

La définition du serveur est surchargée:

- Parfois, "serveur" désigne la machine / ordinateur qui exécute le logiciel serveur.
- Parfois, "serveur" désigne le logiciel exécuté sur la machine/l'ordinateur.

Sockets

- Lorsque le serveur est exécuté pour la première fois, il exécute du code pour créer un socket qui lui permet de recevoir des messages entrants à partir du système d'exploitation.
- Un socket est un canal de communication. Vous pouvez envoyer et recevoir des données sur des sockets.
- NodeJS fera abstraction de cela afin que nous n'ayons pas à penser aux sockets.

Plan du cours

1. Serveur
2. **NodeJS**
3. ExpressJS
4. package.json
5. Paramètres de la route
6. Renvoi de JSON du serveur

NodeJS

NodeJS:

- Un runtime JavaScript écrit en C++.
- Peut interpréter et exécuter JavaScript.
- Inclut le support pour l'API NodeJS.

API NodeJS:

- Un ensemble de bibliothèques JavaScript utiles pour la création de programmes serveur.

V8 (de Chrome):

- L'interpréteur JavaScript ("moteur") utilisé par NodeJS pour interpréter, compiler et exécuter du code JavaScript.

Chrome

Chrome:

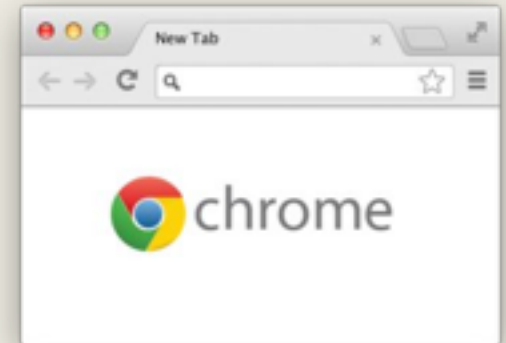
- Un navigateur écrit en C ++.
- Peut interpréter et exécuter du code JavaScript.
- Inclut la prise en charge des API DOM.

API DOM:

- Des bibliothèques JavaScript pour interagir avec une page Web

V8:

- L'interpréteur JavaScript ("moteur") utilisé par Chrome pour interpréter, compiler et exécuter du code JavaScript.



Chrome, V8, DOM



Parseur

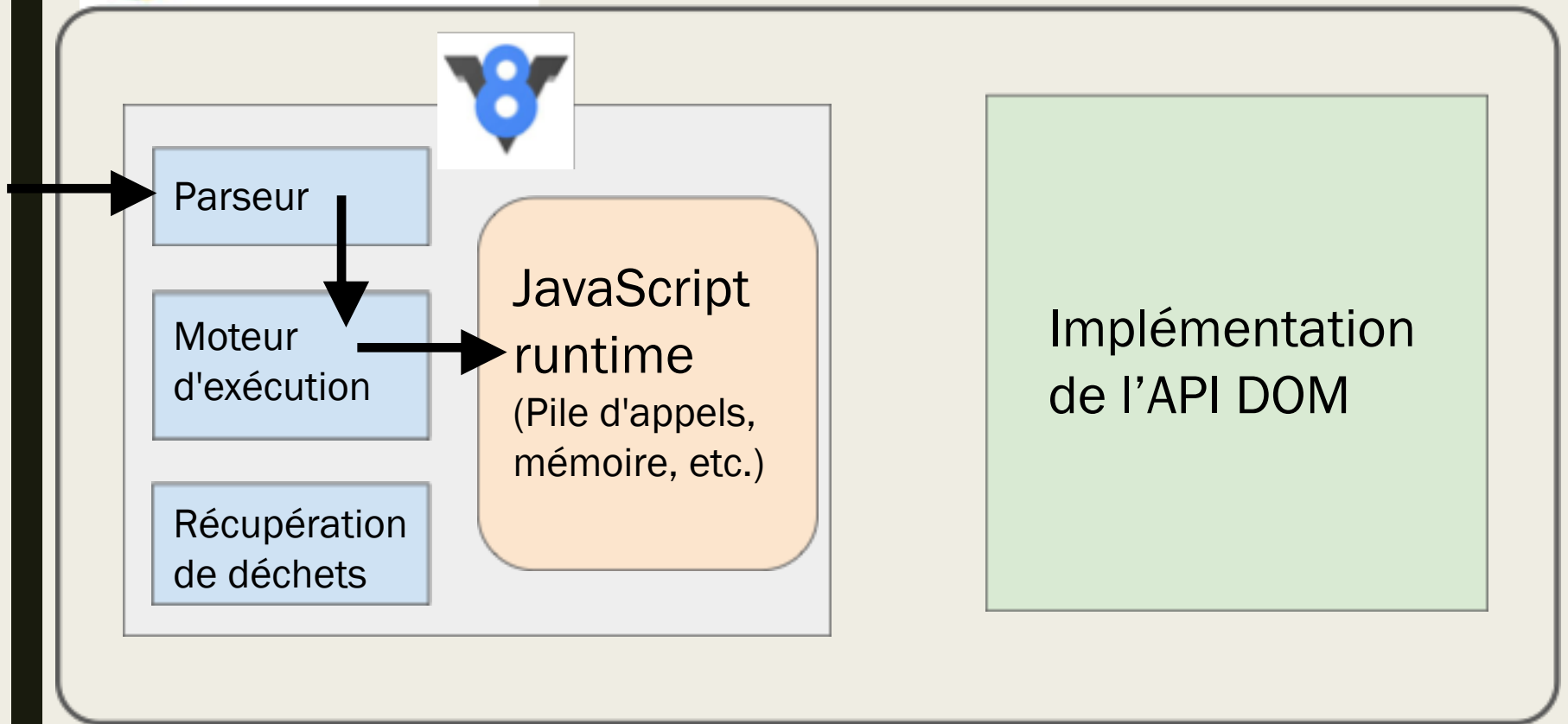
Moteur
d'exécution

Récupération
de déchets

JavaScript
runtime
(Pile d'appels,
mémoire, etc.)

