TP 2: Utilisation de protocoles Web

Objectifs

- Faire des requêtes HTTP (HyperText Transfer Protocol)
- Création d'un serveur HTTP
- Aprendre à gérer les requêtes POST et les téléchargements de fichiers

Node.js a été conçu en pensant aux serveurs Web. En utilisant Node.js, nous pouvons créer rapidement un serveur Web avec quelques lignes de code, nous permettant de personnaliser le comportement de notre serveur. Il est important de comprendre comment Node.js interagit avec les protocoles Web sous-jacents, car ces protocoles Web constituent la base de la plupart des applications Web du monde réel.

Note importante

Dans ce TP, vous utiliserez l'API NodeJS ***http*** pour travailler avec les requêtes HTTP. Cependant, lors de la création de grandes applications complexes, il est courant de les implémenter à l'aide d'un framework de niveau supérieur plutôt que d'interagir avec les API Node.js de base. Plus tard, nous passerons à des frameworks comme ***Express*** pour travailler avec HTTP. Cependant, il est important de comprendre les API sous-jacentes et, dans certains cas, seule l'interaction avec les API Node.js de base vous fournira le contrôle précis requis dans certaines circonstances.

Utiliser le module *http* pour faire des requêtes HTTP

Les programmes et les applications ont souvent besoin d'obtenir des données d'une autre source ou d'un autre serveur. Dans le développement Web moderne, cela est généralement réalisé en envoyant une requête HTTP GET à la source ou au serveur. De même, une application ou un programme peut également avoir besoin d'envoyer des données à d'autres sources ou serveurs. Ceci est généralement réalisé en envoyant une requête HTTP POST contenant les données à la source ou au serveur cible.

En plus d'être utilisés pour créer des serveurs HTTP, les modules **http** et **https** de base de Node.js exposent des API qui peuvent être utilisées pour envoyer des requêtes à d'autres serveurs.

Créez un nouveau répertoire et ajoutez-y le fichier **request.js**:

```
$ mkdir tp2
```

```
$ cd tp2
```

```
$ touch requests.js
```

Copiez et collez le code suivant dans votre fichier javascript:

```
const http = require("http");
// Envoyer une requête HTTP GET. Nous allons envoyer une demande à http://example.com
http.get("http://example.com", (res) => {
      process.stdout.write("*****HTTP GET*****\n");
      res.pipe(process.stdout);
});
// Pour notre requête HTTP POST, nous devrons d'abord définir les données que nous souhaitons
// envoyer avec la requête. Pour y parvenir, nous définissons une variable nommée payload
// contenant une représentation JavaScript Object Notation (JSON) de nos données:
const payload = `{
      "nom": "Gates",
      "prenom": "Bill"
}`;
// Nous devons également créer un objet de configuration pour les options que nous voulons
envoyer avec
// la requête HTTP POST. Nous allons envoyer la requête HTTP POST à http://postman-echo.com.
// Il s'agit d'un point de terminaison de test qui renverra nos en-têtes HTTP, nos paramètres et
// le contenu de notre requête HTTP POST, en reflétant notre requête:
const opts = {
      method: "POST",
      hostname: "postman-echo.com",
      path: "/post",
      headers: {
            "Content-Type": "application/json",
            "Content-Length": Buffer.byteLength(payload),
      },
};
// Pour envoyer la requête HTTP POST, ajoutez le code suivant. Cela écrira les réponses du code
d'état
// HTTP (status code) et du corps (body) dans STDOUT une fois la réponse reçue:
const req = http.request(opts, (res) => {
      process.stdout.write("\n \n *****HTTP POST***** \n");
      process.stdout.write("Status Code: " + res.statusCode +   "\n");
      process.stdout.write("Body: ");
      res.pipe(process.stdout);
});
// Nous devrions également détecter toutes les erreurs qui se produisent sur la demande:
req.on("error", (err) => console.error("Error: ", err));
// Enfin, nous devons envoyer notre requête avec le payload:
req.end(payload);
```

Pour la requête HTTP GET, nous appelons la fonction **http.get()** avec deux paramètres. Le premier paramètre est le point de terminaison auquel nous souhaitons envoyer la demande, et le second est la fonction de rappel (Callback). La fonction de rappel s'exécute une fois la requête HTTP GET terminée, et dans cet exemple, notre fonction transmet la réponse que nous recevons du point de terminaison à STDOUT.

Note: Postman (http://postman.com) est une plate-forme de développement d'API et fournit une application cliente REST que vous pouvez télécharger pour envoyer des requêtes HTTP. Postman fournit également un service nommé Postman Echo, qui fournit un point de terminaison auquel vous pouvez envoyer vos requêtes HTTP à des fins de test. Reportez-vous à la documentation Postman Echo à l'adresse : https://docs.postman-echo.com/?version=latest.

