**Node Express**

1. **Clients and Servers**
2. **Ip Adress & Domains**

Une adresse IP est un identifiant numérique unique attribué à chaque appareil connecté à Internet. Elle est utilisée pour identifier et communiquer avec d'autres appareils sur un réseau, ainsi que pour acheminer des paquets de données entre les appareils.

Une image contenant texte, logiciel, Logiciel multimédia, diagramme

Description générée automatiquement

* **C’est quoi un http**

Une image contenant texte, logiciel, Logiciel multimédia, Police

Description générée automatiquement

HTTP signifie Hypertext Transfer Protocol. C'est un protocole d'application utilisé pour transférer des données sur Internet entre un client (tel qu'un navigateur Web) et un serveur. HTTP est le fondement du World Wide Web, et il est utilisé pour accéder et transférer des données sur des sites Web et des applications Web.

HTTP fonctionne selon un modèle client-serveur, où le client envoie une demande au serveur, et le serveur répond avec un message contenant les données demandées ou un message d'erreur si la demande ne peut être satisfaite. Les requêtes et les réponses HTTP sont composées d'en-têtes et d'un corps de message.

Les requêtes HTTP sont constituées de plusieurs parties, notamment :

1. Ligne de requête : Cela comprend la méthode HTTP (GET, POST, etc.), l'URL de la ressource demandée et la version HTTP utilisée.
2. En-têtes de requête : Ils fournissent des informations supplémentaires sur la requête, telles que l'agent utilisateur effectuant la demande, le type de données demandées et toute information d'authentification.
3. Corps de la requête : Il contient toutes les données envoyées au serveur, telles que des données de formulaire ou des téléchargements de fichiers.

Les réponses HTTP ont également plusieurs parties, notamment :

1. Ligne d'état : Cela comprend la version HTTP, un code d'état indiquant si la requête a été réussie ou non, et un bref message expliquant l'état.
2. En-têtes de réponse : Ils fournissent des informations supplémentaires sur la réponse, telles que le type de contenu des données renvoyées et des instructions de mise en cache.
3. Corps de la réponse : Il contient les données renvoyées par le serveur, telles qu'une page HTML ou un objet JSON.

HTTP est un protocole sans état, ce qui signifie que chaque requête et réponse est indépendante de toutes les requêtes ou réponses précédentes. Pour maintenir l'état entre les requêtes, les applications Web utilisent souvent des cookies ou d'autres mécanismes pour stocker des informations sur la session du client.

* **Différents types de requêtes http**

Il existe plusieurs types de requêtes HTTP, chacun servant un objectif spécifique lors de la communication entre un client et un serveur. Les méthodes de requête HTTP les plus courantes sont les suivantes :

1. GET : La méthode GET est utilisée pour récupérer des données du serveur. Lorsqu'un client envoie une requête GET, il demande au serveur de renvoyer une représentation de la ressource spécifiée. Les requêtes GET ne doivent récupérer que des données et ne doivent avoir aucun autre effet sur le serveur.
2. POST : La méthode POST est utilisée pour envoyer des données au serveur afin de créer une nouvelle ressource. Elle est couramment utilisée pour soumettre des données à partir de formulaires sur des pages Web et peut également être utilisée à d'autres fins où des données doivent être envoyées au serveur.
3. PUT : La méthode PUT est utilisée pour mettre à jour une ressource existante sur le serveur. Elle envoie des données au serveur, qui sont ensuite utilisées pour mettre à jour la ressource spécifiée.
4. DELETE : La méthode DELETE est utilisée pour demander la suppression d'une ressource spécifiée sur le serveur.
5. PATCH : La méthode PATCH est utilisée pour appliquer des modifications partielles à une ressource sur le serveur. Elle envoie les modifications spécifiées au serveur, qui les applique à la ressource existante.
6. HEAD : La méthode HEAD est similaire à GET, mais elle demande uniquement les en-têtes de la ressource, sans le corps du message. Cela permet au client de vérifier les en-têtes (comme la taille ou le type de contenu) avant de récupérer la ressource complète avec une requête GET.
7. OPTIONS : La méthode OPTIONS est utilisée pour obtenir les options de communication disponibles pour une ressource spécifiée sur le serveur. Cela permet au client de découvrir quelles méthodes de requête et quels en-têtes sont pris en charge par le serveur pour cette ressource.