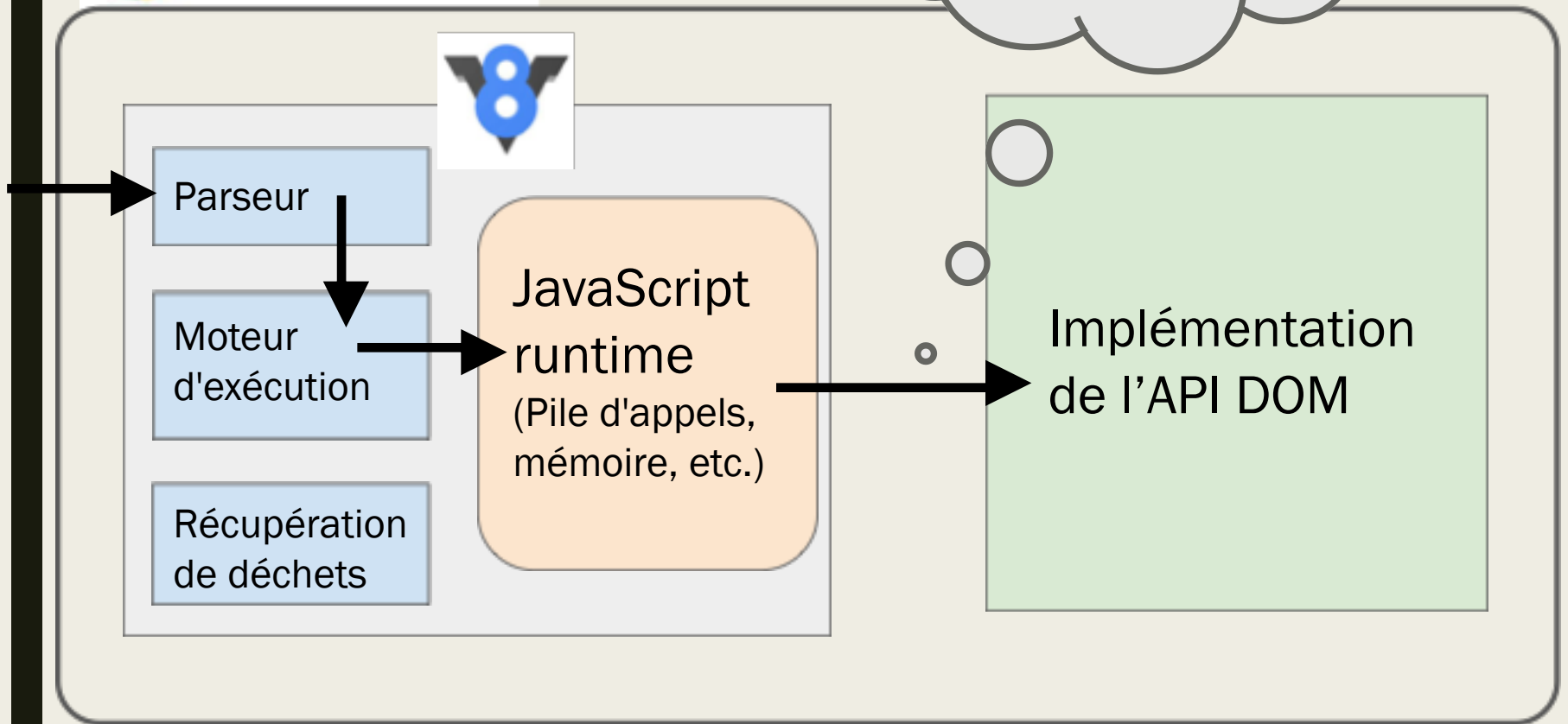
Implémentation
de l'API DOM

Chrome, V8, DOM



```
const name = 'V8';
```

Chrome, V8, DOM



`console.log('V8');`

NodeJS, V8, APIs NodeJS



Parseur

Moteur
d'exécution

Récupération
de déchets

JavaScript
runtime
(Pile d'appels,
mémoire, etc.)

