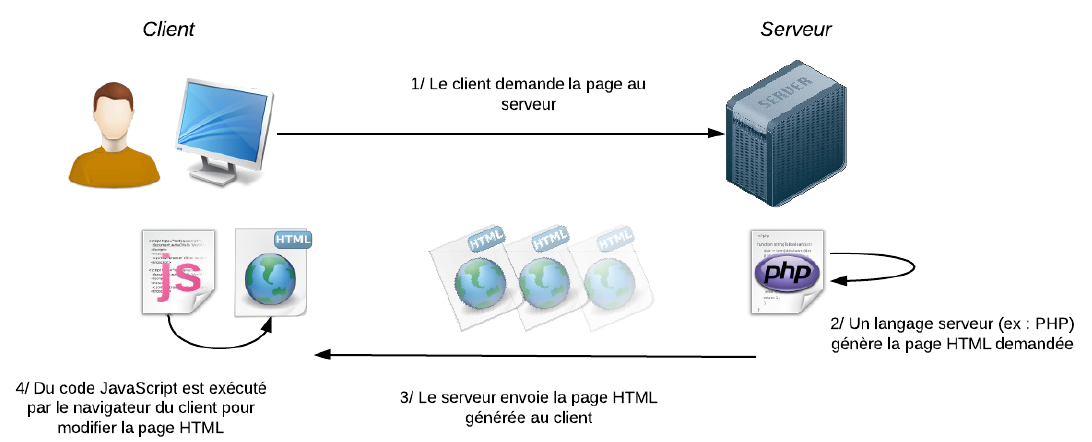
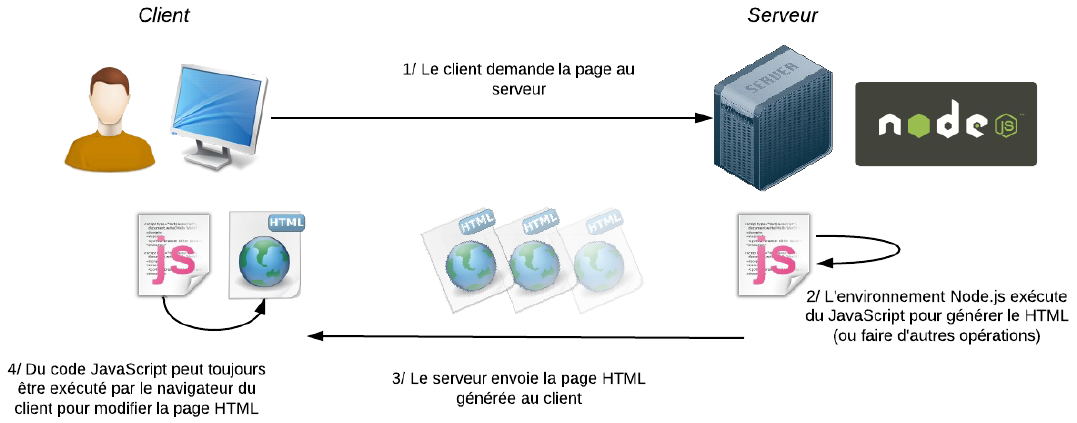
Node.js / Vue.js

