NodeJs Tutrorial The NetNinja

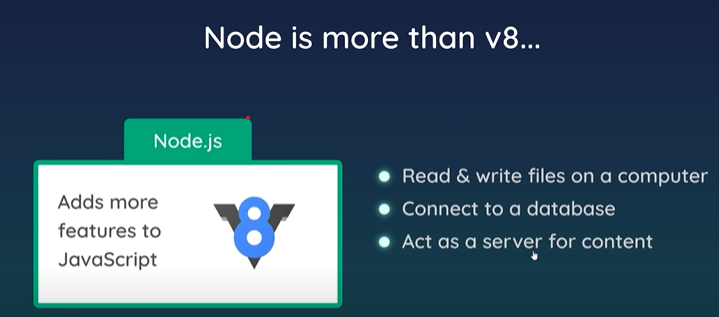
1. Introduction
   1. Introduction

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')

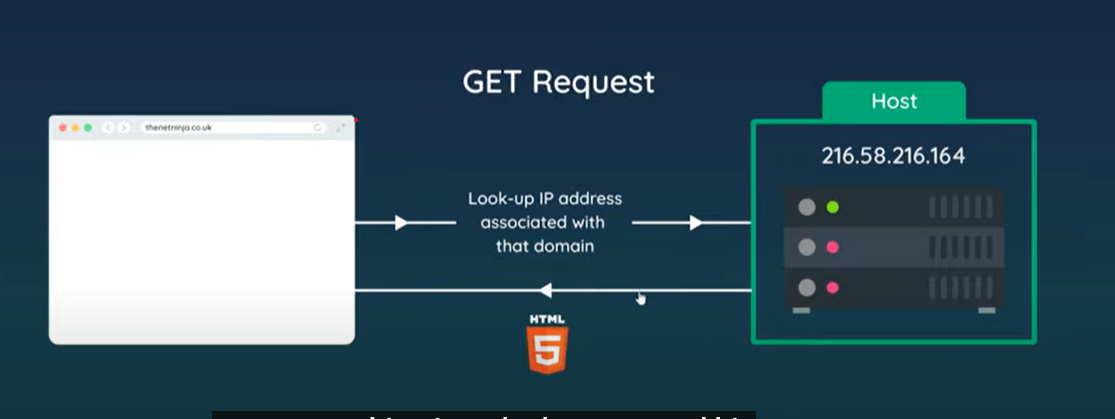
//     writeStream.write(chunk)

//   });

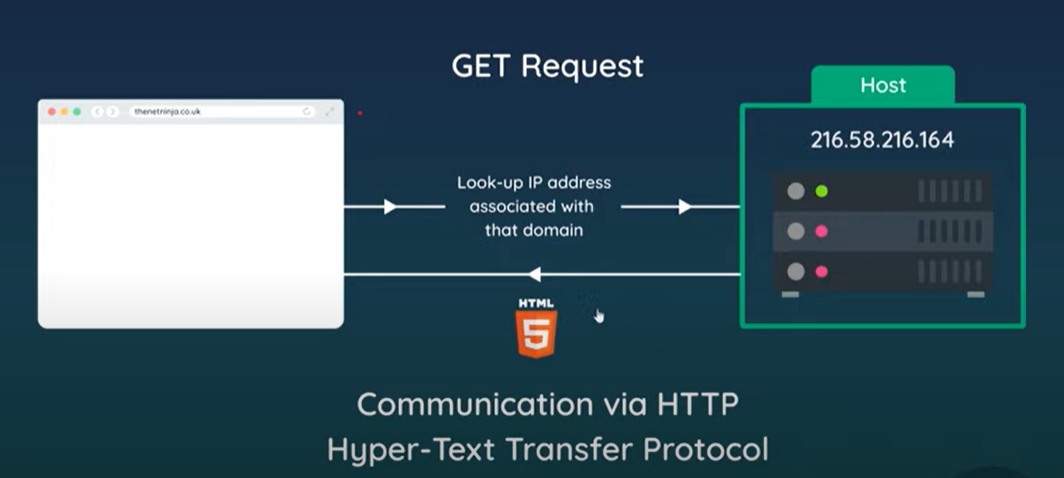
//   Pipe do the same thing

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");

});

1. Npm
2. Introduction to nodeJs and global package

NPM (Node Package Manager) est un gestionnaire de packages open source pour le langage de programmation JavaScript. Il est le gestionnaire de paquets par défaut pour l'environnement d'exécution JavaScript Node.js. NPM permet aux développeurs d'installer, de partager et de gérer les bibliothèques et les modules de code JavaScript qui sont nécessaires pour construire des applications Node.js et d'autres applications JavaScript côté serveur et côté client.