Implémentation
de l'API NodeJS

NodeJS, V8, APIs NodeJS



Parseur

Moteur
d'exécution

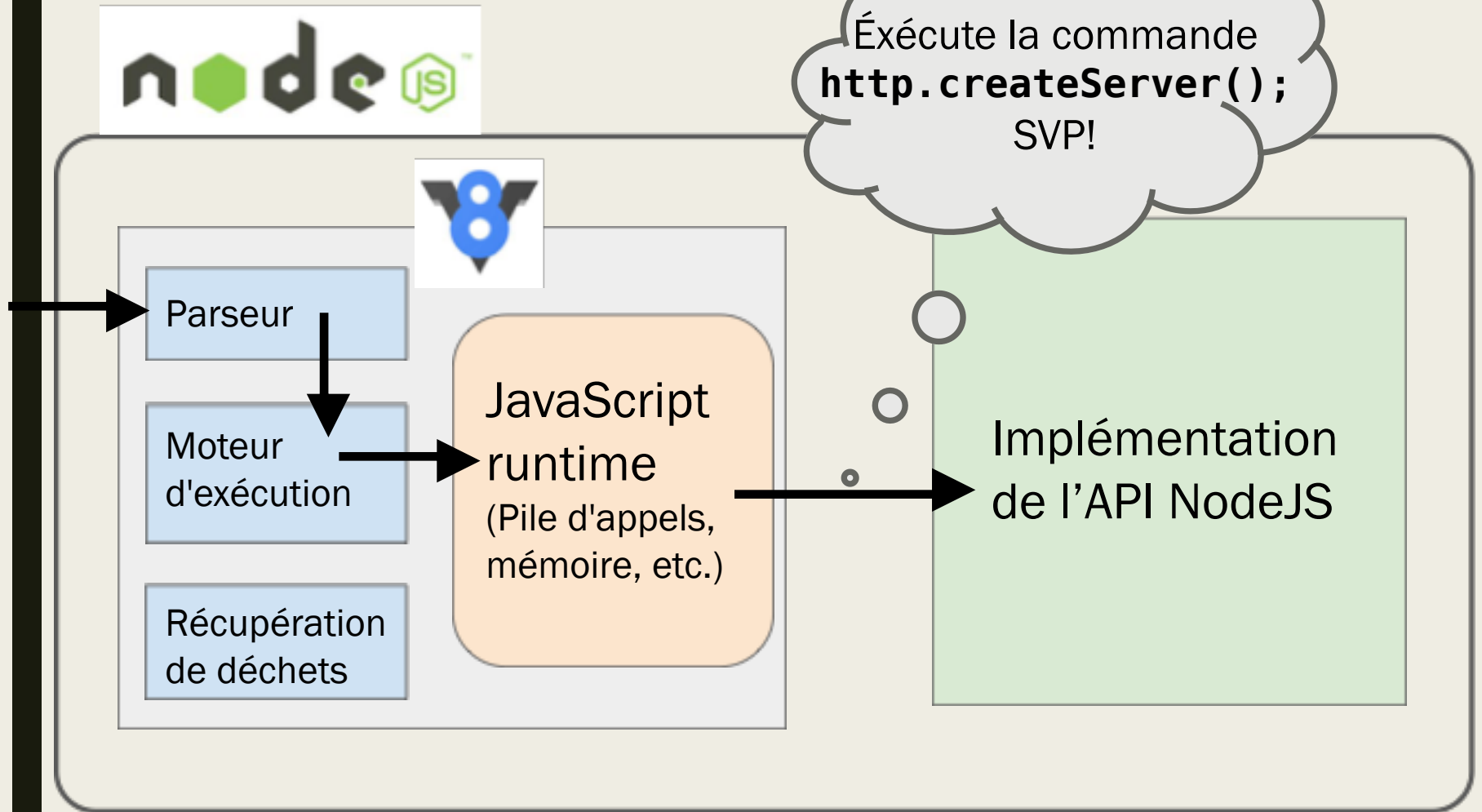
Récupération
de déchets

JavaScript
runtime
(Pile d'appels,
mémoire, etc.)

Implémentation
de l'API NodeJS

```
const x = 15;  
x++;
```

NodeJS, V8, APIs NodeJS



`http.createServer();`

NodeJS, V8, APIs NodeJS



Parseur

Moteur
d'exécution

Récupération
de déchets

JavaScript
runtime
(Pile d'appels,
mémoire, etc.)

Implémentation
de l'API NodeJS

Si nous essayons d'appeler **`document.querySelector ('div');`**
dans le runtime NodeJS?

NodeJS, V8, APIs NodeJS



Parseur

Moteur
d'exécution

Récupération
de déchets

JavaScript
runtime
(Pile d'appels,
mémoire, etc.)

Implémentation
de l'API NodeJS

```
document.querySelector('div');
```

ReferenceError: document is not defined

NodeJS, V8, APIs NodeJS



Parseur

Moteur
d'exécution

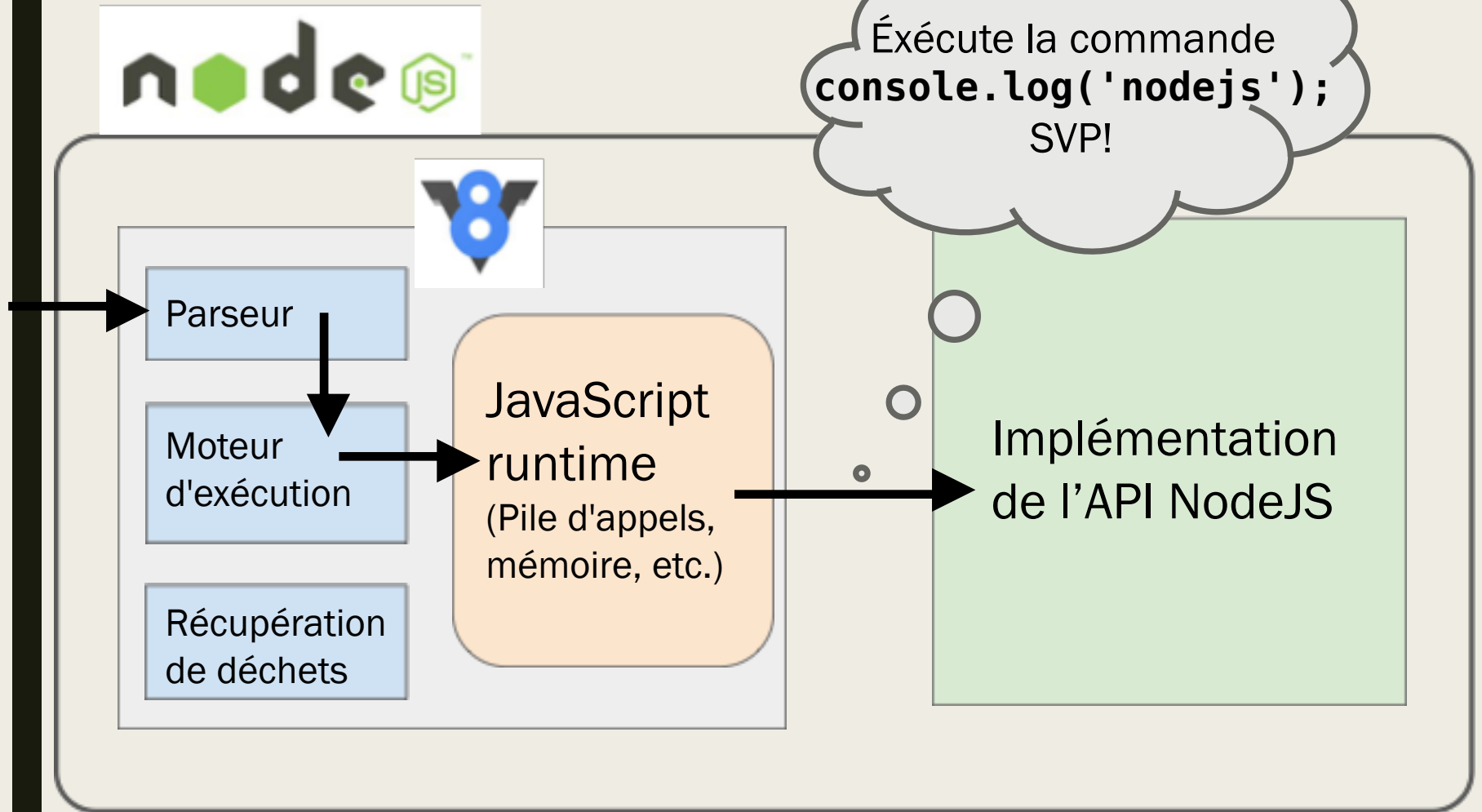
Récupération
de déchets

JavaScript
runtime
(Pile d'appels,
mémoire, etc.)