**Architecture avec un serveur Apache ou Nginx :**



**Architecture Node.js :**

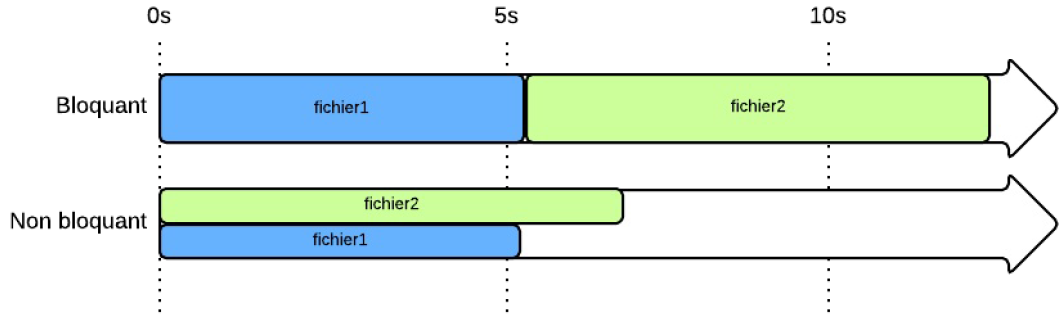


**REMARQUES :**

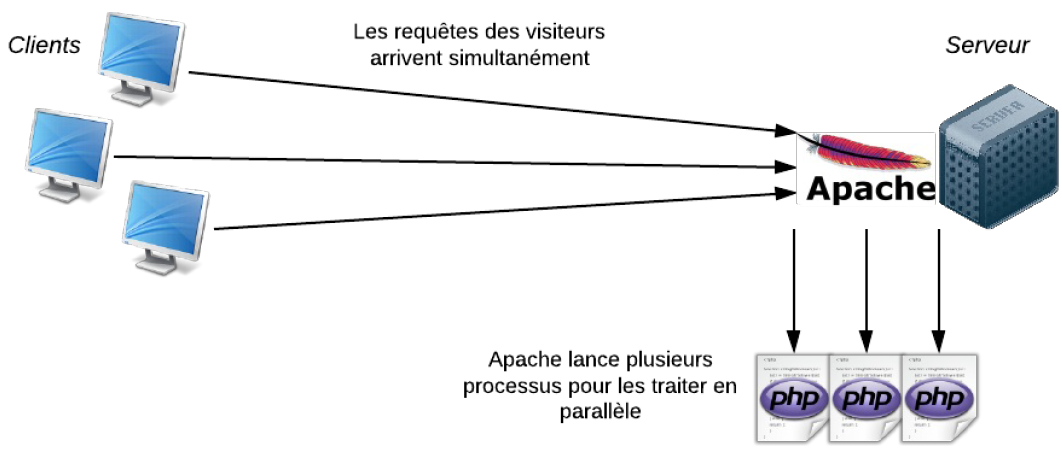
* Node.js n'est pas un framework.
* Node.js est un environnement bas niveau asynchrone.
* Node.js est basé sur Javascript

**RAPPEL : Comparaison des actions synchrone / asynchrone :**

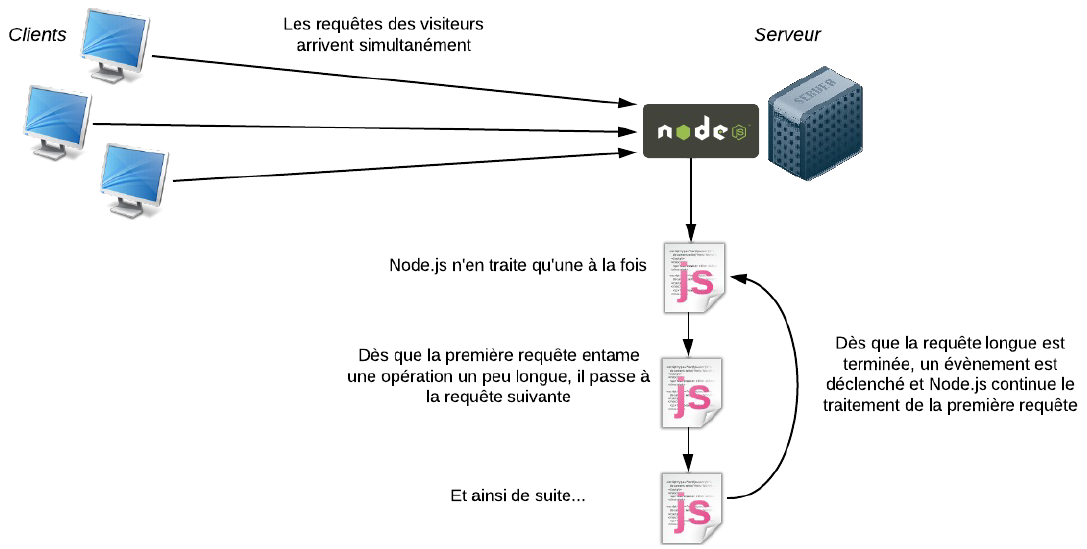
Exemple d’un téléchargement de deux fichiers en mode synchrone (bloquant) et asynchrone (non bloquant) :



**Serveur Apache : Traitement en parallèle des requêtes :**



**Node.js : Traitement asynchrone d’une seule requête à la fois :**



La solution apportée par Node.js repose sur trois bases fondamentales :

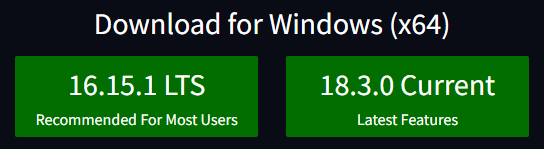
* Le moteur Javascript V8 développé par Google.
* Une boucle d’évènements permettant d’exécuter plusieurs opérations simultanées de façon asynchrone et non bloquante sur un seul fil (thread)
* Une bibliothèque de bas niveau libluv qui elle est multithreading.