Les packages NPM sont publiés dans un registre public en ligne, qui permet aux développeurs de partager et de télécharger des packages gratuitement ou moyennant des frais. Les développeurs peuvent également créer et publier leurs propres packages NPM pour partager leur code avec d'autres développeurs.

NPM est utilisé par de nombreux développeurs JavaScript pour accélérer et faciliter le processus de développement en permettant aux développeurs de réutiliser du code existant et de résoudre les dépendances entre les différents packages utilisés dans une application.

**In this example we will install nodeMon**

1. Package.json File

Nodemon est un outil très utile en développement Node.js car il permet de redémarrer automatiquement l'application Node.js à chaque fois qu'un changement est détecté dans le code source. Cela permet d'éviter aux développeurs de devoir redémarrer manuellement l'application à chaque modification, ce qui peut être fastidieux et chronophage.

For installing nodemon globaly in our computer we will run **npm i -g nodemon**

**npx nodemon server 🡺** This cmd will start and restart the server

On utilise la commande "npm init" pour initialiser un nouveau projet Node.js et créer un fichier "package.json" qui décrit les détails du projet et les dépendances nécessaires pour le faire fonctionner.

Le fichier "package.json" est un fichier clé pour tout projet Node.js, car il contient des informations sur le projet, telles que le nom du projet, la version, l'auteur, la description, les scripts, les dépendances, les devDependencies, etc. Ce fichier est utilisé par NPM pour installer et gérer les dépendances de l'application, ce qui facilite la configuration et la distribution de l'application.

La commande "npm init" permet aux développeurs de spécifier les détails du projet tels que le nom, la version, la description, l'auteur, les scripts, les dépendances, etc. et génère automatiquement un fichier "package.json" avec ces informations. Cette commande est utile car elle permet de gagner du temps et d'éviter les erreurs lors de la création manuelle d'un fichier "package.json".

En somme, la commande "npm init" est un outil important pour créer rapidement un nouveau projet Node.js et configurer son environnement de développement. Elle permet également de faciliter la gestion des dépendances du projet et la collaboration entre développeurs.

1. Installing packages locally

**npm i packageName**

1. Dependencies

En Node.js, les dépendances (ou "dependencies" en anglais) sont des modules ou des bibliothèques externes qui sont nécessaires au bon fonctionnement d'une application Node.js. Ces dépendances peuvent être des modules de base de Node.js ou des modules tiers qui sont installés via le gestionnaire de packages de Node.js (NPM).

Pour utiliser un module tiers dans une application Node.js, les développeurs doivent ajouter le module en tant que dépendance dans le fichier "package.json" de l'application. Cela peut être fait manuellement en éditant le fichier, ou en utilisant la commande "npm install" suivie du nom du module. NPM ajoutera automatiquement le module en tant que dépendance dans le fichier "package.json" et l'installera dans le dossier "node\_modules" de l'application.

Une fois qu'une dépendance est installée, elle peut être utilisée dans l'application Node.js en l'exigeant dans le code à l'aide de la fonction "require". Par exemple, pour utiliser le populaire framework web "Express" dans une application Node.js, les développeurs devraient l'installer en utilisant "npm install express" et ensuite l'exiger dans leur code en utilisant "const express = require('express')".

Les dépendances sont importantes en Node.js car elles permettent aux développeurs de réutiliser du code et d'éviter de réinventer la roue pour des fonctionnalités communes. En utilisant des bibliothèques et des modules existants, les développeurs peuvent gagner du temps et se concentrer sur les aspects uniques de leur application.

1. Express Apps
2. Definition

Express est un framework web pour Node.js. Il est conçu pour simplifier le processus de création d'applications web en fournissant un ensemble de fonctionnalités de base pour la création de serveurs web, telles que la gestion des routes, la gestion des requêtes et des réponses HTTP, la gestion des cookies, la gestion des sessions, etc.

Express est très populaire dans l'écosystème Node.js et est utilisé par de nombreux développeurs pour créer des applications web rapides et évolutives. Il est également hautement personnalisable et extensible, permettant aux développeurs d'ajouter facilement des fonctionnalités supplémentaires via des modules complémentaires tiers.

En résumé, Express est un outil puissant et flexible pour la création d'applications web en utilisant Node.js, permettant aux développeurs de se concentrer sur la création de fonctionnalités plutôt que sur la configuration et la gestion du serveur web.

