NodeJs Tutrorial The NetNinja

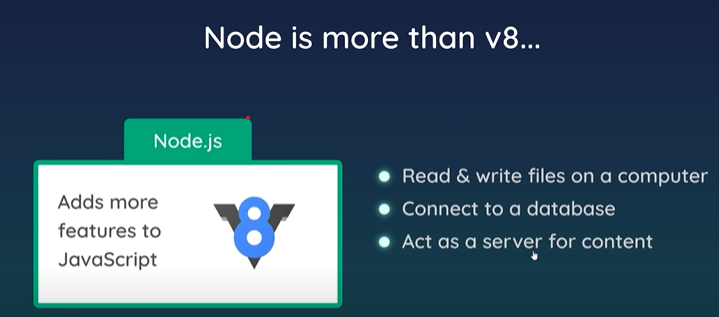
1. Introduction
   1. Introduction

Node.js est une plateforme logicielle open-source basée sur le moteur JavaScript V8 de Google(**V8 aussi nous aide a compiler le code Js sur le navigateur**). Elle permet l'exécution de code JavaScript côté serveur, permettant ainsi de développer des applications web et des services réseau performants et évolutifs.

Une image contenant diagramme

Description générée automatiquement

V8 nous aide a run le code Js sur notre serveur ou notre computer



Node.js a un avantage unique car des millions de développeurs frontaux qui écrivent du JavaScript pour le navigateur sont maintenant capables d'écrire le code côté serveur en plus du code côté client sans avoir à apprendre un langage complètement différent.

Dans Node.js, les nouvelles normes ECMAScript peuvent être utilisées sans problème, car vous n'avez pas besoin d'attendre que tous vos utilisateurs mettent à jour leurs navigateurs - vous avez la charge de décider quelle version ECMAScript utiliser en changeant la version de Node.js, et vous pouvez également activer des fonctionnalités expérimentales spécifiques en exécutant Node.js avec des drapeaux

* 1. Setup
     1. Pourquoi NodeJs

Une image contenant texte

Description générée automatiquement

* + 1. In this Course

Une image contenant texte

Description générée automatiquement

* + 1. Before you start

Une image contenant texte

Description générée automatiquement

1. NodeJs Basic
2. Global Object

Inside the browser the window object is the global object Example:(In the browser we can use set imeout inside window.setTimeout because Window is the global object )

In Node the global object is : global

* 1. Some function and properties and variable and …. In the Global object
* setTimeout
* setInterval
* \_\_dirname
* \_\_filename

// Global Object

global.setTimeout(()=>{

    console.log("in the timeout1")

},3000)

setTimeout(()=>{

    console.log("in the timeout2")

    // faite stopper l'interval int

    clearInterval(int)

},4000)

const int= setInterval(() => {

    console.log("in the interval")

}, 1000);

// Full absolute pth for this directory

console.log(\_\_dirname)

//Full absolute path for this fileName

console.log(\_\_filename)

Une image contenant texte

Description générée automatiquement

1. require

In Node.js, the require() function is used to import modules, which are separate pieces of code that can be reused in multiple files or projects.

When a module is required using require(), Node.js searches for the corresponding file on the file system and loads it into the current script. This allows the module to be used and its functions, objects or variables to be accessed from within the script.

People.js

const people = ["Houssam", "Mohcine", "Hamza", "Salah", "Yasser"];

const ages = [20, 25, 30, 35];

module.exports = { people, ages };

Modules.js

const xyz=require('./people')

const {people} =require('./people')

console.log("Test1",xyz)

console.log("Test2",people)

Une image contenant texte

Description générée automatiquement

1. the File System
   1. fs

In Node.js, the built-in **fs** (file system) module provides a way to work with the file system on the computer running the Node.js application.

Some of the most common uses of the **fs** module in Node.js include:

1. Reading and writing files: The **fs** module provides functions for reading and writing files, including creating, deleting, moving, and copying files.
2. Working with directories: The **fs** module includes functions for working with directories, such as creating and deleting directories, and listing the contents of a directory.
3. Watching for file changes: The **fs** module includes functions for watching files and directories for changes, allowing you to react to changes in real-time.
4. Changing file permissions: The **fs** module provides functions for changing file permissions, such as read, write, and execute permissions.