A noter que Node.js s’avère moins performant que PHP, Python pour les applications gourmandes en CPU. (Data science, machine learning)

Installation de Node.js

Sur le site de Node : <https://nodejs.org/en/>

Cliquer sur le bouton de gauche (version 16.15.0 – stable)



Une fois chargé, lancer l’installeur.

Tester l’installation

* Créer un dossier nodejs
* Dans ce dossier, insérer un fichier test.js dans lequel écrire cette commande :

console.log('Bonjour Node.js !');

* Ouvrir une console
* Naviguer jusqu’au répertoire nodejs
* Entrer la commande : node test.js



Créer un serveur http simple

* Créer un fichier server.js dans lequel on écrit ce code :

let *http* = require('http');

let hostname = '127.0.0.1';

let port = 5000;

let server = *http*.createServer(function(*req*, *res*) {

*res*.statusCode = 200;

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

*res*.write(

    '<!DOCTYPE html>'+

    '<html>'+

    ' <head>'+

    ' <meta charset="utf-8" />'+

    ' <title>Ma page Node.js !</title>'+

    ' </head>'+

    ' <body>'+

    ' <p>Voici un paragraphe <strong>HTML</strong> !</p>'+

    ' </body>'+

    '</html>'

    );

*res*.end();

});

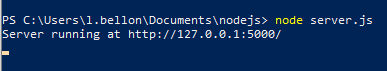
server.listen(port, hostname, function() {

  console.log("Server running at http://" + hostname + ":" + port + "/");

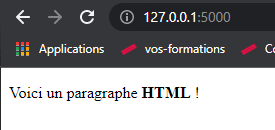
});

Nous verrons plus loin que les paramètres req et res contiennent respectivement les valeurs passées par l’url et la réponse du serveur node. Ci-dessus, cette réponse est enrichie par un statut (200), par un header et par un bout de code html.

* Depuis une console, lancer le serveur



* Vérifier que côté client on obtient :



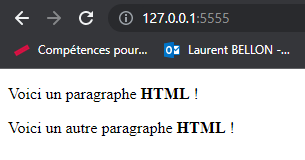
* Procéder à quelques tests complémentaires :

>> Ajouter un autre paragraphe html dans le code server.js

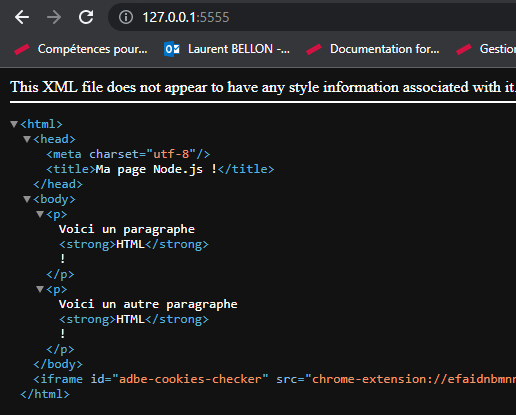
    ' <p>Voici un autre paragraphe <strong>HTML</strong> !</p>'+

>> Rafraichir le navigateur. Rien ne se passe. Comment y remédier ?

>> Modifier le port d’écoute de 5000 en 5555. Vérifier qu’on obtient :



>> Modifier le code de server.js pour que le client reçoive la réponse au format balisé (XML) et obtenir ceci :



Connexion à une base MySQL

* Dans le répertoire nodejs, créer un sous-répertoire monprojetsql
* A partir d’une console, naviguer jusqu’à monprojetsql et taper npm install mysql
* Dans monprojetsql, créer un fichier connexion.js dans lequel saisir ce code :

let mysql = require('mysql');

console.log('Get connection ...');

var connection = mysql.createConnection({

  database: 'testnodejs',

  host: "localhost",

  user: "root",

  password: "root"

});

connection.connect(function(*err*) {

  if (*err*) throw *err*;

  console.log("Connected!");

});

* Sous phpMyAdmin, créer une base nommée testnodejs
* Depuis une console, vérifier que la connexion est opérationnelle :



Modifier le code de connexion.js de façon à créer une table ‘**employés’** comportant les champs suivants :

* + - Id – integer - indice auto incrémenté et clé primaire
    - FirstName – varchar – Prénom de l’employé
    - LastName – varchar – Nom de l’employé
    - Hire\_date – Date – La date d’entrée de l’employé (YYYY-MM-DD)

let mysql = require('mysql');

console.log('Get connection ...');

var connection = mysql.createConnection({

  database: "testnodejs",

  host: "localhost",

  user: "root",

  password: "root"