1. Creating an express app

const express = require("express");

// express app

const app = express();

// Listen for requests

app.listen(3000)

app.get("/",(req,res)=>{

    res.end('<p>Home page</p>')

})

3. Routing & Html

// express app

const app = express();

// Listen for requests

app.listen(3000);

path.join(\_\_dirname, "views", "index.js");

app.get("/", (req, res) => {

  // res.end('<p>Home page</p>')

  res.sendFile(path.join(\_\_dirname, "views", "index.html"));

});

app.get("/about", (req, res) => {

  // res.end('<p>About page</p>')

  res.sendFile(path.join(\_\_dirname, "views", "about.html"));

});

1. Redirects & 404’s
   1. Redirects

// redirects

app.get("/about-us", (req, res) => {

  res.redirect("/about");

});

* 1. 404’s

En utilisant **app.use** à la fin d'un fichier, on peut définir une fonction de gestion d'erreurs pour les requêtes qui ne correspondent à aucune route définie dans l'application. C'est communément appelé une réponse 404 ("Not Found").

Lorsque vous définissez une route dans votre application Express, vous pouvez spécifier une fonction de gestion de cette route qui sera appelée lorsque la requête correspond à la route spécifiée. Cependant, si une requête est faite pour une route qui n'a pas été définie, Express ne peut pas trouver de route correspondante et doit renvoyer une réponse 404.

Pour gérer cette situation, vous pouvez définir une fonction de gestion d'erreur à la fin de votre fichier qui utilisera **app.use** avec une fonction de middleware pour gérer toutes les requêtes qui ne correspondent à aucune des routes définies précédemment dans l'application. Cette fonction de middleware sera appelée uniquement si aucune des routes précédentes n'a géré la requête.

Voici un exemple de déclaration de middleware pour gérer les requêtes 404 :

// 404 page

app.use((req,res)=>{

  res.status(404).sendFile(path.join(\_\_dirname, "views", "404.html"))

})

Cette fonction de middleware utilise **res.status** pour renvoyer une réponse HTTP avec un code de statut 404 ("Not Found") et **res.send** pour envoyer une réponse de texte indiquant que la page demandée n'existe pas. En utilisant **app.use**, cette fonction de middleware sera exécutée pour toute requête qui ne correspond à aucune des routes définies précédemment dans l'application.

En résumé, en utilisant **app.use** à la fin d'un fichier pour définir une gestion d'erreur 404, vous pouvez assurer que votre application répondra de manière appropriée à toutes les requêtes, même celles qui ne correspondent à aucune des routes définies dans l'application.

1. View Enginenes
2. Definition

Un view engine (ou template engine) en Node.js est un outil permettant de générer dynamiquement des pages HTML en utilisant des modèles (ou templates) pré-définis, ainsi que des données provenant de sources externes telles que des bases de données, des fichiers JSON, ou des données provenant d'une API.

Les view engines sont souvent utilisés dans les applications web pour simplifier le processus de génération de pages HTML dynamiques. Au lieu d'écrire manuellement le HTML pour chaque page, le view engine permet de définir un modèle qui peut être réutilisé pour générer automatiquement des pages HTML à partir de données variables.

Il existe plusieurs view engines populaires pour Node.js, tels que Pug (anciennement Jade), Handlebars, EJS (Embedded JavaScript), et Mustache. Chacun de ces moteurs a sa propre syntaxe et ses propres fonctionnalités.

Par exemple, avec Pug, vous pouvez définir une page de modèle en utilisant une syntaxe simplifiée et élégante, qui sera ensuite compilée en HTML à la volée par le moteur. Voici un exemple :

Une image contenant texte, écran, capture d’écran

Description générée automatiquement

Lorsque cette page de modèle est utilisée avec des données dynamiques, le moteur Pug remplacera les valeurs des variables pour générer une page HTML complète, qui pourra être renvoyée au client.

En résumé, un view engine en Node.js est un outil qui permet de générer dynamiquement des pages HTML à partir de modèles et de données. Cela permet de simplifier le processus de génération de pages HTML dynamiques, et de faciliter la création d'applications web robustes et évolutives.

1. Ejs
2. EJS (Embedded JavaScript) est un moteur de template pour Node.js qui permet de générer du contenu HTML dynamique à partir de données stockées côté serveur. EJS permet d'insérer des variables, des boucles, des conditions et d'autres instructions JavaScript dans un fichier de modèle HTML, ce qui rend la création de pages web dynamiques plus facile et plus efficace.
3. EJS est souvent utilisé en combinaison avec Node.js car il est facile à intégrer dans des applications Node.js existantes. Il est également assez flexible pour être utilisé avec de nombreux frameworks web différents, tels que Express ou Koa.
4. En outre, EJS permet également de générer des pages HTML en fonction des événements de l'application Node.js, ce qui en fait un choix populaire pour les applications web en temps réel.
5. En résumé, EJS est utilisé en Node.js pour générer du contenu HTML dynamique à partir de données stockées côté serveur et permet une intégration facile avec les frameworks web courants.