Exécutez votre programme et vous devriez voir que l'API Postman Echo répond à votre requête HTTP POST

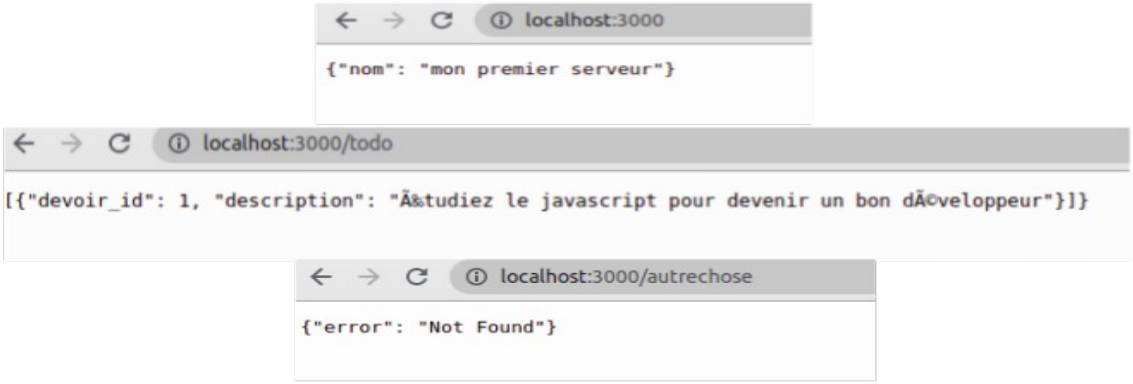
Construire un serveur HTTP pour accepter les requêtes GET

HTTP signifie HyperText Transfer Protocol et est un protocole de couche d'application qui sous-tend le World Wide Web (WWW). HTTP permet la communication entre les serveurs et les navigateurs. Dans cet exemple, nous utiliserons les API de base Node.js pour créer un serveur HTTP qui acceptera uniquement les requêtes GET.

Créez un fichier nommé ***server.js*** qui contiendra notre serveur HTTP. Copiez le code suivant et collez-le dans votre fichier nouvellement créé:

```
const http = require("http");
// définissez le nom d'hôte et le port de votre serveur:
const HOSTNAME = process.env.HOSTNAME || "0.0.0.0";
const PORT = process.env.PORT || 3000;
// Créer le serveur et ajouter une gestion de route. Dans la fonction createServer(),
// nous allons référencer les fonctions error(), todo() et index() que nous allons créer
//  dans les étapes suivantes:
const server = http.createServer((req, res) => {
      if (req.method !== "GET") return error(res, 405);
      if (req.url === "/todo") return devoir(res);
      if (req.url === "/") return index(res);
      error(res, 404);
});
// Créer notre fonction error(). Cette fonction prendra un paramètre de l'objet de réponse
// et un code d'état, où le code est censé être un code d'état HTTP (http status code):
function error(res, code) {
      res.statusCode = code;
      res.end(`{"error": "${http.STATUS_CODES[code]}"}`);
}
// Cette fonction renverra simplement une chaîne JSON statique représentant un élément de la
liste «faire»:
function devoir(res) {
      res.end('[{"devoir_id": 1, "description": "Étudiez le javascript pour devenir un bon
développeur"}]}');
}
// Il s'agit de la fonction index(), qui sera appelée lorsque nous effectuerons une requête GET sur
la route /
// route / ==> localhost:3000/
function index(res) {
      res.end('{"nom": "mon premier serveur"}');
}
// Enfin, nous devons appeler la fonction listen() sur notre serveur. Nous passerons également
une fonction
// de rappel à la fonction listen() qui affichera l'adresse sur laquelle le serveur écoute, une fois
// le serveur démarré:
server.listen(PORT, HOSTNAME, () => {
      console.log(`Le serveur écoute sur le port ${server.  address().port}`);
});
```

Exécutez le fichier server.js et testez-le à l'aide de votre navigateur Web.



Pour la fonction d'erreur, nous passons un paramètre supplémentaire, ***code***. Nous l'utilisons pour transmettre puis renvoyer les codes d'état HTTP. Les codes d'état HTTP font partie de la spécification du protocole HTTP (https://tools.ietf.org/html/rfc2616#section-10). Le tableau suivant montre comment les codes de réponse HTTP sont regroupés:

Range	Use
1xx	Information
2xx	Success
3xx	Redirection
4xx	Client errors
5xx	Server errors

Le module ***http*** expose un objet constant qui stocke tous les codes de réponse HTTP et leurs descriptions correspondantes: ***http.STATUS_CODES***. Nous l'avons utilisé pour renvoyer le message de réponse avec ***http.STATUS_CODE***.

Travailler avec le format de module ES6/ES2015

Les modules ES6 sont un nouveau format de module conçu pour tous les environnements JavaScript. Alors que Node.js a toujours eu un bon système de modules, JavaScript côté navigateur n'en a pas. Cela signifiait que la communauté côté navigateur devait utiliser des solutions non standardisées. Par conséquent, les modules ES6 sont une grande amélioration pour l'ensemble du monde JavaScript, en mettant tout le monde sur la même page avec un format et des mécanismes de module communs.

Un problème auquel nous devons faire face est l'extension de fichier à utiliser pour les modules ES6. Node.js doit savoir s'il faut analyser à l'aide de la syntaxe du module ES6. Pour les distinguer, Node.js utilise l'extension de fichier **.mjs** pour désigner les modules *ES6* et **.js** pour désigner les modules *CommonJS*.

Créez un nouveau fichier nommé ***server.mjs*** et ajoutez le code suivant:

```
import * as http from 'http';
import * as util from 'util';
import * as os from 'os';
```

Cette version de l'instruction **import** est très similaire à une instruction **require** Node.js traditionnelle car elle crée un objet avec des champs contenant les objets exportés depuis le module.

Examinons maintenant une autre application serveur qui effectue différentes actions en fonction de l'URL.