});

connection.connect(function(*err*) {

  if (*err*) throw *err*;

  console.log("Connected!");

   // Drop EMPLOYEES table if Exists!!

   var sql1 = "DROP TABLE IF EXISTS Employes ";

   connection.query(sql1, function(*err*, *results*) {

       if (*err*) throw *err*;

       console.log("Table Employes dropped");

   });

   // Create EMPLOYEES Table.

   var sql2 = "CREATE TABLE Employes " +

       " Ecrire la suite de la requête SQL "

   connection.query(sql2, function(*err*, *results*) {

       if (*err*) throw *err*;

       console.log("Table Employés créée");

   });

});

* Peupler la table employés en insérant ces lignes de codes dans la méthode connection.connect() :

NOTA : Les dates du tableau hireDates ne sont pas au format attendu par la table ‘employés’ ; utiliser la directive SQL : STR\_TO\_DATE(<input\_string>, <input\_format>

   var firstNames = ["John", "Jack", "Paul"];

   var fullNames = ["Hikes", "Smith", "Gates"];

   var hireDates = ["22/10/2001", "11/11/2000", "12/12/1990"];

   // Insert Datas to EMPLOYEES.

   for (var i = 0; i < empNos.length; i++) {

       var sql3 = "Ecrire ici la requête d’insertion…"

       connection.query(sql3, function(*err*, *results*) {

           if (*err*) throw *err*;

           console.log("Insert a record!");

       });

   }

* Vérifier que les données sont bien insérées en base.
* Créer un fichier readdata.js de façon à récupérer tous les employés :

let mysql = require('mysql');

console.log('Get connection ...');

var connection = mysql.createConnection({

  database: 'testnodejs',

  host: "localhost",

  user: "root",

  password: "root"

});

connection.connect(function(*err*) {

  if (*err*) throw *err*;

  console.log("Connected!");

   var sql\_template = "Select \* from ??";

   var replaces = ['Employes'];

   sql = mysql.format(sql\_template, replaces);

   connection.query(sql, function(*err*, *rows*, *fields*) {

       if (*err*) throw *err*;

>>>> Ecrire ici la boucle qui permet d’afficher dans la console le résultat de recherche (utiliser console.log())

   });

});

* Modifier les variables sql\_template et replaces de façon à ne récupérer que les employés dont la date d’entrée est postérieure au 20/11/1995 (attention au format de date).
* Depuis une console, vérifier le résultat de recherche obtenu.

Les routes

Sous node.js, la gestion des routes est nativement réalisée au moyen du module url.

* Créer un répertoire route sous nodejs et créer un fichier app.js contenant ce code :

let *http* = require('http');

let *url* = require('url');

let hostname = '127.0.0.1';

let port = 5000;

let server = *http*.createServer(function(*req*, *res*) {

    let page = *url*.parse(*req*.url).pathname;

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

    if (page == '/') {

*res*.statusCode = 200;

*res*.write(You are at home page');

    }

    else if (page == '/controller') {

*res*.statusCode = 200;

*res*.write(You are at the controller);

    }

else if (page == '/controller/method') {

*res*.statusCode = 200;

*res*.write(You are at the method of the controller);

    }

else {

*res*.statusCode = 404;

*res*.write(Page does not exist);

    }

*res*.end();

});

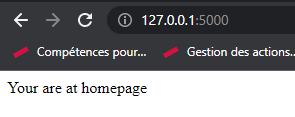
server.listen(port, hostname, function() {

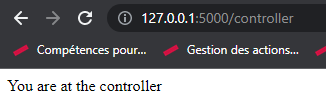
  console.log("Server running at http://" + hostname + ":" + port + "/");

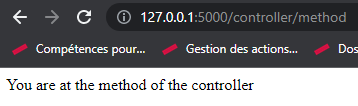
});

NOTE : La méthode parse est obsolète depuis quelques mois 😉, mais néanmoins fonctionnelle.