Implémentation
de l'API NodeJS

Si nous essayons d'appeler **`console.log('nodejs');`** dans le runtime NodeJS?

NodeJS, V8, APIs NodeJS



`console.log('nodejs');`

(L'API NodeJS a implémenté son propre `console.log`)

Installation

- Téléchargement le paquet et l'installation de NodeJS:

<https://nodejs.org/fr/download/>

- Documentation:

<https://nodejs.org/fr/docs/>

Commande node

- Similaire à la console JavaScript dans Chrome ou lorsque vous exécutez Python.

- Exemple:

```
$ node
```

```
> let x = 5;
```

```
undefined
```

```
> x++;
```

```
5
```

```
> x;
```

```
6
```

```
>
```

NodeJS

- NodeJS peut être utilisé pour écrire des scripts en JavaScript, sans aucun lien avec les serveurs.

```
function printPoem() {  
  console.log('Roses are red,');  
  console.log('Violets are blue,');  
  console.log('Sugar is sweet,');  
  console.log('And so are you.');
```

```
  console.log();  
}  
  
printPoem();  
printPoem();
```

Commande node

- La commande `node` peut être utilisée pour exécuter un fichier JS avec la syntaxe: `node nomFichier`

- Exemple:

```
$ node simplescript.js
```

```
Roses are red,  
Violets are blue,  
Sugar is sweet,  
And so are you.
```

```
Roses are red,  
Violets are blue,  
Sugar is sweet,  
And so are you.
```


Node pour les serveurs

- Voici un serveur très basique écrit pour NodeJS:

```
const http=require('http');
const server=http.createServer();
server.on('request',function(req,res){
    res.statusCode =200;
    res.setHeader('Content-Type','text/plain');
    res.end('Hello world from node\n');
});
server.on('listening', function(){
    console.log('server is running');
});
server.listen(3000);
```

(ATTENTION: nous n'écrirons pas de serveurs comme celui-ci!
Nous allons utiliser ExpressJS pour vous aider, à apprendre plus tard.

require() [1]

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

```
const server=http.createServer();
```

- **require()** de NodeJS charge un module, similaire à **import** en Java ou à **include** en C ++.
- Nous pouvons **require()** des modules inclus avec NodeJS, ou des modules que nous avons écrits nous-mêmes.
- Dans cet exemple, "**http**" fait référence au module HTTP NodeJS.

require() [2]

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

```
const server=http.createServer();
```

- La variable **http** renvoyée par **require('http')** peut être utilisée pour passer des appels à l'API HTTP.
- **http.createServer()** crée un objet Server.

Emitter.on [1]

```
server.on('request', function(req, res){  
    res.statusCode = 200;  
    res.setHeader('Content-Type', 'text/plain');  
    res.end('Hello world from node\n');  
});  
server.on('listening', function(){  
    console.log('server is running');  
});  
server.listen(3000);
```

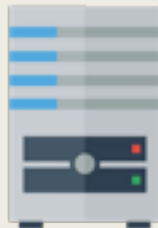
La fonction **on()** est l'équivalent de **addEventListener**.

Emitter.on [2]

```
server.on('request', function(req, res){  
  res.statusCode = 200;  
  res.setHeader('Content-Type', 'text/plain');  
  res.end('Hello world from node\n');  
});
```

L'événement **request** est émis à chaque nouvelle requête HTTP à traiter par le programme NodeJS.

Server



Emitter.on [3]

```
server.on('request', function(req, res){  
    res.statusCode = 200;  
    res.setHeader('Content-Type', 'text/plain');  
    res.end('Hello world from node\n');  
});
```

Le paramètre **req** donne des informations sur la requête entrante et
Le paramètre **res** est le paramètre de réponse que nous écrivons via
des appels de méthode.

- **statusCode** : définit le code d'état HTTP.
- **setHeader()** : Définit les en-têtes HTTP.
- **end()** : écrit le message dans le corps de la réponse puis signale au serveur que le message est complet.

listen() et listening

```
server.on('listening', function(){  
    console.log('server is running');  
});  
server.listen(3000);
```

La fonction **listen()** commence à accepter les connexions sur le numéro de port indiqué.

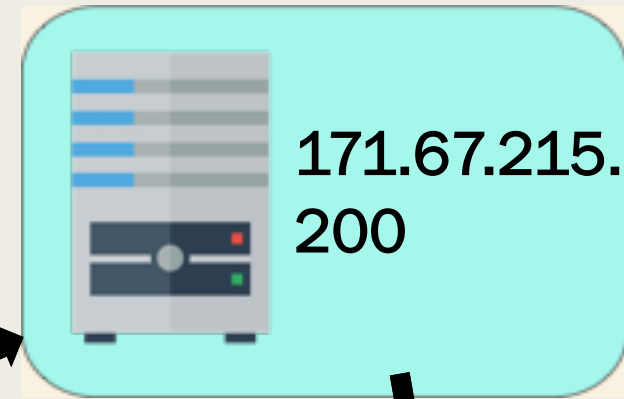
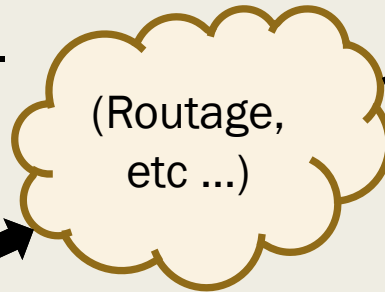
L'événement **listening** sera émis lorsque le serveur aura été lié à un port.