The most useful Function in fs

In Node.js, the **fs** (file system) module provides many useful functions for working with files and directories. Here are some of the most commonly used functions:

1. **fs.readFile()** - Asynchronously reads the entire contents of a file.
2. **fs.writeFile()** - Asynchronously writes data to a file, replacing the file if it already exists.
3. **fs.appendFile()** - Asynchronously appends data to a file, creating the file if it does not exist.
4. **fs.existsSync()** - Synchronously tests whether a file or directory exists.
5. **fs.mkdir()** - Asynchronously creates a new directory with the specified name.
6. **fs.readdir()** - Asynchronously reads the contents of a directory.
7. **fs.stat()** - Asynchronously retrieves information about a file or directory.
8. **fs.watch()** - Watches a file or directory for changes.
9. **fs.createReadStream()** - Creates a readable stream for a file.
10. **fs.createWriteStream()** - Creates a writable stream for a file.
11. **fs.rmdir()** – Async Remove a directory
12. **fs.unlink()** – Async Remove a file

These are just a few of the many functions provided by the **fs** module. The most useful function will depend on the specific requirements of your application. However, **fs.readFile()** and **fs.writeFile()** are two of the most commonly used functions in Node.js, as they provide a simple way to read from and write to files asynchronously.

const fs =require('fs')

//  Reading Files

// In Node.js, fs.readFile() is an asynchronous function for reading files, which means that it does not block the execution of the rest of the code while waiting for the file to be read.

fs.readFile('./docs/blog1.txt',(err,data)=>{

    if(err){

        console.log("err",err)

    }

    console.log(data.toString())

})

console.log('Last Line')

// Writing Files

// il va modifier le fichier blog1.txt et mettre hello world Async

fs.writeFile('./docs/blog1.txt','hello world',()=>{

    console.log('file was written')

})

// il va cree le fichier blog2.txt car il existe pas puis il va mettre hello again Async

fs.writeFile('./docs/blog2.txt','hello again',()=>{

    console.log('file was written')

})

// Directories

//  existsSync ==>verifie si le fichier ou le dossier exist Syncrone

if(!fs.existsSync('./assets')){

    // cree un dossier Async

    fs.mkdir('./assets',(err)=>{

        if(err){

            console.log("err",err)

        }

        console.log('Folder created')

    })

} else{

    // Supprimer un dossier Async

    fs.rmdir('./assets',(err)=>{

        if(err){

            console.log("err",err)

        }

        console.log('Folder deleted')

    })

}

// deleting files

if(fs.existsSync('./docs/deleteme.txt')){

    fs.unlink('./docs/deleteme.txt',(err)=>{

        if(err){

            console.log(err)

        }

        console.log('file deleted')

    })

}

* 1. Streams & Buffers

In Node.js, streams are a way of handling continuous data flow in a more efficient and scalable way. Streams allow data to be read or written in chunks, rather than loading it all into memory at once. This is particularly useful for working with large amounts of data or for handling data that is being received or sent over a network.

There are four types of streams in Node.js:

Readable: A readable stream represents a source of data from which data can be read. Examples include reading data from a file or receiving data from a network socket.

Writable: A writable stream represents a destination to which data can be written. Examples include writing data to a file or sending data over a network socket.

Duplex: A duplex stream is both readable and writable. Examples include sending and receiving data over a network socket at the same time.

Transform: A transform stream is a type of duplex stream that can modify or transform data as it is read from or written to a source or destination.

const fs = require("fs");

const readStream = fs.createReadStream("./docs/blog3.txt", {

  encoding: "utf8",

});

// ReadStream

readStream.on("data", (chunk) => {

  console.log("================= New Chunk ==================================\n\n\n\n\n\n");

  console.log(chunk);

});

// Write stream

const writeStream=fs.createWriteStream('./docs/blog4.txt')

// readStream.on("data", (chunk) => {

//     console.log("================= New Chunk ==================================\n\n\n\n\n\n");

//     console.log(chunk);

//     writeStream.write('\n New Chunk \n')