* Côté client, vérifier que les routes sont opérationnelles :







* Modifier le code app.js de façon à gérer les paramètres passés dans l’URL au moyen du module querystring :

let *http* = require('http');

let *url* = require('url');

let *querystring* = require('querystring');

let hostname = '127.0.0.1';

let port = 5000;

let server = *http*.createServer(function(*req*, *res*) {

    let page = *url*.parse(*req*.url).pathname;

    let params = *querystring*.parse(*url*.parse(*req*.url).query);

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

    if (page == '/') {

*res*.statusCode = 200;

*res*.write(You are at homepage');

    }

    else if (page == '/controller') {

*res*.statusCode = 200;

*res*.write(You are at the controller);

    }

else if (page == '/controller/method') {

*res*.statusCode = 200;

*res*.write(You are at the method of the controller);

    }

else {

*res*.statusCode = 404;

*res*.write(Page does not exist);

    }

    if ('prenom' in params && 'nom' in params) {

*res*.write('<br>You are ' + params['prenom'] + ' ' + params['nom']);

    } else {

*res*.write('<br>Fisrtname and lastname are empty !');

    }

*res*.end();

});

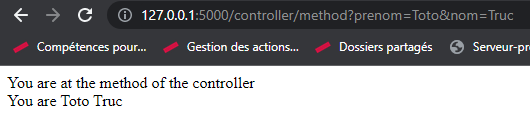
server.listen(port, hostname, function() {

  console.log("Server running at http://" + hostname + ":" + port + "/");

});

NOTE : La méthode queryStringParseparse est obsolète depuis quelques mois 😉, mais néanmoins fonctionnelle.

Résultat :



Installation du paquet « Express » :

La gestion des routes par les bibliothèques url et querystring est assez laborieuse et obsolète. Maintenant que nous avons compris comment ça marche, nous allons utiliser le framework : Express

* Créer un nouveau répertoire nommé node-express-app
* Sous une console, se rendre dans ce répertoire et taper npm init
* Répondre aux questions – Laisser une réponse vide au besoin.

Un fichier package.json a été créé. Il contient les dépendances qui vont être installées. Le répertoire node\_modules n’est en général jamais commité. Un autre developpeur peut récupérer le code d’une application. Le serveur aura en charge de faire un npm install pour récupérer les dépendances dont le projet a besoin pour fonctionner à partir du fichier package.json.

* Dans la console, taper : npm install express
* Créer un fichier index.js dans le répertoire node-express-app et y intégrer Express

const express = require('express');

const app = express();

La constante app est l’instance de la classe Express. Elle contiendra le serveur et ses méthodes.

* Définir le port d’écoute du serveur (par exemple 5000)

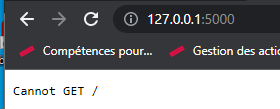
app.listen(5000, ()=> {

console.log(‘Server is listening @5000’) ;

});

* Depuis une console, vérifier qu’en tapant node index.js, on a bien le message ‘Server is listening’ qui s’affiche.

Côté client, un message d’erreur s’affiche, puisqu’aucune route n’est encore configurée.



Pour la suite, nous prendrons le cas d’une chaine de restaurants et la gestion des employés qui travaillent dans chaque restaurant.

* Lister les fonctionnalités :

>> Créer un restaurant

>> Lister l’ensemble des restaurants

>> Récupérer/modifier les détails d’un restaurant en particulier

>> Supprimer un restaurant

>> Créer un employé

>> Lister l’ensemble des employés d’un restaurant

>> Afficher les détails d’un employé

>> Supprimer un employé

* Créer les routes correspondant à ces fonctionnalités

>> POST /restaurant

>> GET /restaurants

>> GET /restaurants/:id

>> PUT /restaurants/:id

>> DELETE / restaurants/:id

>> POST /restaurants/:id/employe

>> GET /restaurants/:id/employes

>> GET /restaurants/:id/employes/:idEmploye

>> PUT /restaurants/:id/employes/:idEmploye

>> DELETE/restaurants/:id/employes/:idEmploye

* Créer une base de données **: restaurants** et coder la connexion dans index.js

let mysql = require('mysql');

console.log('Get connection ...');

var connection = mysql.createConnection({

  database: 'test',

  host: "localhost",

  user: "root",

  password: "root"

});

* Créer les tables **restaurants** et **employés** à partir de index.js

La table **restaurants** comprend les champs :

* + - Id – integer(11) - indice auto incrémenté et clé primaire
    - name – varchar(100) – nom du restaurant
    - city – varchar(100) – ville du restaurant
    - nbcouverts – integer(10) – le nombre de couverts
    - terrasse – varchar(3) – OUI ou NON
    - parking – varchar(3) – OUI ou NON

La table ‘**employés**’ comprend les champs :