C’est comme react t’as pas besoin de l’apprendre voici le lien en sait jamais : <https://www.youtube.com/watch?v=yXEesONd_54&list=PL4cUxeGkcC9jsz4LDYc6kv3ymONOKxwBU&index=7&ab_channel=TheNetNinja>

1. Middlware
2. Definition

En Node.js, un middleware est une fonction qui peut être utilisée pour intercepter et manipuler les requêtes HTTP entrantes avant qu'elles ne soient traitées par la fonction de routage principale.

Les middlewares sont souvent utilisés pour effectuer des tâches telles que la vérification de l'authentification de l'utilisateur, la gestion des erreurs, la compression des données, la mise en cache et la gestion des sessions.

En d'autres termes, un middleware est un bloc de code qui agit comme une couche intermédiaire entre la requête d'un client et le serveur. Il peut modifier la requête, ajouter des informations à la réponse, appeler la fonction de traitement suivante ou envoyer une réponse à la requête.

Les middlewares sont souvent empilés les uns sur les autres pour former une chaîne de traitement de la requête. Chaque middleware peut effectuer une tâche spécifique et passer la requête à la prochaine fonction de middleware dans la chaîne. Cela permet de créer des applications web modulaires et facilement extensibles en ajoutant ou en supprimant des fonctions de middleware en fonction des besoins.

**Code which runs (on the server) between getting request and sending response**

**Middleware run from top to buttom**

Une image contenant diagramme

Description générée automatiquementUne image contenant texte, signe, capture d’écran

Description générée automatiquementUne image contenant texte

Description générée automatiquement

1. App.use()
   1. Definition

En Node.js, **app.use** est utilisé pour ajouter des middlewares à une application. Les middlewares sont des fonctions qui peuvent être utilisées pour intercepter et manipuler les requêtes HTTP entrantes avant qu'elles ne soient traitées par la fonction de routage principale.

L'utilisation de middlewares permet de réaliser de nombreuses tâches utiles dans une application Node.js, telles que :

* La vérification de l'authentification de l'utilisateur
* La gestion des erreurs
* La compression des données
* La mise en cache
* La gestion des sessions
* La gestion des cookies
* L'analyse des données de la requête (par exemple, pour récupérer les données d'un formulaire)
* La validation des données de la requête
* La modification des en-têtes HTTP
* La gestion des CORS (Cross-Origin Resource Sharing)
* Etc.

En utilisant **app.use** pour ajouter des middlewares à une application, vous pouvez créer une chaîne de traitement de requêtes qui exécutera les middlewares dans l'ordre où ils ont été ajoutés. À chaque middleware, vous pouvez soit passer la requête à la fonction de middleware suivante en appelant **next()**, soit y répondre immédiatement avec une réponse HTTP.

Par exemple, voici comment on pourrait ajouter un middleware pour vérifier si l'utilisateur est authentifié avant de permettre l'accès à une page spécifique :

app.use('/page-secrete', function(req, res, next) {

  if (req.isAuthenticated()) {

    next();

  } else {

    res.redirect('/connexion');

  }

});

Dans cet exemple, le middleware vérifie si l'utilisateur est authentifié en appelant la méthode **isAuthenticated()** (qui est définie ailleurs dans l'application). Si l'utilisateur est authentifié, il appelle la fonction **next()** pour passer la requête à la prochaine fonction de middleware. Sinon, il redirige l'utilisateur vers la page de connexion.

* 1. Faire Attention avec app.use

Lorsque vous utilisez **app.use** et **app.get** dans votre application Express, l'ordre dans lequel vous les définissez dans votre code est important.

En effet, les middleware définis avec **app.use** seront exécutés pour chaque requête qui arrive sur le serveur, alors que les fonctions de routage définies avec **app.get** ne seront exécutées que pour les requêtes HTTP GET.

Si vous définissez une route avec **app.get** avant un middleware avec **app.use**, la route sera exécutée avant que le middleware ne soit appelé(**c’est seulement le cas si on fait un appel de app.get(‘/user’) sinon le middleware va s’exécute**), ce qui signifie que le middleware ne sera pas appliqué à cette route.

Par exemple, si vous avez le code suivant :

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

  res.send('Hello World');

});

app.use(function(req, res, next) {

  console.log('Middleware applied');

  next();

});

Dans ce cas, la route **/user** sera exécutée avant le middleware, car elle est définie avant le middleware. Si vous voulez que le middleware soit exécuté avant la route, vous devez les définir dans l'ordre inverse :

app.use(function(req, res, next) {

  console.log('Middleware applied');

  next();

});

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

  res.send('Hello World');

});

Dans ce cas, le middleware sera exécuté avant la route **/user**.