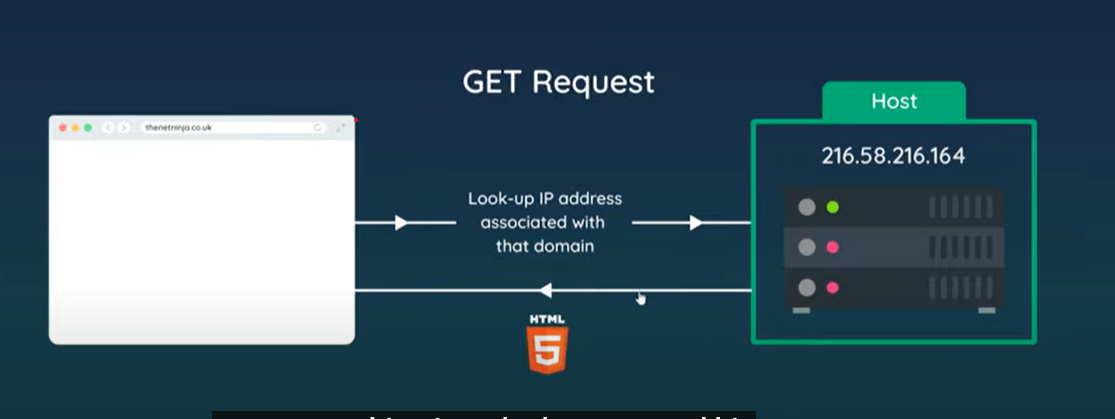
//     writeStream.write(chunk)

//   });

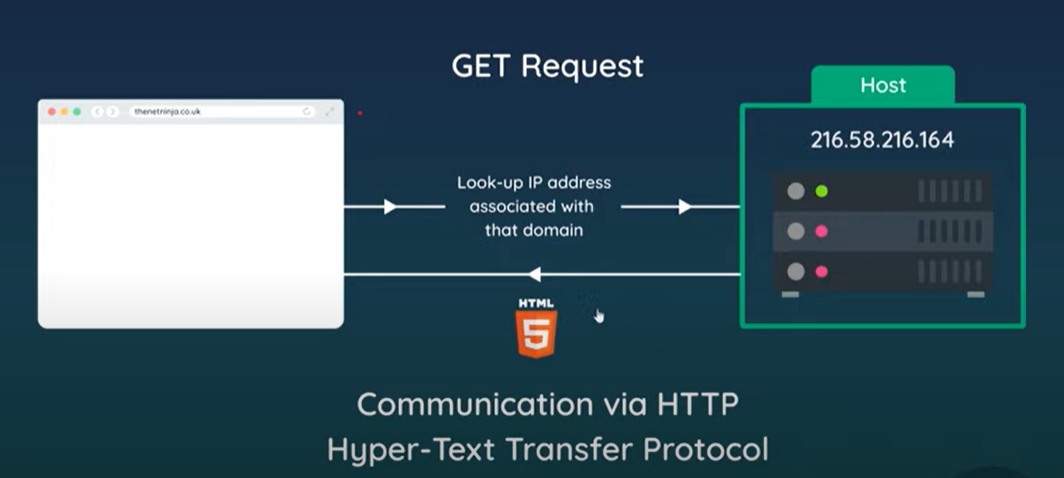
//   Pipe do the same thing

readStream.pipe(writeStream)

1. Clients and servers
2. Ip Adresses & Domains

An IP address is a unique numerical identifier assigned to every device that is connected to the internet. It is used to identify and communicate with other devices on a network, and to route data packets between devices 

* What is an http:



HTTP stands for Hypertext Transfer Protocol. It is an application protocol used for transferring data over the internet between a client (such as a web browser) and a server. HTTP is the foundation of the World Wide Web, and it is used for accessing and transferring data on websites and web applications.

HTTP operates on a client-server model, where the client sends a request to the server, and the server responds with a message containing the requested data or an error message if the request cannot be fulfilled. HTTP requests and responses are composed of headers and a message body.

HTTP requests are made up of several parts, including:

1. Request line: This includes the HTTP method (GET, POST, etc.), the URL of the resource being requested, and the HTTP version being used.
2. Request headers: These provide additional information about the request, such as the user agent making the request, the type of data being requested, and any authentication information.
3. Request body: This contains any data being sent to the server, such as form data or file uploads.