Ajoutez le code suivant à votre fichier **server.mjs**:

```
const listenOn = 'http://localhost:3000';
const server = http.createServer();
server.on('request', (req, res) => {
	var requrl = new URL(req.url, listenOn);
	if (requrl.pathname === '/') homePage(req, res);
	else if (requrl.pathname === "/osinfo") osInfo(req, res);
	else { res.writeHead(404, {'Content-Type': 'text/plain'}); res.end("bad URL "+ req.url); }
});
server.listen(new URL(listenOn).port);
console.log(`Le serveur écoute: ${listenOn}`);
function homePage(req, res) {
	res.writeHead(200, {'Content-Type': 'text/html'});
	res.end(
		`<html><head><title>Bonjour!</title></head>
		<body><h1>Bonjour, c'est la page d'accueil!</h1>
		<p><a href='/osinfo'>OS Info</a></p>
		</body></html>`);
}
function osInfo(req, res) {
	res.writeHead(200, {'Content-Type': 'text/html'});
	res.end(
		`<html><head><title>Operating System Info</title></head>
		<body><h1>Informations sur le système d'exploitation</h1>
		<table>
		<tr><th>TMP Dir</th><td>${os.tmpdir()}</td></tr>
		<tr><th>Host Name</th><td>${os.hostname()}</td></tr>
		<tr><th>OS Type</th><td>${os.type()} ${os.platform()}
		${os.arch()} ${os.release()}</td></tr>
		<tr><th>Uptime</th><td>${os.uptime()} ${util.inspect(os.loadavg())}</td></tr>
		<tr><th>Memory</th><td>total: ${os.totalmem()} free: $
{os.freemem()}</td></tr>
		<tr><th>CPU's</th><td><pre>${util.inspect(os.cpus())}</pre></td></tr>
		<tr><th>Network</th><td><pre>${util.inspect(os.networkInterfaces())}</pre></
td></tr>
		</table>
		</body></html>`);
}
```

L'événement de **request** est émis par **HTTPServer** chaque fois qu'une demande arrive d'un navigateur Web. Dans ce cas, nous souhaitons répondre différemment en fonction de l'URL de la demande, qui arrive en tant que **req.url.** Cette valeur est une chaîne contenant l'URL de la requête HTTP. Comme il existe de nombreux attributs pour une URL, nous devons analyser l'URL afin de pouvoir faire correspondre correctement le nom de chemin pour l'un des deux chemins: / et **/osinfo**.

L'analyse d'une URL avec la classe URL nécessite une URL de base, que nous avons fournie dans la variable **listenOn**. Remarquez comment nous réutilisons cette même variable dans quelques autres endroits, en utilisant une chaîne pour configurer plusieurs parties de l'application.

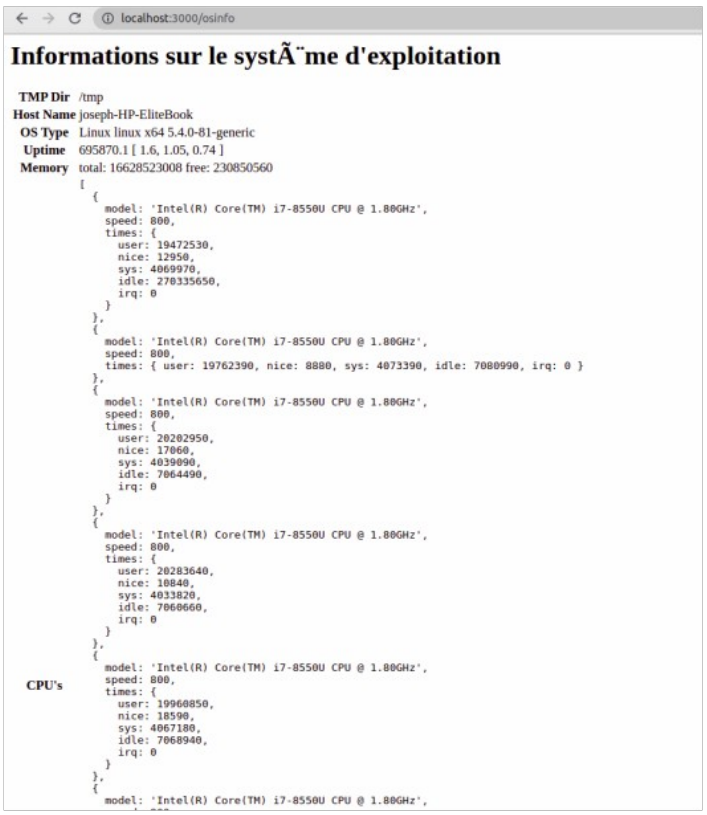
Selon le chemin, les fonctions **homePage** ou **osInfo** sont appelées.

C'est ce qu'on appelle le routage des demandes, où nous examinons les attributs de la demande entrante, tels que le chemin de la demande, et acheminons la demande vers les fonctions du gestionnaire.

Dans les fonctions de gestion, les paramètres **req** et **res** correspondent aux objets **request** et **response**. Lorsque **req** contient des données sur la demande entrante, nous envoyons la réponse en utilisant **res**. La fonction **writeHead** définit l'état de retour (**200** signifie succès, tandis que **404** signifie que la page n'est pas trouvée) et la fonction **end** envoie la réponse.