Ces différentes méthodes de requête HTTP permettent aux clients et aux serveurs de communiquer de manière flexible et d'effectuer diverses opérations sur les ressources du serveur.

1. **Création d’un serveur**

Ce code crée une variable constante appelée "http" et lui assigne le module http. Vous pouvez ensuite utiliser cette variable pour créer un serveur HTTP, effectuer des requêtes HTTP et gérer les réponses HTTP dans votre application Node.js. Par exemple, pour créer un serveur HTTP en utilisant le module http, vous pouvez utiliser le code suivant :

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");

});

Ce code crée un serveur HTTP qui écoute sur le port 3000 et répond à toutes les requêtes avec un message en texte brut disant "Hello, World!". Lorsqu'un client fait une requête à ce serveur, la fonction createServer crée une nouvelle instance de serveur HTTP, et la fonction de rappel gère la requête entrante et envoie une réponse de retour au client.

* 1. **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. **L’objet Response(Repose)**

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.

const http = require("http")

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

  console.log(req.url,req.method);

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

res.write(‘Hello Ninja’)

  res.end()

});

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

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

});

1. **Retourner une page HTML**

On crée note fichier Html (Dans ce cas **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. **Statuts 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.

1. **Basic Routing et la redirection**

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.

 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. **Express**
2. **Définition**

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.

1. **Création d’une application express**

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>')

})

1. **Middlware**

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.

Une image contenant texte, signe, capture d’écran

Description générée automatiquementUne image contenant diagramme

Description générée automatiquement

Une 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. **Redirects & 404’s**
2. **Redirect**

// 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 :

const express = require("express");

.

.

.

// 404 page

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

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

})

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. **Rest api**
2. **Introduction**

Une image contenant texte, Visage humain, personne, capture d’écran

Description générée automatiquement

1. **C’est quoi un rest api**

Un REST API (ou RESTful API) est un type d'API (Application Programming Interface) qui suit les principes et les contraintes de l'architecture REST (Representational State Transfer).

L'architecture REST est une approche standardisée pour la conception de systèmes distribués et de services web qui permettent aux clients de communiquer avec des serveurs via des requêtes HTTP (Hypertext Transfer Protocol) en utilisant des méthodes standardisées telles que GET, POST, PUT, DELETE, etc.

Un REST API permet donc aux clients de communiquer avec un serveur web en utilisant des requêtes HTTP, en envoyant et en recevant des données dans un format standardisé tel que JSON ou XML. Les ressources sont identifiées par des URI (Uniform Resource Identifiers) et manipulées via les méthodes standardisées.

Les API RESTful sont largement utilisées pour fournir des services web tels que des interfaces de programmation pour les applications mobiles, les sites web, les services cloud, etc. Ils offrent une interface simple, cohérente et facile à utiliser pour les développeurs de clients et de serveurs.

1. **différence entre rest et restful api**

La différence entre REST et RESTful API est souvent une source de confusion, car les termes sont souvent utilisés de manière interchangeable. Cependant, voici une explication de la différence entre les deux :

* REST (Representational State Transfer) est un style architectural qui définit un ensemble de contraintes pour la conception de systèmes distribués, tels que les services web. Il s'agit d'une approche standardisée pour la conception de systèmes distribués qui permet aux clients de communiquer avec des serveurs via des requêtes HTTP en utilisant des méthodes standardisées telles que GET, POST, PUT, DELETE, etc.
* RESTful API, quant à lui, est un terme utilisé pour décrire une API (Application Programming Interface) qui suit les principes et les contraintes de l'architecture REST. Une API RESTful fournit donc une interface web qui permet aux clients de communiquer avec un serveur web en utilisant des requêtes HTTP et en utilisant des ressources identifiées par des URI (Uniform Resource Identifiers) et manipulées via des méthodes standardisées.