Il est donc important de faire attention à l'ordre dans lequel vous définissez vos routes et middleware pour vous assurer que votre application fonctionne correctement.

1. Des middlewares existe déjà
   1. Morgan

Il est possible que vous faisiez référence à Morgan, qui est un middleware de journalisation pour Node.js. Morgan est utilisé pour enregistrer les requêtes HTTP entrantes et sortantes dans une application Node.js.

La journalisation est une pratique importante dans le développement d'applications car elle permet de suivre les erreurs et les comportements inattendus de l'application. Morgan fournit un middleware facile à utiliser qui permet aux développeurs d'enregistrer des informations telles que l'adresse IP de l'utilisateur, la méthode HTTP utilisée, l'URL demandée et le code de statut de la réponse envoyée.

L'utilisation de Morgan peut aider les développeurs à diagnostiquer les problèmes de l'application en fournissant des informations sur les erreurs et les comportements inattendus. De plus, cela peut également aider à améliorer les performances de l'application en fournissant des données sur les temps de réponse et les temps de traitement.

En fin de compte, Morgan est un outil utile pour les développeurs Node.js qui cherchent à améliorer le débogage et les performances de leurs applications en enregistrant des informations sur les requêtes HTTP.

app.use(morgan("dev"));

* 1. Static file

If we add some static file (images or css files ) to our project file then we wouldn’t be able to automatically access that file from the browser

Les fichiers statiques sont des fichiers tels que des images, des feuilles de style CSS et des fichiers JavaScript qui ne changent pas pendant l'exécution de l'application. Pour servir ces fichiers statiques dans une application Node.js, vous pouvez utiliser un middleware appelé "express.static".

Voici un exemple de code qui montre comment utiliser le middleware "express.static" pour servir des fichiers statiques dans une application Node.js :

const express = require('express');

const app = express();

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

app.listen(3000, () => {

  console.log('Le serveur est en écoute sur le port 3000');

});

Dans cet exemple, nous avons créé une application Express et avons utilisé la méthode "use" pour ajouter le middleware "express.static" à notre application. Nous avons spécifié le répertoire "public" en tant que répertoire racine pour les fichiers statiques.

Ainsi, lorsque l'application reçoit une requête pour un fichier statique, telle qu'une image ou une feuille de style, Express recherche ce fichier dans le répertoire "public" et le renvoie au client.

Le middleware "express.static" est un moyen simple et efficace de servir des fichiers statiques dans une application Node.js.

1. MongoDb
2. Difference entre un bdd sql et nosql

Les bases de données SQL (Structured Query Language) et NoSQL (Not Only SQL) sont deux types de systèmes de gestion de bases de données (SGBD) qui diffèrent par leur modèle de données et leur approche de stockage et de récupération des données.

Les bases de données SQL sont des bases de données relationnelles qui stockent des données dans des tables avec des relations définies entre elles. Les données sont stockées dans des colonnes et des lignes et peuvent être jointes ou liées entre elles pour former une vue complète des données. Les bases de données SQL sont largement utilisées dans les applications d'entreprise et sont adaptées aux applications qui nécessitent une intégrité des données élevée, une cohérence et une scalabilité verticale (ajout de ressources à une seule machine).

D'un autre côté, les bases de données NoSQL sont conçues pour stocker et récupérer des données non structurées ou semi-structurées. Elles ne suivent pas le modèle de données relationnel et permettent une évolutivité horizontale (ajout de nœuds dans un cluster) plutôt que verticale. Les bases de données NoSQL peuvent être utilisées pour des applications web à grande échelle qui nécessitent des temps de réponse rapides et une évolutivité horizontale. Elles offrent également une grande flexibilité pour stocker différents types de données, ce qui les rend idéales pour les applications nécessitant des types de données variables.

En résumé, les principales différences entre les bases de données SQL et NoSQL sont les suivantes :

* Modèle de données : SQL utilise le modèle de données relationnel, tandis que NoSQL utilise des modèles de données différents tels que clé-valeur, document ou graphe.
* Scalabilité : SQL est mieux adapté pour une évolutivité verticale, tandis que NoSQL est mieux adapté pour une évolutivité horizontale.
* Intégrité des données : SQL offre une intégrité des données élevée grâce aux contraintes de clé étrangère, tandis que NoSQL ne fournit pas cette fonctionnalité par défaut.
* Flexibilité : NoSQL est plus flexible en termes de stockage de différents types de données, tandis que SQL est plus rigide dans sa structure de données.