Si l'URL de la requête n'est pas reconnue, le serveur renvoie une page d'erreur en utilisant un code de résultat **404**. Le code de résultat informe le navigateur de l'état de la demande, où un code **200** signifie que tout va bien et un code **404** signifie que la page demandée n'existe pas. Il existe bien sûr de nombreux autres codes de réponse HTTP, chacun ayant sa propre signification.

Exécutez le script en utilisant **$ node server.mjs** et testez-le avec votre navigateur.



Gestion des requêtes HTTP POST

La méthode HTTP POST est utilisée pour envoyer des données au serveur, par opposition à la méthode HTTP GET, qui est utilisée pour obtenir des données.

Pour pouvoir recevoir des données POST, nous devons indiquer à notre serveur comment accepter et gérer les requêtes POST. Une requête POST contient généralement des données dans le corps de la requête, qui sont envoyées au serveur pour être traitées. La soumission d'un formulaire Web se fait généralement via une requête HTTP POST.

Note importante:

En PHP, il est possible d'accéder aux données POST via le tableau **$_POST**. PHP ne suit pas l'architecture non bloquante de Node.js, ce qui signifie que le programme PHP attendrait, ou bloquerait, jusqu'à ce que les valeurs **$_POST** soient renseignées. Node.js, cependant, fournit une interaction asynchrone avec les données HTTP à un niveau inférieur, ce qui nous permet d'interagir avec le corps du message entrant en tant que flux. Cela signifie que la gestion du flux entrant est sous le contrôle et la préoccupation du développeur.

Dans et exemple, nous allons créer un serveur Web qui accepte et gère les requêtes HTTP POST.

Créez un fichier nommé **post_server.js** qui contiendra notre serveur HTTP. Vous devez également créer un sous-répertoire appelé ***public***, contenant un fichier nommé **form.html** qui contiendra un formulaire HTML:

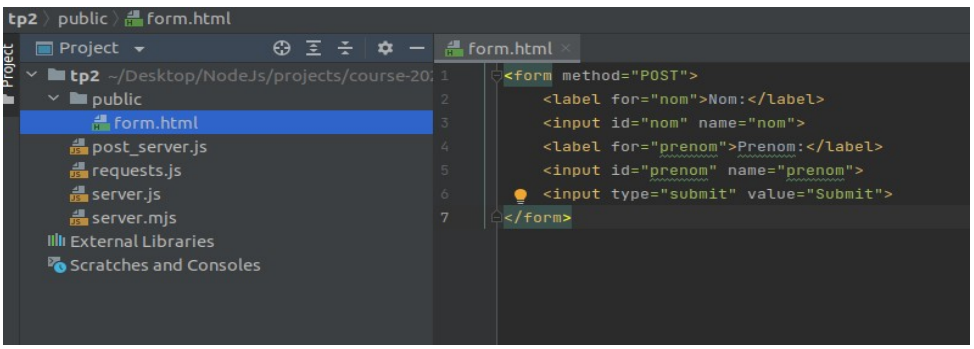
```
$ touch post_server.js
```

```
$ mkdir public
```

```
$ touch public/form.html
```

Tout d'abord, configurons un formulaire HTML avec des champs de saisie pour nom et prenom. Ouvrez **form.html** et ajoutez les éléments suivants:

```
<form method="POST">
      <label for="nom">Nom:</label>
      <input id="nom" name="nom">
      <label for="prenom">Prenom:</label>
      <input id="prenom" name="prenom">
      <input type="submit" value="Submit">
</form>
```



Copiez et collez le code suivant dans **post_server.js**:

```
const http = require("http");
const fs = require("fs");
const path = require("path");
// Créer une référence au fichier form.html
const form = fs.readFileSync(path.join(__dirname, "public", "form.html"));
// Configurez le serveur et créez une fonction get() pour renvoyer le formulaire,
// une fonction post() et une fonction d'erreur nommée error()
http
	.createServer((req, res) => {
		//Si vous recevez une requête http avec la méthode GET
		if (req.method === "GET") {
			get(res);
			return;
		}
		//Si vous recevez une requête http avec la méthode POST
		if (req.method === "POST") {
			post(req, res);
			return;
		}
		error(405, res);
	})
	.listen(3000);
// Cette fonction affichera la page form.html
function get(res) {
	res.writeHead(200, {
		"Content-Type": "text/html",
	});
	res.end(form);
}
// Cette fonction répondra avec le message OK (status code 200)
function post(req, res) {
	if (req.headers["content-type"] !== "application/x-www-form-urlencoded") {
		error(415, res);
		return;
	}
	let input = "";
	req.on("data", (chunk) => {
		input += chunk.toString();
	});
	req.on("end", () => {
		console.log(input);
		res.end(http.STATUS_CODES[200]);
	});
}
function error(code, res) {
	res.statusCode = code;
	res.end(http.STATUS_CODES[code]);
}
```

Redémarrez votre serveur et revenez à http://localhost:3000 dans votre navigateur et soumettez le formulaire. Vous devriez voir un message OK retourné. Si vous regardez la fenêtre Terminal où vous exécutez votre serveur, vous pouvez voir que le serveur a reçu vos données.

$ node post_server.js

Gestion des requêtes HTTP POST qui envoient des données JSON

Il est courant que les API modernes prennent en charge les interactions avec les données JSON. Plus précisément, cela signifie accepter et gérer le contenu avec le type de contenu ***application/json***. Convertissons le serveur de cet exemple pour gérer les données JSON.