En résumé, REST est une architecture de conception de systèmes distribués, tandis qu'une API RESTful est une implémentation d'une interface web qui suit les principes de cette architecture.

1. **MVC**

Le modèle MVC (Modèle-Vue-Contrôleur) est un motif architectural largement utilisé dans le développement de logiciels, notamment dans le développement web. Il vise à séparer les responsabilités et à organiser le code de manière modulaire et maintenable.

Voici une description des différents composants du modèle MVC :

1. Modèle (Model) : Le modèle représente les données et la logique métier de l'application. Il traite les opérations liées à la manipulation des données, telles que la lecture, l'écriture, la mise à jour et la suppression. Le modèle est indépendant de l'interface utilisateur et ne connaît pas les détails de l'affichage ou de l'interaction avec l'utilisateur.
2. Vue (View) : La vue est responsable de l'affichage des données et de l'interaction avec l'utilisateur. Elle reçoit les données du modèle et les présente à l'utilisateur de manière appropriée, généralement en générant une interface utilisateur (par exemple, une page HTML) ou en envoyant des données au client. La vue ne contient pas de logique métier complexe et se concentre uniquement sur l'affichage des données.
3. Contrôleur (Controller) : Le contrôleur agit en tant qu'intermédiaire entre le modèle et la vue. Il reçoit les entrées de l'utilisateur, traite les actions et met à jour le modèle en conséquence. Il récupère également les données nécessaires du modèle et les transmet à la vue pour l'affichage. Le contrôleur gère la logique de l'application et coordonne les interactions entre le modèle et la vue.

Le modèle MVC permet de séparer clairement les responsabilités et d'améliorer la maintenabilité du code. Il facilite également la réutilisation des composants, car le modèle, la vue et le contrôleur peuvent être modifiés indépendamment les uns des autres.

Dans une application web, le modèle MVC est souvent utilisé pour structurer l'architecture du serveur (backend) et du client (frontend). Le modèle représente la logique de traitement des données côté serveur, la vue est responsable de la présentation des données au client, et le contrôleur gère les requêtes du client et coordonne les actions entre le modèle et la vue.

En résumé, le modèle MVC est un modèle architectural qui divise une application en trois composants distincts (Modèle, Vue et Contrôleur) pour faciliter la séparation des responsabilités et améliorer la maintenabilité du code.

1. **Ce qu’on va construire de l’un de nos projets**

Une image contenant texte, Visage humain, capture d’écran, homme

Description générée automatiquement

1. **Routes**

Le "Express Router" est un outil intégré à Express.js qui permet de découper et d'organiser les routes de votre application en modules séparés. Cela facilite la gestion des routes et des fonctionnalités de votre application, en les organisant de manière modulaire.

L'utilisation du "Express Router" permet de définir des groupes de routes et de gérer les chemins d'accès (endpoints) associés dans des fichiers séparés. Cela rend le code plus lisible, plus facile à maintenir et facilite la collaboration entre les membres de l'équipe travaillant sur l'application.

Voici un exemple de code illustrant comment utiliser le "Express Router" dans une application Express.js :

// app.js (ou votre fichier principal)

const express = require('express');

const app = express();

// Importer le routeur

const orderRouter = require('./routes'); // Chemin vers votre fichier de routeur

// Utiliser le routeur pour gérer les routes

app.use('orders', orderRouter);

// Démarrer le serveur

app.listen(3000, () => {

  console.log('Le serveur est en cours d\'exécution sur le port 3000');

});

// routes.js (Votre fichier de routeur)

const express = require('express');

const router = express.Router();

// Définir une route pour la page d'accueil

router.get('/', (req, res) => {

  res.send(All order);

});

// Définir une autre route

router.get('/:orderId', (req, res) => {

  res.send(‘une order’);

});

// Exporter le routeur pour l'utiliser dans app.js

module.exports = router;

Dans cet exemple, nous avons défini un routeur dans un fichier séparé appelé "routes.js". Ce routeur gère les routes pour all order et la page une order". Ensuite, dans notre fichier principal "app.js", nous avons importé ce routeur et l'avons utilisé avec la méthode **app.use()** pour gérer les routes associées. Cela permet de garder le code organisé et de séparer les différentes fonctionnalités de l'application.