Q: Qu'est-ce qu'un port? Qu'est-ce que c'est binding?

Ports et binding

- **port:** dans le contexte de la mise en réseau, un lieu de connexion "logique" (par opposition à un lieu physique)
 - *Un nombre compris entre 0 et 65535 (entier non signé de 16 bits)*
- Lorsque vous démarrez un processus serveur, vous indiquez au système d'exploitation le numéro de port à lui associer. Ceci s'appelle **binding**.

Le système d'exploitation ouvre une connexion TCP sur le port **80** de l'ordinateur au **171.67.215.200**.



Le processus serveur s'exécutant sur le port **80** répond aux demandes.



Une connexion TCP nécessite une adresse IP et un numéro de port.

- Si aucun numéro de port n'est spécifié, **80** est la valeur par défaut pour les demandes HTTP.

Ports par défaut

Il existe de nombreux ports connus, c'est-à-dire les ports qui seront utilisés par défaut pour des protocoles particuliers:

- 21: File Transfer Protocol (FTP)
- 22: Secure Shell (SSH)
- 23: Telnet remote login service
- 25: Simple Mail Transfer Protocol (SMTP)
- 53: Domain Name System (DNS) service
- 80: Hypertext Transfer Protocol (HTTP) utilisé dans World Wide Web
- 110: Post Office Protocol (POP3)
- 119: Network News Transfer Protocol (NNTP)
- 123: Network Time Protocol (NTP)
- 143: Internet Message Access Protocol (IMAP)
- 161: Simple Network Management Protocol (SNMP)
- 194: Internet Relay Chat (IRC)
- 443: HTTP Secure (HTTPS)

Serveur de développement

```
server.on('listening', function(){  
    console.log('server is running');  
});  
server.listen(3000);
```

Pour notre serveur de développement, nous pouvons choisir le numéro de port que nous voulons. Dans cet exemple, nous avons choisi **3000**.

Lancement d'un serveur

- Lorsque nous exécutons `node server.js` dans le terminal, nous allons voir comme suit:

```
$ node server.js  
server is running
```

Le processus ne se termine pas après l'exécution de la commande, car il attend maintenant les demandes HTTP sur le port 3000.

Q: Comment envoyons-nous une requête HTTP sur le port 3000?

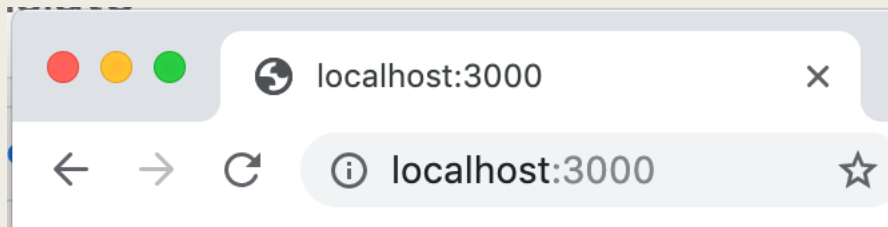
Localhost

- Nous pouvons envoyer une requête HTTP GET s'exécutant sur l'un des ports de l'ordinateur local à l'aide de l'URL:

`http://localhost:portNumber`

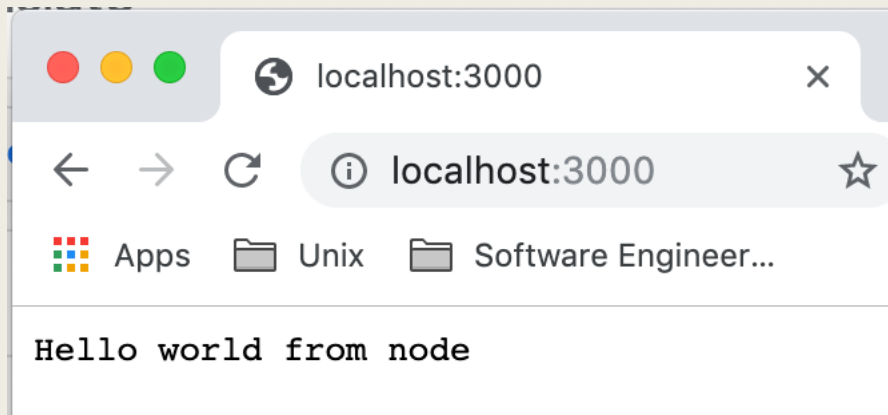
`http://localhost:3000`

- **Localhost** est un nom d'hôte qui signifie "cet ordinateur".



Réponse du serveur

- Voici le résultat de la demande adressée à notre serveur HTTP:



Node pour les serveurs

```
const http=require('http');
const server=http.createServer();
server.on('request',function(req,res){
    res.statusCode =200;
    res.setHeader('Content-Type','text/plain');
    res.end('Hello world from node\n');
});
server.on('listening', function(){
    console.log('server is running');
});
server.listen(3000);
```

Ce serveur renvoie la même réponse, quelle que soit la demande.

Node pour les serveurs

```
var http = require('http');

http.createServer(function(request, response) {
  var headers = request.headers;
  var method = request.method;
  var url = request.url;
  var body = [];
  request.on('error', function(err) {
    console.error(err);
  }).on('data', function(chunk) {
    body.push(chunk);
  }).on('end', function() {
    body = Buffer.concat(body).toString();
    // BEGINNING OF NEW STUFF

    response.on('error', function(err) {
      console.error(err);
    });

    response.statusCode = 200;
    response.setHeader('Content-Type', 'application/json');
    // Note: the 2 lines above could be replaced with this next one:
    // response.writeHead(200, {'Content-Type': 'application/json'})

    var responseBody = {
      headers: headers,
      method: method,
      url: url,
      body: body
    };
  });
```

Les API de serveur NodeJS sont en fait assez bas niveau:

- Vous construisez la demande manuellement
- Vous écrivez la réponse manuellement
- Il y a beaucoup de code de traitement fastidieux