Tout d'abord, copiez le fichier **post_server.js** existant dans un nouveau fichier nommé **json-server.js**:

$ cp post_server.js json-server.js

Ensuite, nous allons modifier notre fonction **post()** pour vérifier que le Content-Type de la requête est défini sur **application/json.** Nous devons également modifier notre fonction **end()** pour analyser et renvoyer les données JSON:

```
function post(req, res) {
	if (req.headers["content-type"] !== "application/json") {
		error(415, res);
		return;
	}
	let input = "";
	req.on("data", (chunk) => {
		input += chunk.toString();
	});
	req.on("end", () => {
		const parsed = JSON.parse(input);
		if (parsed.err) {
			error(400, "Bad Request", res);
			return;
		}
		console.log("Received data: ", parsed);
		res.end('{"data": ' + input + "}");
	});
}
```

Testons maintenant si notre serveur peut gérer la route POST. Nous allons le faire à l'aide de l'outil de ligne de commande ***curl***. Démarrez votre serveur dans une fenêtre de terminal:

$ node json-server.js

Et, dans une autre fenêtre Terminal, saisissez la commande suivante:

$ curl --header "Content-Type: application/json" --request POST --data '{"nom":"Gates","prenom":"Bill"}' http://localhost:3000

```
(base) joseph@joseph-HP-EliteBook:~/Desktop/NodeJs/projects/course-2021/session2/tp2$ curl --header "Content-Type: application/json" --request POST --data '{"nom"
:"Gates","prenom":"Bill"}' http://localhost:3000
{"data": {"nom":"Gates","prenom":"Bill"}}
```

Maintenant, nous pouvons ajouter le script suivant à notre fichier **form.html**, qui convertira nos données de formulaire HTML en JSON et les enverra via une requête POST au serveur. Ajoutez ce qui suit après la balise de fermeture de formulaire (</form>):

```
<script>
    document.forms[0].addEventListener("submit", (event) => {
        event.preventDefault();
        let data = {
            nom: document.getElementById("nom").value,
            prenom: document.getElementById("prenom").value,
        };
        console.log("data", data);
        fetch("http://localhost:3000", {
            method: "post",
            headers: {
                "Content-Type": "application/json",
            },
            body: JSON.stringify(data),
        }).then(response =>
            response.json().then(data => ({
                    data: data,
                    status: response.status
                })
            ).then(res => {
                alert(`Response status: ${res.status}, response data
                 ${res.data.data.nom} ${res.data.data.prenom}`);
            }));
    });
</script>
```

Redémarrez votre serveur JSON avec **$ node json-server.js** et accédez à **http://localhost:3000** dans votre navigateur. Si nous remplissons maintenant les champs de saisie dans notre navigateur et soumettons le formulaire, nous devrions voir un message que la demande a été envoyée avec succès au serveur. Notez que notre utilisation de **event.preventDefault()** empêchera le navigateur de rediriger la page Web lors de la soumission du formulaire.

Notre formulaire et notre serveur se comportent de la même manière que le serveur que nous avons créé dans l'exemple de gestion des requêtes HTTP POST, à la différence que le formulaire *frontend* interagit avec le *backend* via une requête HTTP POST qui envoie une représentation JSON des données du formulaire. L'interface client interagissant avec le serveur principal via JSON est typique des architectures Web modernes.

Utilisation de “*formidable*” pour gérer les téléchargements de fichiers

Le téléchargement d'un fichier sur le Web est une activité courante, qu'il s'agisse d'une image, d'une vidéo ou d'un document. Les fichiers nécessitent un traitement différent par rapport aux données POST simples. Les navigateurs intègrent les fichiers téléchargés dans des messages en plusieurs parties.

Les messages en plusieurs parties permettent de combiner plusieurs éléments de contenu en une seule charge utile. Pour gérer les messages en plusieurs parties, nous devons utiliser un analyseur en plusieurs parties.

Dans cet exemple, nous utiliserons le module “formidable” comme analyseur syntaxique en plusieurs parties pour gérer les téléchargements de fichiers.

créez un fichier **file-server.js**:

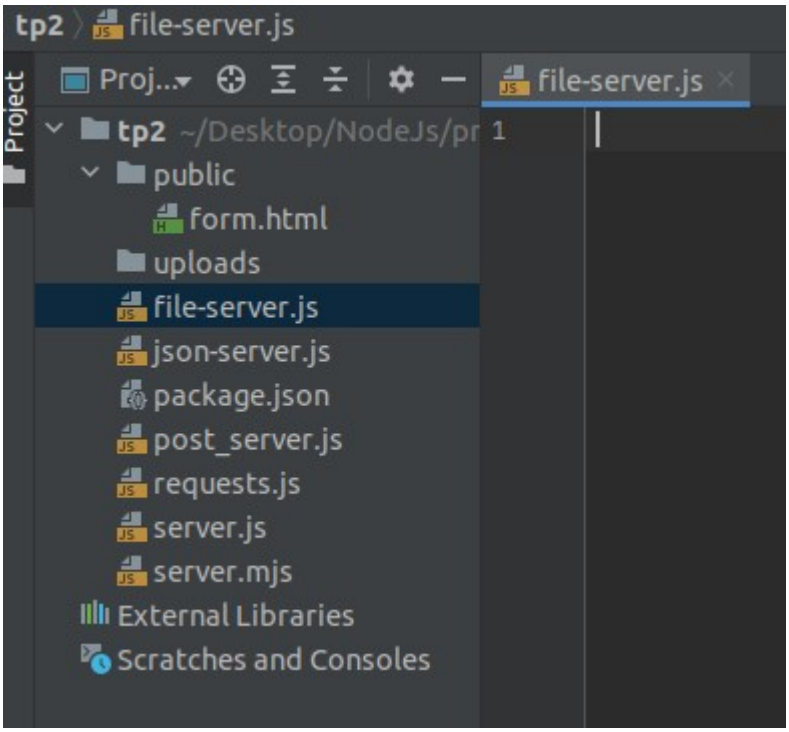
$ touch file-server.js

Nous utiliserons un module npm pour cet exemple, nous devons initialiser notre projet:

$ npm init –yes

Nous devons créer un autre sous-répertoire nommé uploads pour stocker nos fichiers téléchargés:

$ mkdir uploads



Créez un fichier nommé ***fileupload.html*** dans le répertoire ***public***. Ajoutez le contenu suivant à ***fileupload.html***:

```
<!DOCTYPE html>
<html lang="en">
<head>
      <meta charset="UTF-8">
      <title>File Upload</title>
</head>
<body>
<form method="POST" enctype="multipart/form-data">
   <label for="userfile">File:</label>
   <input type="file" id="userfile" name="userfile"><br>
   <input type="submit">
</form>
</body>
</html>
```

Maintenant, nous devons installer notre module d'analyseur multi-parties, ***formidable***:

$ npm install formidable

Maintenant, nous pouvons commencer à créer notre serveur. Dans **file-server.js**, nous importerons les modules requis et créerons une variable pour stocker le chemin d'accès à notre fichier **fileupload.html**:

```
const fs = require("fs");
const http = require("http");
const path = require("path");
const form = fs.readFileSync(path.join(__dirname, "public", "fileupload.html"));
const formidable = require("formidable");
```

Ensuite, nous allons créer notre serveur avec des gestionnaires pour les requêtes GET et POST. Ceci est similaire au serveur que nous avons construit dans l'exemple de gestion des requêtes HTTP POST:

```
http
	.createServer((req, res) => {
		if (req.method === "GET") {
			get(res);
			return;
		}
		if (req.method === "POST") {
			post(req, res);
			return;
		}
		error(405, res);
	})
	.listen(3000);
function get(res) {
	res.writeHead(200, {
		"Content-Type": "text/html",
	});
	res.end(form);
}
function error(code, res) {
	res.statusCode = code;
	res.end(http.STATUS_CODES[code]);
}
```

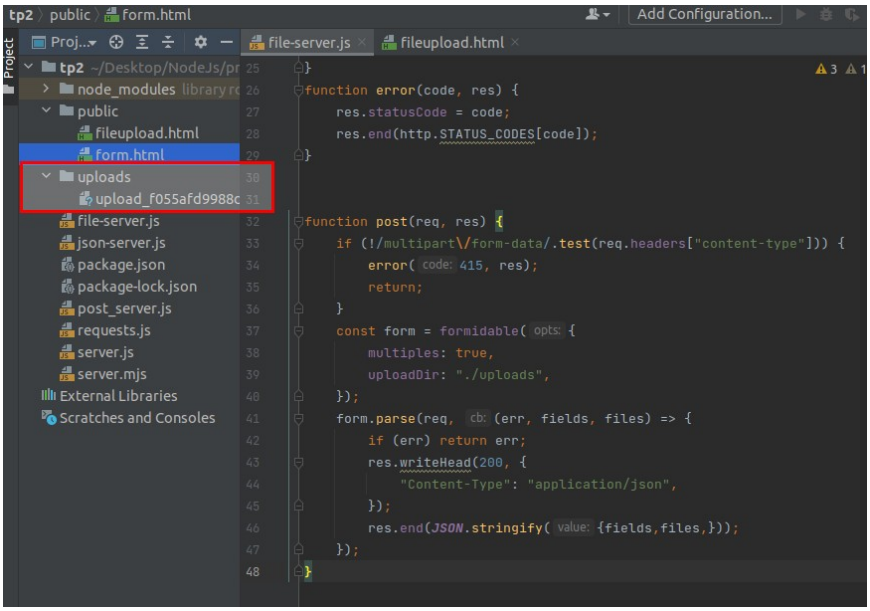
Maintenant, nous allons ajouter notre fonction POST. Cette fonction gérera le téléchargement du fichier:

```
function post(req, res) {
	if (!/multipart\/form-data/.test(req.headers["content-type"])) {
		error(415, res);
		return;
	}
	const form = formidable({
		multiples: true,
		uploadDir: "./uploads",
	});
	form.parse(req, (err, fields, files) => {
		if (err) return err;
		res.writeHead(200, {
			"Content-Type": "application/json",
		});
		res.end(JSON.stringify({fields,files,}));
	});
}
```

Démarrez le serveur et accédez à **http://localhost:3000** dans votre navigateur:

```
$ node file-server.js
```

Cliquez sur “**Choisir un fichier/Choose File**” et sélectionnez n'importe quel fichier à télécharger dans votre explorateur de fichiers. Vous devriez voir votre nom de fichier apparaître à côté du bouton Choisir un fichier. Cliquez sur “**Soumettre/Submit**”. Votre serveur doit avoir correctement reçu et stocké le fichier et répondu avec des données sur le fichier stocké au format JSON:



À faire

Vous allez construire un site web statique pour les geeks de l'IUT BM. Les Geeks de l'IUT BM sont intéressés à proposer des cours gratuits à toute personne intéressée par l'apprentissage du développement web. Essayez de créer un site Web sympa pour eux.