1. **Handling errors et improving the project setup**
2. **Project setup**

On utilise nodemon et morgan

1. **Handling Errors**

Comme on a déjà explique . on met tous a la fin normalement tu sais pourquoi .

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

  const error = new Error("Not found");

  error.status = 404;

  next(error);

});

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

  res.status(error.status || 500);

  res.json({

    error: {

      message: error.message,

    },

  });

});

1. **Parsing the body & handling the cors**
   1. **Parsing the body with (body-parser)**

**body-parser** est un middleware populaire pour Node.js qui permet de traiter les données envoyées dans une requête HTTP POST ou PUT et de les rendre disponibles sous forme d'objet JavaScript facilement exploitable.

Lorsqu'un client envoie des données à un serveur Node.js via une requête HTTP POST ou PUT, les données sont envoyées dans le corps de la requête HTTP. Cependant, le corps de la requête n'est pas automatiquement analysé et transformé en un objet JavaScript exploitable. C'est là que **body-parser** entre en jeu.

**body-parser** analyse le corps de la requête HTTP et le transforme en un objet JavaScript qui peut être utilisé dans le code de l'application. Cela facilite grandement le traitement des données envoyées par les clients et simplifie la gestion des requêtes HTTP.

Voici un exemple de code pour utiliser **body-parser** dans une application Express.js :

const express = require('express');

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

const app = express();

app.use(bodyParser.urlencoded({ extended: false }));

app.use(bodyParser.json());

app.post('/submit', (req, res) => {

  const name = req.body.name;

  const email = req.body.email;

  res.send(`Merci pour votre envoi, ${name} (${email})`);

});

app.listen(3000, () => {

  console.log('Server started on port 3000');

});

Dans cet exemple, nous utilisons **body-parser** pour analyser les données envoyées dans le corps d'une requête HTTP POST envoyée à l'URL **/submit**. Nous utilisons **app.use()** pour ajouter les middleware **bodyParser.urlencoded()** et **bodyParser.json()** à notre application Express. **bodyParser.urlencoded()** analyse les données envoyées en tant que paramètres de requête HTML et **bodyParser.json()** analyse les données envoyées en tant que JSON.

Nous pouvons ensuite accéder aux données envoyées dans la requête HTTP POST via **req.body**. Dans cet exemple, nous utilisons **req.body.name** et **req.body.email** pour accéder aux valeurs envoyées dans les champs de formulaire HTML. Enfin, nous renvoyons une réponse HTTP avec un message de remerciement qui utilise les valeurs envoyées par le client.

* 1. **Fixing Cors errors**

Une image contenant texte, capture d’écran

Description générée automatiquement

Une erreur de CORS (Cross-Origin Resource Sharing) se produit lorsqu'une requête HTTP émanant d'un navigateur web tente d'accéder à une ressource située sur un autre domaine que celui qui a émis la requête.

Les navigateurs web imposent une politique de sécurité Same-Origin, qui interdit à une page web d'accéder à des ressources sur un autre domaine que celui à partir duquel elle a été chargée. Cette politique est mise en place pour empêcher les attaques par injection de code malveillant et autres formes de piratage.

Cependant, il peut y avoir des cas où une application web doit accéder à des ressources situées sur un autre domaine, par exemple lorsque l'application web doit récupérer des données depuis une API distante. C'est là qu'intervient CORS.

CORS est un mécanisme qui permet aux serveurs web de spécifier les domaines à partir desquels les requêtes HTTP sont autorisées à accéder à leurs ressources. Pour permettre une requête HTTP à traverser les limites de domaine, le serveur doit ajouter les entêtes appropriés dans la réponse HTTP. Ces entêtes permettent au navigateur de savoir si la ressource demandée peut être accédée ou non.