Une image contenant texte, capture d’écran, Police, Marque

Description générée automatiquement

1. Les collections dans une bdd noSql

Les bases de données NoSQL (Not Only SQL) sont conçues pour stocker et manipuler des données qui ne sont pas bien structurées ou qui ne s'adaptent pas facilement aux schémas de base de données relationnelles traditionnelles. Les bases de données NoSQL peuvent prendre en charge différents types de modèles de données, tels que le modèle de document, le modèle de colonnes, le modèle clé-valeur, etc.

Dans ce contexte, les collections sont souvent utilisées pour stocker des données de manière non structurée. Les collections sont similaires aux tables dans les bases de données relationnelles, mais elles permettent souvent de stocker des données de manière plus flexible. Par exemple, dans une base de données NoSQL utilisant le modèle de document, une collection peut stocker des documents JSON (JavaScript Object Notation) qui peuvent avoir des champs différents pour chaque document.

Les collections peuvent également être utilisées pour organiser les données en fonction de leur type ou de leur objectif. Par exemple, vous pouvez stocker des données d'utilisateurs dans une collection, des données de produits dans une autre collection et des données de transactions dans une troisième collection. Cette approche permet d'optimiser la performance en permettant des recherches plus rapides et plus efficaces.

En fin de compte, les collections sont un élément important des bases de données NoSQL car elles permettent de stocker des données de manière flexible et de les organiser de manière à optimiser la performance et la récupération.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement

1. MongoDb Setup & Atlas

MongoDB Atlas est un service cloud de base de données géré proposé par MongoDB, Inc., le créateur de la base de données NoSQL MongoDB. Atlas permet aux développeurs de créer, gérer et exécuter des clusters de bases de données MongoDB dans le cloud, sans avoir à se soucier de l'infrastructure sous-jacente.

En utilisant MongoDB Atlas, les développeurs peuvent profiter de plusieurs avantages, notamment :

* Évolutivité : Atlas permet de facilement scaler les clusters de bases de données MongoDB en ajoutant ou en supprimant des nœuds, afin de répondre aux besoins des applications en termes de volume de données et de trafic.
* Disponibilité : Atlas assure une haute disponibilité en répartissant automatiquement les données sur plusieurs nœuds et en utilisant des mécanismes de réplication pour garantir la disponibilité des données même en cas de panne de matériel ou de réseau.
* Sécurité : Atlas prend en charge plusieurs fonctionnalités de sécurité telles que le chiffrement des données au repos et en transit, les listes de contrôle d'accès, la surveillance de la sécurité et l'authentification multi-facteurs.
* Gestion simplifiée : Atlas propose une interface web conviviale pour la gestion et la surveillance des clusters de bases de données MongoDB, ainsi que des outils de surveillance et de diagnostic pour résoudre rapidement les problèmes éventuels.

Atlas est compatible avec les principaux fournisseurs de cloud tels que AWS, Google Cloud Platform et Microsoft Azure, offrant ainsi aux développeurs la possibilité de choisir le fournisseur de leur choix pour héberger leurs clusters de bases de données MongoDB.

<https://www.youtube.com/watch?v=bxsemcrY4gQ&list=PL4cUxeGkcC9jsz4LDYc6kv3ymONOKxwBU&index=9&ab_channel=TheNetNinja>

1. Mongoose, Models & Schémas

Mongoose est une bibliothèque Node.js qui fournit un mapping objet-document (ODM) pour la base de données NoSQL MongoDB. L'objectif principal de Mongoose est de simplifier l'utilisation de MongoDB en permettant aux développeurs de définir des schémas de données, de valider les données entrantes, de définir des relations entre les données, et de manipuler les données de manière plus facile et plus naturelle en utilisant des modèles d'objets JavaScript.

En utilisant Mongoose, les développeurs peuvent :

* Définir des schémas pour les collections MongoDB, qui permettent de définir les types de données pour chaque champ, de définir des clés primaires et étrangères, et de définir des contraintes d'intégrité.
* Effectuer des opérations CRUD (create, read, update, delete) sur les collections MongoDB à l'aide de méthodes simples et naturelles en utilisant des modèles d'objets JavaScript.
* Gérer les relations entre les collections en utilisant des références ou en intégrant des documents emboîtés dans d'autres documents.
* Définir des méthodes personnalisées pour les modèles d'objets JavaScript afin de simplifier l'écriture de code répétitif.
* Valider les données entrantes pour s'assurer que les données sont conformes aux contraintes de schéma définies.

Mongoose est largement u tilisé dans le développement d'applications web et mobiles pour Node.js, car il permet aux développeurs de travailler avec MongoDB de manière plus facile et plus naturelle, tout en tirant parti de la puissance et de la flexibilité de la base de données NoSQL.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

En MongoDB, un schéma est une structure qui définit la forme des documents stockés dans une collection. Un schéma peut définir les champs et leur type de données, ainsi que les contraintes d'intégrité et les index à appliquer sur les données.