HTTP responses also have several parts, including:

1. Status line: This includes the HTTP version, a status code indicating whether the request was successful or not, and a brief message explaining the status.
2. Response headers: These provide additional information about the response, such as the content type of the data being returned and caching instructions.
3. Response body: This contains the data being returned by the server, such as an HTML page or JSON object.

HTTP is a stateless protocol, meaning that each request and response is independent of any previous requests or responses. To maintain state across requests, web applications often use cookies or other mechanisms to store information about the client's session.

* Different types of http requests

In Node.js, there are several different types of HTTP requests that can be made to a server:

1. GET: The GET request is used to retrieve data from a server. When a client sends a GET request to a server, the server responds with the requested data, which is usually in the form of an HTML page, an image, or a JSON object.
2. POST: The POST request is used to submit data to a server. When a client sends a POST request to a server, the server receives the data in the request body and processes it accordingly. This is commonly used for submitting form data or for creating new resources on the server.
3. PUT: The PUT request is used to update existing resources on a server. When a client sends a PUT request to a server, the server updates the resource with the data in the request body.
4. DELETE: The DELETE request is used to delete existing resources from a server. When a client sends a DELETE request to a server, the server deletes the specified resource.
5. PATCH: The PATCH request is used to update a portion of an existing resource on a server. When a client sends a PATCH request to a server, the server updates only the specified portion of the resource with the data in the request body.
6. HEAD: The HEAD request is similar to a GET request, but it only retrieves the headers of the response, not the actual data. This is commonly used to check if a resource exists or to retrieve metadata about a resource without actually downloading it.
7. OPTIONS: The OPTIONS request is used to retrieve information about the communication options available for a particular resource on a server. This is commonly used for cross-origin resource sharing (CORS) and to check which methods are allowed for a resource.

In Node.js, these requests can be handled using the built-in **http** or **https** modules, or by using third-party frameworks such as Express or Koa.

1. Creating a server

This code creates a constant variable called **http** and assigns the **http** module to it. You can then use this variable to create an HTTP server, make HTTP requests, and handle HTTP responses in your Node.js application.

For example, to create an HTTP server using the **http** module, you can use the following code:

const http = require("http");

const server = http.createServer((req, res) => {

  res.writeHead(200, { "Content-Type": "text/plain" });

  res.end("Hello, world!");

});

server.listen(3000, "localhost", () => {

  console.log("Server running on port 3000");

});

This code creates an HTTP server that listens on port 3000 and responds to all requests with a plain text message that says "Hello, world!". When a client makes a request to this server, the **createServer** function creates a new HTTP server instance, and the callback function handles the incoming request and sends a response back to the client.

1. Clients and servers

In NodeJs every time we make a change we have to restart our file because it’s old file and it not picked up the code change (we execute the cmd node server)

1. The Response object

En Node.js, vous pouvez envoyer différents types de contenus au navigateur en utilisant la méthode **response** de l'objet **http.ServerResponse**. Les types de contenus les plus courants sont :

* HTML : vous pouvez renvoyer une page HTML en utilisant la méthode **response.write()** pour écrire le contenu de la page et **response.end()** pour terminer la réponse.
* JSON : vous pouvez envoyer des données au format JSON en utilisant la méthode **response.json()** de certains frameworks comme Express.js.
* Fichiers : vous pouvez envoyer des fichiers tels que des images, des fichiers CSS ou JavaScript en utilisant la méthode **response.sendFile()** ou **response.write()** pour écrire le contenu du fichier et **response.end()** pour terminer la réponse.

Il est également possible d'envoyer d'autres types de contenus tels que des textes bruts, des fichiers PDF ou des fichiers audio ou vidéo en spécifiant correctement l'en-tête **Content-Type** de la réponse.