Pour créer cette application, procédez comme suit:

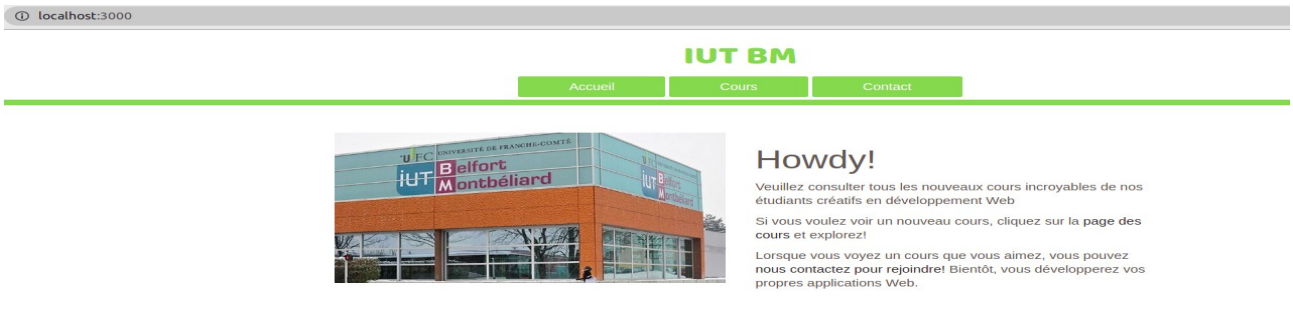
- Téléchargez et extraire le dossier **tp2_ex1_start.zip** depuis *https://cours-info.iut-bm.univ-fcomte.fr/index.php/menu-cours-s3/menu-mmi3ihm/2229-s3-programmation-web-cote - serveur-2021*
- Initialiser le projet node js et installez les modules nécessaires avec npm
- Créez la logique d'application dans main.js
- Utilisez le fichier contentTypes.js pour définir les informations "Content-Type"
- Utilisez le fichier utils.js pour définir une fonction qui ouvre un fichier et renvoie son contenu. Ce fichier JS peut être utilisé par d'autres fichiers JS
- Codez le routeur de l'application et ajoutez les routes pour les vues et les assets
- Gérer les erreurs d'application
- Exécutez l'application
- Créer cette web app à l'aide du module **http**, ne pas utiliser de framework

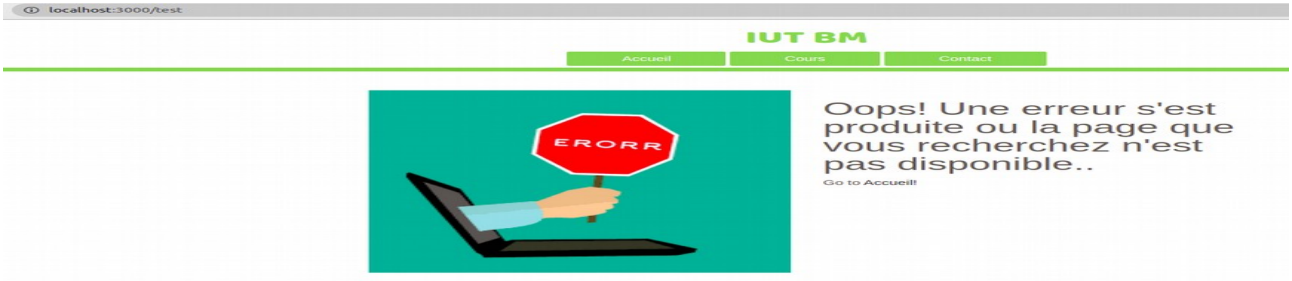
Le dossier zip téléchargé contient les répertoires “public” (assets) et “views” (pages HTML). Ne perdez pas votre temps sur le front-end. Dans le dossier téléchargé, vous avez quatre fichiers javascript vides sur lesquels vous devez travailler: **contentTypes.js**, **utils.js, main.js** et **router.js**

Le répertoire complet du projet d'application ressemblera à la structure de la liste suivante:

```
├── contentTypes.js
├── main.js
├── node_modules
├── package.json
├── package-lock.json
├── public
│   ├── css
│   │   ├── bootstrap.css
│   │   └── iutUFC.css
│   ├── images
```

```
│   │   ├── ALDD17.jpg
│   │   ├── contact.jpg
│   │   ├── error.png
│   │   └── product.jpg
│   └── js
│       └── iutUFC.js
├── router.js
├── utils.js
└── views
    ├── contact.html
    ├── courses.html
    ├── error.html
    └── index.html
```





À faire

Créez un fichier html ***etudiants.html*** dans le répertoire ***"public/"***.