Dans Mongoose, un schéma est défini en utilisant la classe **Schema** et peut être utilisé pour créer des modèles de données qui permettent d'effectuer des opérations CRUD sur les documents de la collection. Les modèles sont définis en utilisant la méthode **model** de Mongoose, qui prend en paramètre un nom pour le modèle et un schéma pour définir la forme des documents.

Voici un exemple simple de définition de schéma et de modèle en utilisant Mongoose :

const mongoose = require('mongoose');

// Définition du schéma pour la collection "users"

const userSchema = new mongoose.Schema({

  name: { type: String, required: true },

  age: { type: Number, required: true },

  email: { type: String, required: true, unique: true },

  createdAt: { type: Date, default: Date.now }

});

// Création du modèle pour la collection "users" en utilisant le schéma défini ci-dessus

const User = mongoose.model('User', userSchema);

Dans cet exemple, le schéma **userSchema** définit les champs **name**, **age**, **email** et **createdAt** pour la collection "users". Le modèle **User** est créé en utilisant la méthode **model** de Mongoose, avec le nom "User" et le schéma **userSchema**.

Une fois le modèle créé, les opérations CRUD peuvent être effectuées sur la collection "users" en utilisant les méthodes de Mongoose telles que **create**, **find**, **findOne**, **updateOne**, **deleteOne**, etc.

En utilisant des schémas et des modèles, les développeurs peuvent définir la forme des données stockées dans une collection MongoDB et effectuer des opérations CRUD de manière plus facile et plus naturelle, en utilisant des objets JavaScript plutôt que des requêtes MongoDB brutes.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, Police, diagramme

Description générée automatiquement

* 1. Connecte ma bdd to node

const mongoose = require("mongoose");

// connect to mongodb

const uri =

  "mongodb+srv://test1234:test1234@cluster0.6gk9dki.mongodb.net/?retryWrites=true&w=majority";

mongoose

  .connect(uri, { useNewUrlParser: true, useUnifiedTopology: true })

  .then((resulr) => {

    console.log("Connect to db");

    // listen for requests

    // We're only going to listen for requests after this connection is complete

    app.listen(3000);

  })

  .catch((err) => {

    console.log("Erreur", err);

  });

* 1. Create our blog schema and our blog model

const mongoose = require("mongoose");

const Schema = mongoose.Schema;

const blogSchema = new Schema(

  {

    title: {

      type: String,

      required: true,

    },

    snippet: {

      type: String,

      required: true,

    },

    body: {

      type: String,

      required: true,

    },

  },

  { timestamps: true }

);

const Blog = mongoose.model("Blog", blogSchema);

module.exports = Blog;

We create a blog schema and we create a model is the thing that surrounds that and then provides us with an interface by which to communicate a db collection for that document type

const Blog = mongoose.model("Blog", blogSchema);

it takes in as a first argument the name of this model (it look for the name of the collection inside our db ) the second argument is going to be the scema we want to base this model on so what type of objects and we’re going to store inside this collection

* 1. Getting & Saving Data

const express = require("express");

const mongoose = require("mongoose");

const Blog = require("./models/blog");

// express app

const app = express();

// connect to mongodb

const uri =

  "mongodb+srv://test1234:test1234@cluster0.6gk9dki.mongodb.net/node-tuts?retryWrites=true&w=majority";

mongoose

  .connect(uri, { useNewUrlParser: true, useUnifiedTopology: true })

  .then((resulr) => {

    console.log("Connect to db");

    // listen for requests

    // We're only going to listen for requests after this connection is complete

    app.listen(3000);

  })

  .catch((err) => {

    console.log("Erreur", err);

  });

// mongoose and mongo sandbox routes

app.get("/add-blog", (req, res) => {

  const blog = new Blog({

    title: "new blog",

    snippet: "about my new blog",

    body: "more about my new blog",

  });

  blog

    .save()

    .then((result) => {

      res.send(result);

    })

    .catch((err) => {

      console.log("Err", err);

    });

});

app.get("/all-blogs", (req, res) => {

  Blog.find()

    .then((result) => {

      res.send(result);

    })

    .catch((err) => {

      console.log(err);

    });

});

app.get("/single-blog", (req, res) => {

  Blog.findById("64566c65948725187f178daa")

    .then((result) => {

      res.send(result);

    })

    .catch((err) => {

      console.log(err);

    });

});

* 1. Outputting Documents in Views

app.get("/", (req, res) => {

  res.redirect("/blogs");

});

app.get("/blogs", (req, res) => {

  Blog.find()

    .sort({ createdAt: -1 })

    .then((result) => {

      res.render("index", { title: "All Blogs", blogs: result });

    })

    .catch((err) => {

      console.log(err);

    });

});

1. Get, Post & Delete Requests

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* 1. Post Requests

Une "POST request" (ou requête POST en français) est une méthode de requête HTTP utilisée pour envoyer des données à un serveur web. Contrairement à une requête GET, qui envoie des données dans l'URL, une requête POST envoie des données dans le corps de la requête.