Une image contenant texte

Description générée automatiquement

First of all we need to set the header for the content type being send to the browser then we write whatever content we want to send to the browser the we ending the response which then sends to the browser

1. Returning Html page

First Thing we create our html file (In this case is index.html)

const http = require("http");

const fs=require("fs")

const server = http.createServer((req, res) => {

  console.log(req.url,req.method);

  res.setHeader('Content-Type','text/html')

  // send an html file

  fs.readFile("./views/index.html",(err,data)=>{

    if(err){

   console.log(err)

   res.end()

    } else{

      res.write(data)

      res.end()

    }

  })

});

server.listen(3000, "localhost", () => {

  console.log("listening for request on port 3000");

});

1. Basic Routing

In this example we will use 3 html file

const http = require("http");

const fs = require("fs");

const path = require("path");

const server = http.createServer((req, res) => {

  console.log(req.url, req.method);

  res.setHeader("Content-Type", "text/html");

  // Basic routing

  let path = "./views/";

  switch (req.url) {

    case "/":

      path += "index.html";

      break;

    case "/about":

      path += "about.html";

      break;

    default:

      path += "404.html";

      break;

  }

  fs.readFile(path, (err, data) => {

    if (err) {

      console.log(err);

      res.end();

    } else {

      res.write(data);

      res.end();

    }

  });

});

server.listen(3000, "localhost", () => {

  console.log("listening for request on port 3000");

});

1. Status code

En Node.js, il existe plusieurs codes de statut HTTP que vous pouvez utiliser pour indiquer le résultat d'une requête HTTP. Les codes de statut HTTP sont des nombres à trois chiffres qui indiquent si une requête a été traitée avec succès ou s'il y a eu une erreur.

Voici quelques-uns des codes de statut HTTP les plus couramment utilisés en Node.js :

* 200 OK : la requête a été traitée avec succès.
* 201 Created : la ressource a été créée avec succès.
* 204 No Content : la requête a été traitée avec succès, mais il n'y a pas de contenu à renvoyer.
* 301 Moved Permanently :est utilisé pour indiquer qu'une ressource a été déplacée de façon permanente vers une autre adresse URL. Cela signifie que toute demande future pour la même ressource doit être dirigée vers la nouvelle adresse URL.
* 400 Bad Request : la requête n'a pas pu être comprise ou traitée en raison d'une syntaxe invalide.
* 401 Unauthorized : l'utilisateur n'est pas autorisé à accéder à la ressource demandée.
* 403 Forbidden : l'utilisateur est authentifié mais n'a pas les droits d'accès nécessaires pour accéder à la ressource.
* 404 Not Found : la ressource demandée n'a pas été trouvée sur le serveur.
* 500 Internal Server Error : une erreur interne s'est produite sur le serveur lors du traitement de la requête.

Vous pouvez utiliser ces codes de statut HTTP dans la méthode **response.writeHead()** de l'objet **http.ServerResponse** pour définir le code de statut et les en-têtes de la réponse HTTP.

La propriété **res.statusCode** peut être utilisée pour obtenir le code de statut actuel de la réponse HTTP, ou pour définir un nouveau code de statut avant d'envoyer la réponse au client.

let path = "./views/";

  switch (req.url) {

    case "/":

      path += "index.html";

      res.statusCode=200

      break;

    case "/about":

      path += "about.html";

      res.statusCode=200

      break;

    default:

      path += "404.html";

      res.statusCode=404

      break;

  }

  fs.readFile(path, (err, data) => {

    if (err) {

      console.log(err);

      res.end();

    } else {

      res.write(data);

      res.end();

    }

  });

});

server.listen(3000, "localhost", () => {

  console.log("listening for request on port 3000");

});

1. Redirects

En Node.js, la méthode **res.setHeader()** peut être utilisée pour définir l'en-tête **Location** dans une réponse HTTP. L'en-tête **Location** est utilisé pour spécifier une nouvelle adresse URL vers laquelle le client doit être redirigé.

In this example we change the url handler from about-me and it send us to 404 page but that fine in our solution we need to redirect to about url handler if we change the url to about-me

 let path = "./views/";

  switch (req.url) {

    case "/":

      path += "index.html";

      res.statusCode=200

      break;

    case "/about":

      path += "about.html";

      res.statusCode=200

      break;

    case '/about-me':

      res.statusCode=301

      res.setHeader('location','/about')

      res.end()

    default:

      path += "404.html";

      res.statusCode=404

      break;

  }

  fs.readFile(path, (err, data) => {

    if (err) {

      console.log(err);

      res.end();

    } else {

      res.write(data);

      res.end();

    }

  });

});

server.listen(3000, "localhost", () => {

  console.log("listening for request on port 3000");

});