Si le serveur ne fournit pas les en-têtes appropriés, le navigateur empêchera l'accès à la ressource et générera une erreur de CORS. Cette erreur peut être visible dans la console du navigateur ou renvoyée au client sous forme de message d'erreur.

Pour résoudre une erreur de CORS, le serveur doit envoyer les entêtes CORS appropriés dans la réponse HTTP. Cela peut inclure l'ajout de l'en-tête **Access-Control-Allow-Origin** pour spécifier les domaines autorisés à accéder à la ressource, ou l'utilisation de middleware dans le cas d'une application Node.js, comme **cors** ou en configurant manuellement les en-têtes nécessaires.

Haut du formulaire

Pour résoudre une erreur de CORS dans une application Node.js, il est nécessaire d'ajouter les en-têtes CORS appropriés dans les réponses HTTP. Il existe plusieurs façons de le faire, mais voici deux méthodes couramment utilisées :

**Méthode 1 : Utiliser le module cors**

Le module **cors** est un middleware pour Express.js qui permet de gérer les requêtes CORS. Voici un exemple d'utilisation :

const express = require('express');

const cors = require('cors');

const app = express();

app.use(cors());

// définir les routes de l'application ici

Cette utilisation simple de **cors** autorisera toutes les requêtes provenant de n'importe quel domaine à accéder aux ressources de votre application. Vous pouvez également spécifier des options pour restreindre l'accès à certaines origines en utilisant la méthode **cors(options)**. Voir la documentation du module **cors** pour plus de détails.

### Méthode 2 : Ajouter manuellement les en-têtes CORS

Si vous ne voulez pas utiliser de middleware, vous pouvez également ajouter manuellement les en-têtes CORS dans votre application Express.js en utilisant un middleware personnalisé. Voici un exemple de middleware qui ajoute les en-têtes CORS nécessaires pour autoriser les requêtes de tous les domaines :

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

  res.setHeader('Access-Control-Allow-Origin', '\*');

  res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE');

  res.setHeader('Access-Control-Allow-Headers', 'Content-Type, Authorization');

  next();

});

Ce middleware ajoute les en-têtes CORS suivants :

* **Access-Control-Allow-Origin** : Autorise les requêtes de tous les domaines en utilisant l'astérisque **\***. Vous pouvez spécifier des domaines spécifiques si vous le souhaitez.
* **Access-Control-Allow-Methods** : Spécifie les méthodes HTTP autorisées pour accéder à la ressource. Dans cet exemple, toutes les méthodes HTTP (GET, POST, PUT, DELETE) sont autorisées.
* **Access-Control-Allow-Headers** : Spécifie les en-têtes HTTP autorisés dans la requête. Dans cet exemple, le type de contenu (**Content-Type**) et l'autorisation (**Authorization**) sont autorisés.

Ces en-têtes peuvent être personnalisés en fonction de vos besoins. Il est important de noter que l'ajout de ces en-têtes peut constituer une vulnérabilité de sécurité si elles sont configurées de manière incorrecte. Il est donc important de prendre les mesures de sécurité nécessaires.

**In my example :**

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

  res.header("Access-Control-Allow-Origin", "\*");

  res.header(

    "Access-Control-Allow-headers",

    "Origin,X-Requested-With,Content-Type,Accept,Authorization"

  );

  if (req.method === "OPTIONS") {

    res.header("Access-Control-Allow-Methods", "PUT,POST,PATCH,DELETE");

    return res.status(200).json({});

  }

  next();

});

1. **Uploading une image using Multer**

Multer est un middleware pour Node.js qui permet de gérer facilement les fichiers téléchargés (upload) à partir d'un formulaire HTML ou d'une requête HTTP. En utilisant Multer, les développeurs peuvent facilement télécharger des fichiers et gérer les erreurs associées à cette opération.