Plan du cours

1. Serveur
2. NodeJS
3. **ExpressJS**
4. package.json
5. Paramètres de la route
6. Renvoi de JSON du serveur

ExpressJS

- Nous allons utiliser une bibliothèque appelée ExpressJS au-dessus de NodeJS:

```
const express = require('express');  
const app = express();  
app.get('/', function(req, res){  
  res.send('Hello world!');  
});
```

```
app.listen(3000, function(){  
  console.log('Example app listening on port 3000.');
```

ExpressJS

- **Express** ne fait pas partie des API NodeJS. Si nous essayons de l'utiliser comme ceci, nous aurons une erreur:

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

```
    throw err;  
    ^  
  
Error: Cannot find module 'express'  
    at Function.Module._resolveFilename (node:internal/modules/cjs/loader:1028:15)  
    at Function.Module._load (node:internal/modules/cjs/loader:873:27)  
    at Module.require (node:internal/modules/cjs/loader:1057:15)  
    at require (node:internal/modules/cjs/helpers:103:18)  
    at Object.<anonymous> (index.js:1:17)  
    at Module._compile (node:internal/modules/cjs/loader:1073:14)  
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1123:10)  
    at Module.load (node:internal/modules/cjs/loader:939:32)  
    at Function.Module._load (node:internal/modules/cjs/loader:758:12)  
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:83:12)  
    at node:internal/main/run_main_module:22:47
```

- Nous devons installer **Express** via **npm**.
- Document de la page expressjs:
<https://expressjs.com/fr/starter/installing.html>
- A savoir plus: <https://expressjs.com/fr/guide/routing.html>

npm

- Lorsque vous installez NodeJS, vous installez également **npm**:
- **npm**: Node Package Manager *: Outil de ligne de commande permettant d'installer des packages (bibliothèques et outils) écrits en JavaScript et compatibles avec NodeJS.
- Peut trouver des paquets dans le référentiel en ligne:

<https://www.npmjs.com/>



npm install et uninstall

`npm install nom-paquet`

- Ceci télécharge la bibliothèque **nom-paquet** dans un répertoire **node_modules**.
- La bibliothèque **nom-paquet** peut maintenant être incluse dans votre fichiers JavaScript NodeJS.

`npm uninstall nom-paquet`

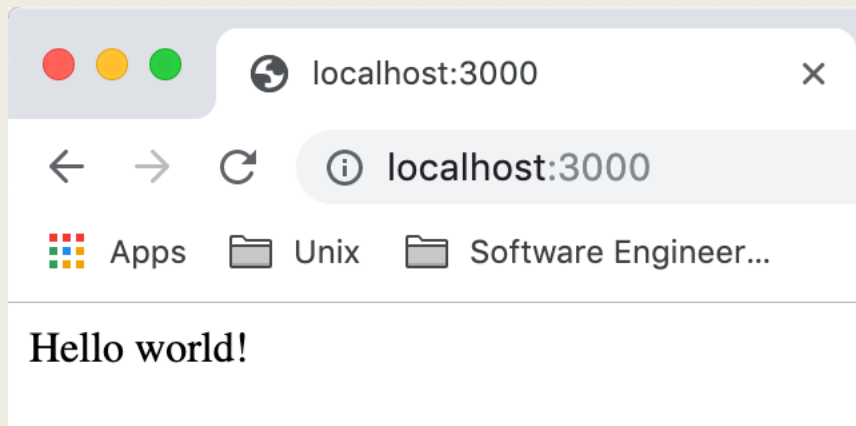
- Ceci supprime la bibliothèque **nom-paquet** du répertoire **node_modules**, en supprimant le répertoire si nécessaire

Exemple de Express

```
$ npm install express
```

```
$ node serverwithexprss.js
```

Example app listening on port 3000.



Routes de Express [1]

- Vous pouvez spécifier des routes dans Express:

```
app.get('/', function(req, res){  
    res.send('Main page!');  
});  
  
app.get('/hello', function(req, res){  
    res.send('GET hello!');  
});  
  
app.post('/hello', function(req, res){  
    res.send('POST hello!');  
});
```

Routes de Express [2]

```
app.get('/hello', function(req, res){  
    res.send('GET hello!');  
});
```

app.method(path, handler)

- Spécifie comment le serveur doit gérer la méthode HTTP demandes faites à l'URL **/path**
- Cet exemple dit:

Quand il y a une requête GET à `http://localhost:3000/hello`, répondez par le texte "GET hello!"

Paramètres de gestionnaire

- Gestionnaire(handler)

```
app.get( '/hello', function( req, res ) {  
    res.send( 'GET hello!' );  
});
```

Express a ses propres objets **Request** et **Response**:

- **req** est un objet **Request**
- **res** est un objet **Response**
- **res.send()** envoie une réponse HTTP avec la donnée contenu
- Envoie le type de contenu "**text/html**" par défaut

Interroger notre serveur

Voici trois manières d'envoyer des requêtes HTTP à notre serveur:

1. Accédez à `http://localhost:3000/<chemin>` dans notre navigateur
 - Ne peut faire que des requêtes GET
2. Appelez la fonction **fetch()** dans la page Web
 - *Nous avons déjà envoyé des requêtes GET, mais nous pouvons envoyer n'importe quel type de requête HTTP*
3. Utilisez l'outil Postman
 - *Outil de débogage*

fetch() à localhost

- Si nous essayons d'extraire localhost du `file:///`
`fetch('http://localhost:3000')`
 `.then(onResponse)`
 `.then(onTextReady);`
- Nous obtenons cette erreur CORS:

```
✖ Access to fetch at 'http://localhost:3000/' from index.html:1  
origin 'http://127.0.0.1:5500' has been blocked by CORS  
policy: No 'Access-Control-Allow-Origin' header is present on  
the requested resource. If an opaque response serves your  
needs, set the request's mode to 'no-cors' to fetch the  
resource with CORS disabled.  
✖ Uncaught (in promise) TypeError: Failed to fetch index.html:1
```