* + - Id – integer(11)- indice auto incrémenté et clé primaire
    - first\_name – varchar(100) – Prénom de l’employé
    - last\_name – varchar(100) – Nom de l’employé
    - hire\_date – Date – La date d’entrée de l’employé (YYYY-MM-DD)
    - restaurant\_id – Id du restaurant où l’employé est affecté

Les routes sont définies au moyen des méthodes fournies par Express. Elles sont encapsulées dans la méthode connection.connect de l’objet connection :

connection.connect(function(*err*) {

    if (*err*) throw *err*;

    console.log("Connected!");

// Création de la table restaurants

…

// Création de la table employés

…

// Route /POST/restaurant

…

// Route /GET/restaurants

…

// Route /GET/restaurants/:id

…

// etc

});

* Créer la route POST/restaurant

La méthode **.post** d'Express permet de définir une route **POST**. Elle prend en premier paramètre une **String** qui définit la route à écouter et **une fonction callback**, qui est la fonction à exécuter si cette route est appelée. Cette fonction callback prend en paramètre **l'objet req**, qui reprend toutes les données fournies par la **requête**, et **l'objet res**, fourni par Express, qui contient les méthodes pour répondre à la requête qui vient d'arriver.

Il n’y a pas de UI pour saisir les données. Les requêtes seront envoyées par Postman.

**Pour pouvoir récupérer les données, il faut ajouter un middleware à** index.js **afin que node.js soit en mesure d’interpréter le body JSON qui lui est envoyé depuis Postman.**

// Middleware

app.use(express.json());

Le *middleware,* lors de son exécution, est lancé entre la réception de la requête et l’exécution des routes de façon à pouvoir interpréter la requête.

La route POST s’écrit de cette manière :

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

        let sql = "INSERT INTO restaurants (name, city, nbcouverts, terrasse, parking) " +

           " VALUES ('" + *req*.body.name + "', '"

                        + *req*.body.city + "', '"

                        + *req*.body.nbcouverts + "', '"

                        + *req*.body.terrasse + "', '"

                        + *req*.body.parking  + "')";

        connection.query(sql, function(*err*, *results*) {

           if (*err*) throw *err*;

           console.log("Insert a record !");

       });

*res*.status(200);

    });

* Tester la route POST au moyen de Postman.



* Créer deux autres restaurants
* Coder la route GET/restaurants

**Express** fournit la méthode .send( <body> ) qui envoie la réponse <body> sous requête http. <body> peut-être une chaine, un objet ou un tableau. La méthode renvoie un objet :

app.get('/restaurants', (*req*, *res*) => {

        var sql\_template = "Select \* from ?? ";

        var replaces = ['restaurants'];

        sql = mysql.format(sql\_template, replaces);

        connection.query(sql, function(*err*, *rows*) {

            if (*err*) throw *err*;

            res.send(rows)

        });

*res*.status(200);

    });

Démarrer le serveur node. Côté client taper l’url 127.0.0.1 :5000/restaurants. Vérifier le résultat dans le navigateur.

* Coder la route GET/restaurants/:id

    app.get('/restaurants/:id', (*req*, *res*) => {

        let id = parseInt(*req*.params.id);

        let sql\_template = "Select \* from ?? WHERE ?? = " + id;

        let replaces = ['restaurants', 'id'];

        sql = mysql.format(sql\_template, replaces);

        connection.query(sql, function(*err*, *row*, *fields*) {

            if (*err*) throw *err*;

*res*.send(*row*);

 });

*res*.status(200);

    });

* Vérifier le résultat dans le navigateur.
* Coder la route PUT/restaurants/:id

app.put('/restaurants/:id', (*req*, *res*) => {

        let id = parseInt(*req*.params.id);

        // Coder ici la requête

// Formater la requête

// Executer la requête

*res*.status(200);

    })

* Vérifier avec Postman.
* Coder la route DELETE/restaurants/:id

   app.delete('/restaurants/:id', (*req*, *res*) => {

        let id = parseInt(*req*.params.id)

        // Coder ici la requête

// Formater la requete

// Executer la requête

*res*.status(200);

    })

* Vérifier avec Postman.

Appliquer la même logique pour les routes restantes :

* POST /restaurants/:idResto/employe
* GET /restaurants/: idResto /employes
* GET /restaurants/: idResto /employes/:idEmploye
* PUT /restaurants/: idResto /employes/:idEmploye
* DELETE/restaurants/: idResto /employes/:idEmploye

Avec Postman, créer quelques employés.

En toute rigueur, lorsqu’un restaurant est supprimé, les employés doivent l’être aussi.

* Modifier le code dans ce sens.