Voici quelques raisons pour lesquelles on utilise Multer en Node.js :

1. Traitement des fichiers : Multer permet de gérer le traitement des fichiers téléchargés de manière simple et efficace. Il peut gérer les fichiers de tous les types MIME (ex : image, audio, vidéo, etc.) et permet de définir des limites de taille et de nombre de fichiers.
2. Sécurité : Multer permet de renommer automatiquement les fichiers pour éviter les collisions de noms, ce qui peut empêcher les attaques par injection de fichiers.
3. Compatibilité avec Express : Multer est compatible avec le framework Express de Node.js, ce qui facilite l'intégration dans des applications Express existantes.
4. Configuration flexible : Multer offre une configuration flexible pour répondre aux besoins spécifiques des applications. Il permet de définir les emplacements de stockage des fichiers, les noms de fichiers personnalisés, etc.
5. Support de middleware : Multer peut être utilisé comme middleware, ce qui facilite l'intégration avec d'autres middleware et assure une gestion efficace des fichiers.

En résumé, Multer est un outil pratique pour gérer les fichiers téléchargés dans les applications Node.js. Il offre une grande flexibilité, une sécurité accrue et est facile à intégrer dans des applications existantes

const express = require('express');

const multer = require('multer');

// Création de l'instance du middleware multer

const upload = multer({ dest: 'uploads/' }); // Spécifie le dossier de destination des fichiers téléchargés

const app = express();

// Endpoint pour gérer le téléchargement du fichier

app.post('/upload', upload.single('file'), (req, res) => {

  // `upload.single('file')` indique à multer de traiter un seul fichier avec le nom de champ 'file' dans le formulaire

  // Le fichier téléchargé est accessible via `req.file`

  if (!req.file) {

    return res.status(400).send('Aucun fichier téléchargé.');

  }

  // Faire quelque chose avec le fichier (par exemple, le déplacer, le traiter, etc.)

  res.status(200).send('Fichier téléchargé avec succès.');

});

// Démarrer le serveur

app.listen(3000, () => {

  console.log('Serveur démarré sur le port 3000');

});

Dans cet exemple, nous utilisons Express pour créer une application web. Lorsque l'utilisateur envoie une requête POST à l'endpoint **/upload** avec un fichier, le middleware **multer** intervient pour traiter le fichier téléchargé.

**upload.single('file')** spécifie que nous attendons un seul fichier avec le nom de champ **'file'** dans le formulaire. Le fichier téléchargé sera stocké dans le dossier **'uploads/'** (vous devez vous assurer que ce dossier existe).

Si aucun fichier n'est téléchargé ou si le téléchargement échoue, une réponse d'erreur est renvoyée avec le statut 400. Sinon, vous pouvez effectuer les actions souhaitées avec le fichier téléchargé (par exemple, le déplacer, le renommer, l'enregistrer dans une base de données, etc.), puis renvoyer une réponse réussie avec le statut 200.

N'oubliez pas d'installer le module **multer** en exécutant **npm install multer** avant d'exécuter cet exemple.

1. **Adding user signUp**
2. **Bcrypt**

bcrypt est un package populaire en Node.js qui permet de hasher et de vérifier les mots de passe de manière sécurisée. Il utilise une fonction de hachage de mots de passe adaptative et cryptographiquement solide pour protéger les informations d'identification des utilisateurs.

1. **jsonWebToken**

Le package **jsonwebtoken** est une bibliothèque populaire en Node.js qui permet de générer et de vérifier les tokens JWT. Il fournit des méthodes simples et pratiques pour travailler avec les tokens JWT, notamment pour la création, la vérification et la gestion des informations d'authentification.

Voici un exemple d'utilisation du package **jsonwebtoken** en Node.js :

const jwt = require('jsonwebtoken');

// Exemple de création d'un token JWT

const payload = { id: 1, username: 'utilisateur' };

const secretKey = 'votre\_clé\_secrète';

const token = jwt.sign(payload, secretKey, { expiresIn: '1h' });

console.log('Token JWT :', token);

// Exemple de vérification d'un token JWT

jwt.verify(token, secretKey, (err, decoded) => {

  if (err) {

    console.error(err);

    return;

  }

  console.log('Informations du token décrypté :', decoded);

});

La méthode **sign** est utilisée pour créer un token JWT en spécifiant les informations d'authentification (**payload**), une clé secrète (**secretKey**) pour signer le token, et éventuellement d'autres options telles que la durée de validité (**expiresIn**).

