Créer un serveur avec Node.js

Objectifs:

- Comprendre les conventions du protocole HTTP dans le cadre du développement d'API
- Comprendre le fonctionnement du module `http` de Node.js.
- Implémenter quelques routes dynamiques

Rappels sur HTTP

URL

`https://localhost:5000/hello?name=Bob`

`protocol://hostname:port/pathname?query`

Documentation du module URL de node

Les "méthodes"

- `POST` -> Create
- `GET` -> Read
- `PUT` -> Udtate/Replace
- `PATCH` -> Update/modify
- `DELETE` -> Delete

Les "status codes"

- 2xx: Success (200: OK, 201: Created)
- 4xx: Client error (400: Bad Request, 404: Not Found)
- 5xx: Server error (500: Internal Server Error, 504: Gateway Timeout)

Un serveur HTTP avec Node.js

Le serveur le plus simple

`index.js`

```
import http from "http"

const server = http.createServer((req, res) => {
  res.end("Hello world")
})

server.listen("3000")
```

```
$ node index.js
```

Détaillons le code

```
// Importer le module "http", natif à node.js
import http from "http"

// `createServer` accepte en argument une function (le callback)
// qui est appelé avec les objets "request" et "response"
const server = http.createServer((req, res) => {
  res.end("Hello world")
})

// lance le serveur sur le port 3000
server.listen("3000")
```

Documentation officielle

L'objet "response"

```
// res.setHeader() permet d'ajouter des Header HTTP
res.setHeader('Content-Type', 'text/plain; charset=UTF-8')

// res.write() permet d'écrire le contenu de la réponse
res.write('Hello ')
res.write('world')

// par défaut, le code sera 200, mais il est possible de le changer
res.statusCode = 200

// res.end() permet d'envoyer la requête
res.end()

// il est aussi possible d'écrire le contenu directement dans .end()
res.end('Hello world')
```

Documentation officielle

L'objet "request"

L'objet `request` contient beaucoup d'informations sur la requête HTTP du client. Notamment :

- `req.method` contient la méthode HTTP de la requête

- `req.url` contient l'URL de la requête

Par exemple (fictif) :

```
if (req.method === 'GET') {
  if (req.url === '/hello') sayHello(req, res)
}
```

```
export function sayHello(req, res) {
  req.end('Hello')
}
```

Documentation officielle

Route et query

Node.js fournit un module natif pour parser les URL. Considérons la requête `GET http://localhost:3000/hello?name=Alice`

```
import { URL } from 'url'
// ....

// req.url === /hello?name=Alice
const url = new URL(req.url, `http://${req.headers.host}`)

console.log(url.searchParams.get('name')) // Alice
// or another way to do it
const query = Object.fromEntries(url.searchParams)
console.log(query.name) // Alice

// url.pathname contient la route
console.log(url.pathname) // '/hello'
```

Documentation officielle Objct.fromEntries()

POST / PUT / PATCH et le `body`

Avec les requêtes de type `POST / PUT / PATCH`, il est possible d'envoyer de la donnée dans le `body`. Cette data peut être sous différent formats. Nous allons nous intéresser au JSON.

Pour tester des requêtes POST, il est possible d'utiliser la commande `curl` ou bien des outils tels que Insomnia ou Postman

Exemple avec curl

```
curl -d '{"key1":"value1", "key2":"value2"}' \
  -H "Content-Type: application/json" \
  -X POST http://localhost:3000/data
```

Parser le `body`

La donnée est envoyée sous forme de chunks. Pour lire

```
async function parseBody(req) {
  const buffer = []

  for await (const chunk of req) {
    buffer.push(chunk)
  }

  const data = Buffer.concat(buffer).toString()

  return data
}
```

Gestion du JSON

Le JSON est un format de structure de données inspiré des object JavaScript.

```
{
  "name": "Alice",
  "age": 23
}
```

Pour transformer une chaîne de caractères JSON en un objet JS, on utilise `JSON.parse()`, et pour faire l'inverse, `JSON.stringify()`

```
const json = '{"name": "Alice", "age": 23}'
const obj = JSON.parse(json) // obj.name === "Alice" et obj.age === 23

JSON.stringify(obj) // '{"name": "Alice", "age": 23}'
```

Documentation MDN - Working with JSON Documentation MDN - parse / stringify

---

Gestion des erreurs

Deux aspects importants :

- Éviter que le serveur crash
- Fournir des messages d'erreur explicite au client

```
const json = '{"name": "Alice", "age": 23' // invalid json
try {
  const data = JSON.stringify(json)
  return data
} catch (error) {
  console.log(error)
  // handle error
  return null
}
```