CORS

- Cross-Origin Resource Sharing (partage de ressources entre origines multiples).
- Stratégies de navigateur pour indiquer quelles ressources qu'une page Web peut charger.
- Nous ne pouvons pas faire de requêtes d'origine croisée par défaut pour des ressources chargées via `fetch()`.
- Le problème est que nous essayons de `fetch()` avec `http://localhost:3000` depuis `file:///`
 - *Car les deux ressources ont des origines différentes, ceci est interdit par la stratégie CORS par défaut.*

Solution 1: Activer CORS

```
app.get('/', function(req, res){  
    res.header("Access-Control-Allow-Origin", "*");  
    res.send('Main page!');  
});
```

Solution 2: Données statiques du serveur

- Nous pouvons à la place servir notre HTML/CSS/JS de manière statique à partir du même serveur:

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

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

```
app.use(express.static('public'))
```

```
app.get('/', function(req, res){
```

```
res.send('Main page!');
```

```
});
```

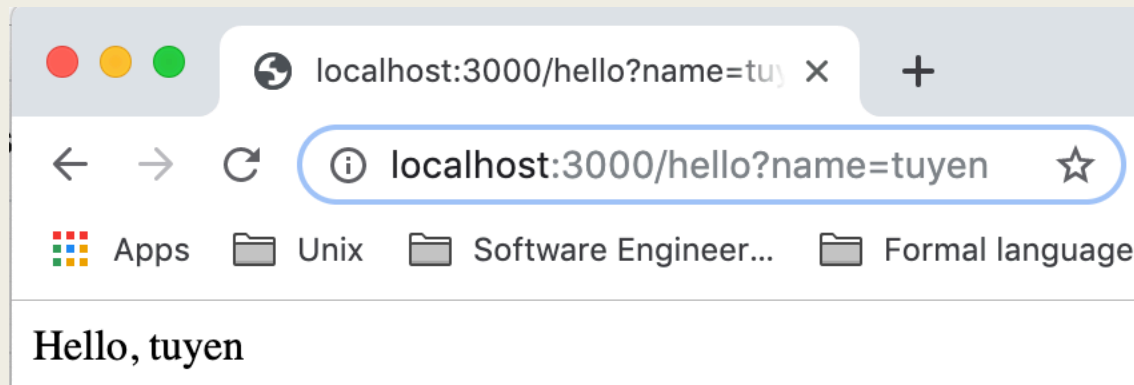
```
example2
└─ node_modules
   └─ public
      <> index.html
      JS myscript.js
      # stylesheet.css
      {} package-lock.json
      JS server.js
```

Paramètres de requête GET dans Express [1]

```
app.get('/hello', function(req, res){  
  const queryParams = req.query;  
  console.log(queryParams);  
  const name = req.query.name;  
  res.send('Hello, ' + name);  
});
```

Les paramètres de requête sont enregistrés dans **req.query**.

Paramètres de requête GET dans Express [2]



fetch () avec POST [1]

- Au côté serveur, nous définissons notre gestionnaire dans **app.post ()** pour gérer les requêtes POST.

```
app.post ( '/hello', function ( req, res ) {  
    res.send ( 'POST hello!' );  
});
```

fetch() avec POST [2]

Au côté client:

```
fetch('/hello',  
      {method: 'POST'})  
  .then(onResponse)  
  .then(onTextReady);  
  
function onResponse(response){  
  return response.text();  
}  
  
function onTextReady(text){  
  console.log(text);  
}
```

fetch() avec POST [2]

- Nous pouvons également envoyer des paramètres de requête via POST:

```
fetch('/hello?name=ntmtuyen',  
      {method: 'POST'})  
  .then(onResponse)  
  .then(onTextReady);
```

```
function onResponse(response){  
  return response.text();  
}
```

```
function onTextReady(text){  
  console.log(text);  
}
```

(ATTENTION: nous ne ferons pas de requêtes POST comme celle-ci!

Nous enverrons des données dans le corps de la requête au lieu de via des paramètres de requête.)

fetch () avec POST [3]

- Au côté serveur, ces paramètres sont accessibles de la même manière:

```
app.post('/hello', function(req, res){  
    const queryParams = req.query;  
    const name = queryParams.name;  
    res.send('POST hello,' + name);  
});
```

Corps du message de POST [1]

- En générale, le style d'envoi des données via des paramètres de requête est médiocre dans une requête POST.
- A la place, nous devrions spécifier un corps de message dans notre appel `fetch()`.

```
const message = { name: 'nguyen van an', email:
'nvan@gmail.com' };
const fetchOptions = { method: 'POST',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json' },
  body: JSON.stringify(message) };
fetch('/helloworld', fetchOptions)
.then(onResponse)
.then(onTextReady);
```

body-parser

- Nous pouvons utiliser la bibliothèque **body-parser** pour récupérer JSON .
- Ce n'est pas la bibliothèque d'API NodeJS, nous devons donc l'installer:

```
npm install body-parser
```

- Au côté serveur, ajoutez:

```
const bodyParser = require('body-parser');
```

```
const jsonParser = bodyParser.json();
```

- Cela crée un analyseur JSON stocké dans **jsonParser**, que nous pouvons ensuite transmettre aux routes dont nous voulons analyser le corps des messages en tant que json.

Corps du message de POST [2]

- Nous pouvons accéder le corps du message via **req.body**:

```
app.post('/helloparsed', jsonParser, function (req,
    res) {
    const body = req.body;
    const name = body.name;
    const email = body.email;
    res.send('POST: Name: ' + name + ', email: ' +
        email);
});
```

GET vs POST

- Utilisez la requête GET pour extraire des données, pas pour les écrire.
- Utilisez la requête POST pour écrire des données, sans les récupérer.
- Utilisez également des méthodes HTTP plus spécifiques:
 - *PATCH: met à jour la ressource spécifiée*
 - *DELETE: supprimer la ressource spécifiée*

Plan du cours

1. Serveur
2. NodeJS
3. ExpressJS
4. **package.json**
5. Paramètres de la route
6. Renvoi de JSON du serveur

Installation de dépendances

- Dans notre exemple, nous avons dû installer les paquets npm **express** et **body-parser**.

```
npm install express
```

```
npm install body-parser
```

- Ceux-ci sont écrits dans le répertoire **node_modules**.