La méthode **verify** permet de vérifier et de décrypter un token JWT en spécifiant le token, la clé secrète et une fonction de rappel (**callback**) qui est appelée avec les informations décryptées (**decoded**) du token si la vérification est réussie.

Le package **jsonwebtoken** offre également d'autres fonctionnalités, telles que la possibilité de spécifier des algorithmes de signature personnalisés, de gérer des options avancées pour la création et la vérification des tokens, ainsi que de manipuler les différentes parties du token (en-tête, charges utiles, signature).

Il est important de noter que la clé secrète utilisée pour signer et vérifier les tokens JWT doit être conservée de manière sécurisée et ne pas être exposée publiquement.

1. **User signUp**

L'authentification dans un service RESTful en Node.js est un processus qui permet de vérifier l'identité d'un utilisateur avant de lui accorder l'accès aux ressources protégées. Il existe plusieurs méthodes d'authentification couramment utilisées dans les applications Node.js. Voici quelques approches populaires :

1. Authentification basée sur les tokens JWT (JSON Web Tokens) : Dans ce cas, l'utilisateur se connecte avec ses identifiants (par exemple, nom d'utilisateur et mot de passe) et reçoit un token JWT signé. Ce token est ensuite envoyé avec chaque requête HTTP dans l'en-tête Authorization. Le serveur peut alors vérifier l'intégrité du token et extraire les informations d'authentification pour autoriser l'accès aux ressources.
2. Authentification basée sur les sessions : Cette méthode utilise des cookies pour suivre l'état d'authentification de l'utilisateur. Lorsque l'utilisateur se connecte, le serveur crée une session et renvoie un cookie contenant l'identifiant de session au client. Le client envoie ensuite ce cookie avec chaque requête HTTP, et le serveur vérifie si l'identifiant de session correspond à une session valide pour autoriser l'accès.
3. Authentification basée sur les tokens d'accès : Dans ce cas, lors de la connexion, l'utilisateur reçoit un token d'accès qui est envoyé avec chaque requête pour prouver son identité. Le serveur vérifie la validité du token d'accès à chaque requête pour autoriser l'accès.

Voici un exemple simple d'implémentation d'authentification basée sur les tokens JWT en utilisant le package **jsonwebtoken** en Node.js :

const express = require('express');

const jwt = require('jsonwebtoken');

const app = express();

const secretKey = 'votre\_clé\_secrète';

// Middleware pour vérifier l'authentification

function authenticate(req, res, next) {

  const token = req.headers.authorization;

  if (!token) {

    return res.status(401).json({ message: 'Token manquant' });

  }

  jwt.verify(token, secretKey, (err, decoded) => {

    if (err) {

      return res.status(403).json({ message: 'Token invalide' });

    }

    // Ajoute les informations d'authentification décodées à la requête pour une utilisation ultérieure

    req.user = decoded;

    next();

  });

}

// Route protégée nécessitant une authentification

app.get('/api/ressource-protegee', authenticate, (req, res) => {

  // Utilisez req.user pour accéder aux informations de l'utilisateur authentifié

  res.json({ message: 'Ressource protégée' });

});

// Route pour se connecter et obtenir le token JWT

app.post('/api/login', (req, res) => {

  // Vérifiez les identifiants de l'utilisateur et générez le token JWT en conséquence

  const user = { id: 1, username: 'utilisateur' };

  const token = jwt.sign(user, secretKey);

  res.json({ token });

});

// Démarrer le serveur

app.listen(3000, () => {

  console.log('Serveur démarré sur le port 3000');

});

N'oubliez pas de gérer correctement la sécurité de votre application, notamment en sécurisant la transmission des données sensibles, en stockant les mots de passe de manière sécurisée et en mettant en place des mesures de protection contre les attaques courantes (par exemple, les attaques par force brute).

Une image contenant texte, Visage humain, personne, capture d’écran

Description générée automatiquement

Une image contenant texte, Visage humain, personne, capture d’écran

Description générée automatiquement