Les données envoyées dans une requête POST peuvent être de différents types, tels que des formulaires, des fichiers, des images, des vidéos ou tout autre type de contenu. Le serveur web qui reçoit la requête POST peut alors traiter ces données et y répondre en conséquence.

Les requêtes POST sont souvent utilisées pour envoyer des informations sensibles, telles que des identifiants de connexion ou des informations de carte de crédit, car les données ne sont pas visibles dans l'URL et sont donc plus sécurisées. Les applications web, les API et les services web utilisent fréquemment des requêtes POST pour envoyer des données entre clients et serveurs.

app.use(express.urlencoded({ extended: true }));

La méthode **app.use(express.urlencoded({ extended: true }))** est utilisée dans les applications Node.js pour ajouter un middleware à l'application.

Plus spécifiquement, cette ligne de code configure le middleware **urlencoded** fourni par le module **body-parser** d'Express, qui analyse le corps des requêtes HTTP entrantes avec un type de contenu **application/x-www-form-urlencoded**.

En d'autres termes, lorsque le navigateur envoie une requête POST avec des données encodées dans le corps de la requête, le middleware **urlencoded** analyse ces données et les transforme en un objet JavaScript utilisable dans l'application.

Le paramètre **{ extended: true }** indique au middleware **urlencoded** de prendre en charge les valeurs de types complexes, tels que les tableaux et les objets imbriqués, plutôt que de se limiter à des chaînes de caractères simples.

En somme, cette ligne de code est souvent incluse dans l'initialisation d'une application Node.js pour configurer le middleware **urlencoded** et faciliter la manipulation des données de formulaire dans l'application.

app.post("/blogs", (req, res) => {

  const blog = new Blog(req.body);

  blog

    .save()

    .then((result) => {

      res.redirect("/blogs");

    })

    .catch((err) => {

      console.log(err);

    });

});

* 1. Route Parameters

app.get("/blogs/:id", (req, res) => {

  const id = req.params.id;

  Blog.findById(id)

    .then((result) => {

      res.render("details", { blog: result, title: "Blog Details" });

    })

    .catch((err) => {

      console.log(err);

    });

});

* 1. Delete Requests

Une "delete request" (ou requête DELETE en français) est une méthode de requête HTTP utilisée pour demander la suppression d'une ressource spécifique sur un serveur web. Contrairement à une requête GET ou POST, qui permettent de récupérer ou de modifier des données, une requête DELETE demande explicitement la suppression d'une ressource.

Pour envoyer une requête DELETE, un client web doit envoyer une requête HTTP DELETE à l'URL de la ressource qu'il souhaite supprimer. Le serveur web qui reçoit la requête DELETE doit alors supprimer la ressource demandée et renvoyer une réponse appropriée.

Il est important de noter que toutes les ressources ne peuvent pas être supprimées avec une requête DELETE. Les serveurs web peuvent être configurés pour refuser les requêtes DELETE sur certaines ressources, pour des raisons de sécurité ou pour protéger les données. Il est également possible que les utilisateurs ne disposent pas des autorisations nécessaires pour supprimer certaines ressources.

Les requêtes DELETE sont fréquemment utilisées dans les applications web et les API pour permettre aux utilisateurs de supprimer des ressources qu'ils ont créées ou auxquelles ils ont accès. Cependant, elles doivent être utilisées avec précaution pour éviter la suppression accidentelle ou malveillante de données importantes.

app.delete("/blogs/:id", (req, res) => {

  const id = req.params.id;

  Blog.findByIdAndDelete(id)

    .then((result) => {

      res.json({ redirect: "/blogs" });

    })

    .catch((err) => {

      console.log(err);

    });

});

Code in FrontEnd Side :

<script>

    const trashcan = document.querySelector('a.delete');

    trashcan.addEventListener('click', (e) => {

      const endpoint = `/blogs/${trashcan.dataset.doc}`;

      fetch(endpoint, {

        method: 'DELETE',

      })

      .then(response => console.log(response.json()))

      .then(data => window.location.href = data.redirect)

      .catch(err => console.log(err));

    });

  </script>