Les modules

Ils sont placés dans le répertoire node\_modules à la racine du site. Voici un exemple simple.

Code de monmodule.js

var direBonjour = function() {

    console.log('Bonjour !');

}

var direBye = function() {

    console.log('Bye bye !');

}

*exports*.direBonjour = direBonjour;

*exports*.direBye = direBye;

Appel dans app.js

app.get('/hello', function(*request*, *response*) {

    monmodule.direBonjour();

    monmodule.direBye();

Résultat de l’url : http://localhost:5000/hello



* Refactoriser le code de l’application node-express-app en séparant le code en trois modules :
* Connexion à la base
* Création des tables
* Paramétrage des routes

Application front en Vue.js

**Vue.js** est un framework Javascript. **Vue.js** peut être appliqué sur la totalité d'un projet ou seulement sur une partie.

Il va s’agir de récupérer les données Back-end depuis la base de données et les afficher en front avec deux méthodes : une **page html** simple et une **application front-end**.

Récupérer les données Back et les afficher en Front

1. **A partir une page HTML**

* Sous le répertoire nodejs, créer un fichier html nommé demo-axios.html.
* Dans la balise <head> faire un appel cdn à **vue.js**

 <script *src*="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>

Une méthode courante consiste à utiliser la bibliothèque **Axios** pour interroger un serveur back. Il s’agit d’un client http basé sur les Promesses.

* Dans la balise <head> faire un appel cdn à axios

<script *src*="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>

* Dans la balise <body>, Créer une balise <div id="app">
* Créer une liste en utilisant une boucle **v-for** parcourant un tableau **restaurants**

        <div *id*="app">

            <ul>

                <li v-for="restaurant in restaurants">{{ restaurant }}</li>

            </ul>

        </div>

* A la suite de la balise <div>, créer une balise <script> dans laquelle
* Instancier une classe Vue() pointant sur l’élément ‘’#app’’
* Passer par l’élément data un tableau de chaines de caractères

       <script>

            let app = **new** Vue({

                el: "#app",

                data: {

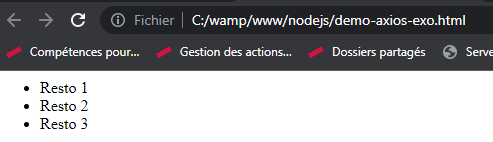
                    restaurants: ["Resto 1", "Resto 2", "Resto 3"]

                },

            })

        </script>

Résultat :



* Dans l’objet passé à Vue(), ajouter un paramètre **methods** (après **data**)

 methods: {

 }

* Créer une méthode update() qui fait appel à la bibliothèque **axios**
* Utiliser la méthode **get** de **axios**, en lui passant en paramètre l’url du serveur node-express-app

 methods: {

      update() {

        axios.get("http://127.0.0.1:5000/restaurants")

      }

Ajouter la promesse sur la réponse du serveur et l’afficher dans la console

  .then(*res* => *console.log*(*res*))

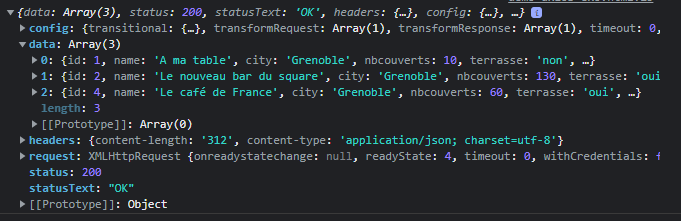
Cette méthode est appelée au clic sur un bouton de mise à jour placé en haut de page html :

<button *v-on:click*="update()">Mettre à jour</button>

La directive v-on permet d’appliquer un écouteur sur la balise. Lorsque l’évènement est détecté, la fonction callback entre guillemets est appelée.

A noter que v-on:click peut-être par la syntaxe plus courte @:click

* Démarrer le serveur node-express-app. Vérifier la réponse du serveur dans la console du navigateur.



Seule la propriété data est intéressante.

* Modifier le code en conséquence :

          .then(*res* => *this*.posts = *res*.data)

* Gérer les erreurs de connexion en ajoutant :

          .catch(*error* => *this*.posts = [{"name":"Erreur de chargement"}])

* Arrêter le serveur back. Rafraichir le navigateur. Vérifier que le message d’erreur s’affiche bien.
* Modifier le code html pour afficher tous les restaurants et leurs attributs, ainsi que les listes du personnel pour chaque restaurant
* Ajouter des boutons de suppression pour chaque restaurant et membre de l’équipe qui fera effectivement la suppression en base de données.