Chargement du code serveur

- Lorsque vous chargez du code NodeJS dans un dépôt Github (ou dans un n'importe quel dépôt de code), vous ne devez pas charger le répertoire `node_modules`.
 - *vous ne devriez pas modifier le code dans le répertoire `node_modules`, il n'y a donc aucune raison de le placer sous contrôle de version.*
 - *Cela augmentera également la taille de votre dépôt de manière significative.*
- Q: Mais si vous ne chargez pas le répertoire `node_modules` dans votre dépôt de code, comment les utilisateurs sauront-ils quelles bibliothèques ils doivent installer?

Gérer les dépendances

- Si nous n'incluons pas le répertoire `node_modules` dans notre dépôt, nous devons en quelque sorte informer les autres utilisateurs des modules `npm` qu'ils doivent installer.
- `npm` fournit un mécanisme pour cela: `package.json`

package.json

- Vous pouvez placer un fichier nommé `package.json` dans le répertoire racine de votre projet NodeJS pour spécifier les métadonnées relatives à votre projet.
- Créez un fichier `package.json` à l'aide de la commande suivante:

```
npm init
```

Cela vous posera une série de questions de la commande, puis générera un fichier `package.json` basé sur vos réponses.

Exemple de package.json

```
{  
  "name": "example2",  
  "version": "1.0.0",  
  "description": "",  
  "main": "server.js",  
  "dependencies": {  
    "body-parser": "^1.19.0",  
    "express": "^4.17.1"  
  },  
  "devDependencies": {},  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1",  
    "start": "node server.js"  
  },  
  "author": "",  
  "license": "ISC"  
}
```

Enregistrement des dépendances dans package.json

- Lorsque vous installez des packages, vous devez passer le paramètre `--save`

```
$ npm install --save express
```

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

- Si vous supprimez le répertoire `node_modules`:

```
$ rm -rf node_modules
```

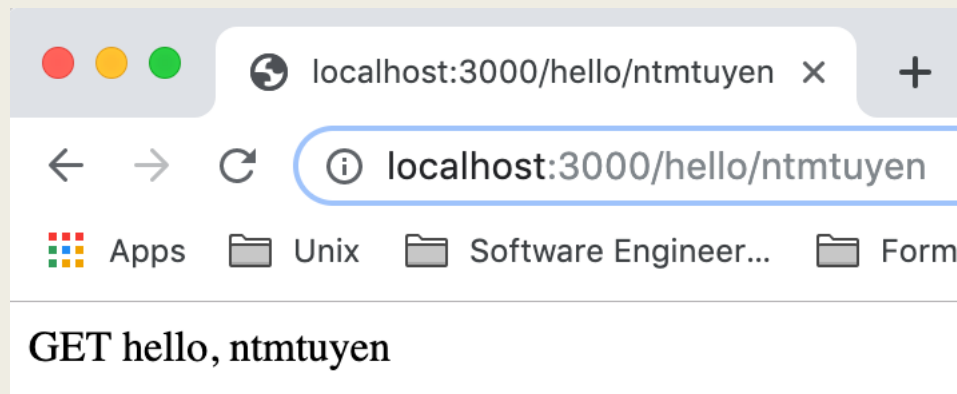
Plan du cours

1. Serveur
2. NodeJS
3. ExpressJS
4. package.json
- 5. Paramètres de la route**
6. Renvoi de JSON du serveur

Paramètres de la route [1]

- Nous pouvons utiliser la syntaxe `:nomVariable` dans le chemin pour spécifier un paramètre de route.
- Exemple:

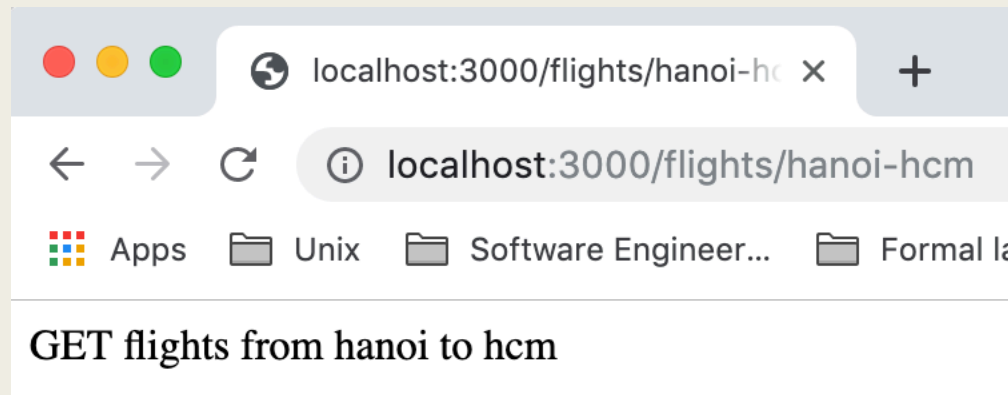
```
app.get('/hello/:name', function(req, res){  
    const routeParams = req.params;  
    const name = routeParams.name;  
    res.send('GET hello, ' + name);  
});
```



Paramètres de la route [2]

- Il est possible d'ajouter plusieurs paramètres dans un URL.
- Par exemple:

```
app.get('/flights/:from-:to', function(req, res){  
  const routeParams = req.params;  
  const from = routeParams.from;  
  const to = routeParams.to;  
  res.send('GET flights from ' + from + ' to ' + to);  
});
```



Plan du cours

1. Serveur
2. NodeJS
3. ExpressJS
4. package.json
5. Paramètres de la route
6. **Renvoi de JSON du serveur**

Réponse sous forme JSON

- Nous pouvons renvoyer une réponse sous la forme JSON en utilisant `res.json(object)` à la place:

```
app.get('/json', function(req, res){  
    const response = {  
        greeting: 'Hello world',  
        awesome: true  
    };  
    res.json(response);  
});
```

- Le paramètre que nous passons à `res.json()` devrait être un objet JavaScript.

Question ?