Exemple :

|  |  |  |
| --- | --- | --- |
| Restau 1 | Chambéry – 50 couverts – pas de terrasse – pas de parking | X |
| Equipe | Paul Harris | x |
| Yves Avitte | x |
| Jean Bonnot | x |

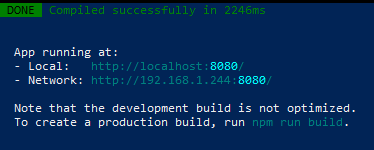
|  |  |  |
| --- | --- | --- |
| Restau 2 | Grenoble – 25 couverts – terrasse – pas de parking | X |
| Equipe | John Attan | x |
| Sylvie Bouge | x |
| Isabelle Mouton | x |

1. A partir d’une application front

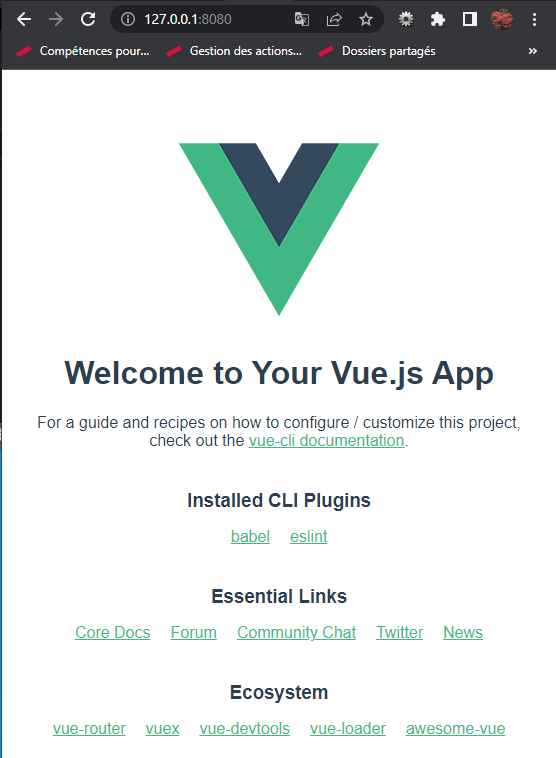
* Installer **Vue CLI** depuis une console : npm install @vue/cli

**Vue CLI** dispose de fonctionnalités natives, notamment la création automatisée d’une application front :

* Depuis une console, sous le répertoire **nodejs**, entrer : vue create front-vuejs
* Choisir le preset default (vue 2)
* Pour démarrer l’application front, depuis une console et dans le répertoire de l’application, taper : npm run serve



Depuis un navigateur, entrer l’url http://127.0.0.1:8080. On obtient cette page :



* Depuis une console, sous le répertoire **front-vuejs**, taper : npm install axios

Modifier le fichier app.vue pour ne garder que les lignes :

<template>

  <div *id*="app">

    <HelloWorld />

  </div>

</template>

<script>

import HelloWorld from './components/HelloWorld.vue'

export default {

  name: 'App',

  components: {

    HelloWorld

  }

}

</script>

<style>

</style>

* Renommer le composant **HelloWorld.vue** par **Accueil.vue**
* Modifier les fichiers **App.vue** et **Accueil.vue** en conséquence.
* Modifier **Accueil.vue** de façon à afficher la liste des restaurants et les listes des employés.

<template>

  <div *class*="hello">

    <h1>Restaurants</h1>

      <div :*key*="index" v-for="(restaurant, index) in restaurants">

// Coder ici l’affichage des restaurants

      </div>

  </div>

</template>

<script>

// Charger la bibliothèque axios

import axios from "axios"

export default {

  name: 'Accueil',

  data(){

    return{

      restaurants: null

    }

  },

  mounted(){

// Ecrire ici la requête axios

  }

}

</script>

* En sommet de page, ajouter deux composants qui contiennent chacun un formulaire qui permettent d’ajouter un restaurant ou un employé (l’affectation à un restaurant se fera par un select / option). Employer la directive v-model.
* Interfacer avec la base de données.
* Coder la validation des formulaires :
* Les noms / prénoms doivent avoir au moins 3 caractères parmi **a-z A-Z –** et **‘**
* Le nombre de couverts doit être un nombre.

En cas de non validation d’un champ d’un formulaire, un <span> indique à l’utilisateur l’erreur. De même le bouton d’envoi sera disabled tant que le formulaire n’est pas valide.

* Effectuer les mêmes tests côté back-end avant l’insertion en base.