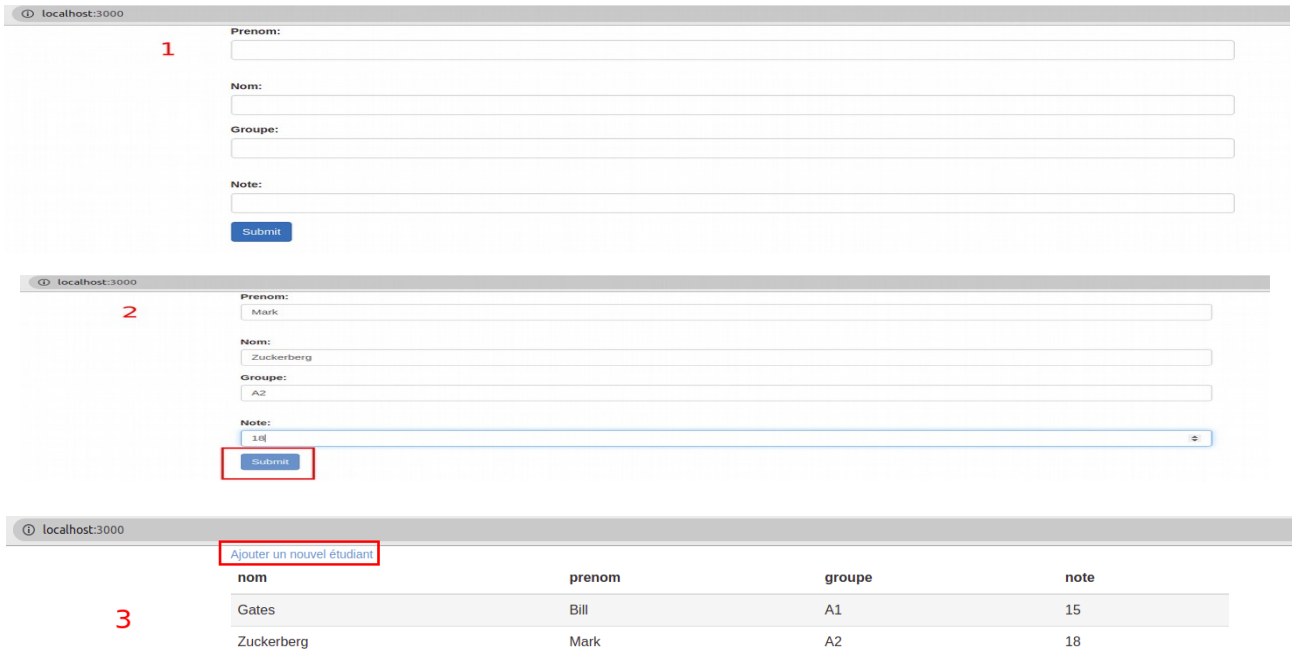
Copiez le code suivant et collez-le dans ***etudiants.html***:

```
<!DOCTYPE html>
<html lang="en">
<head>
      <meta charset="UTF-8">
      <title>Etudiants</title>
      <meta name="viewport" content="width=device-width, initial-scale=1">
      <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/css/bootstrap.min.css">
      <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
      <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.4.1/js/bootstrap.min.js"></
script>
</head>
<body>
<div class="container">
      <form action="/" method="POST">
            <div class="form-group">
                  <label for="prenom">Prenom:</label><br>
                  <input class="form-control" type="text" id="prenom" name="prenom"><br>
            </div>
            <div class="form-group">
                  <label for="nom">Nom:</label><br>
                  <input class="form-control" type="text" id="nom" name="nom">
            </div>
            <div class="form-group">
                  <label for="groupe">Groupe:</label><br>
                  <input class="form-control" type="text" id="groupe" name="groupe"><br>
            </div>
            <div class="form-group">
                  <label for="note">Note:</label><br>
                  <input class="form-control" type="number" id="note" name="note">
            </div>
            <input class="btn btn-primary" type="submit">
      </form>
</div>
</body>
</html>
```

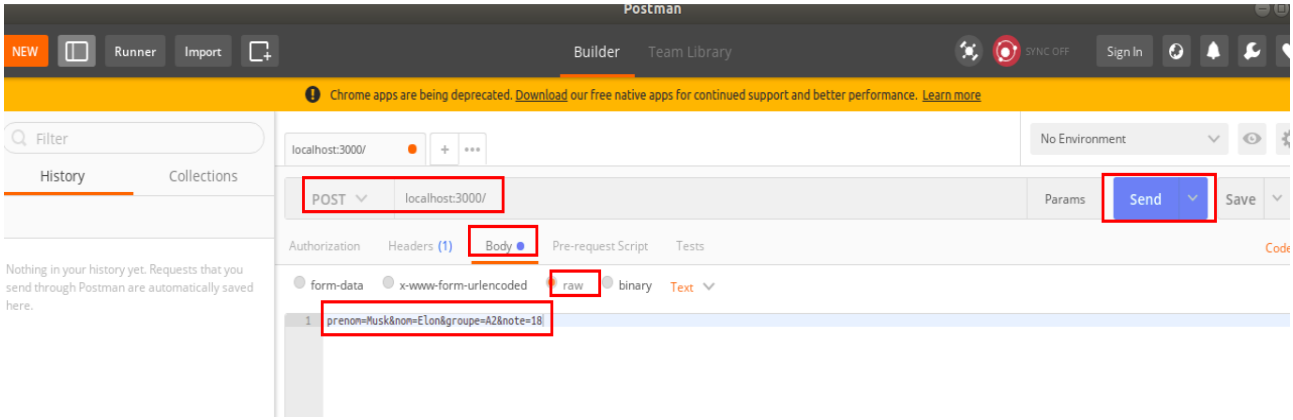
Créez un serveur http qui effectue les opérations suivantes:

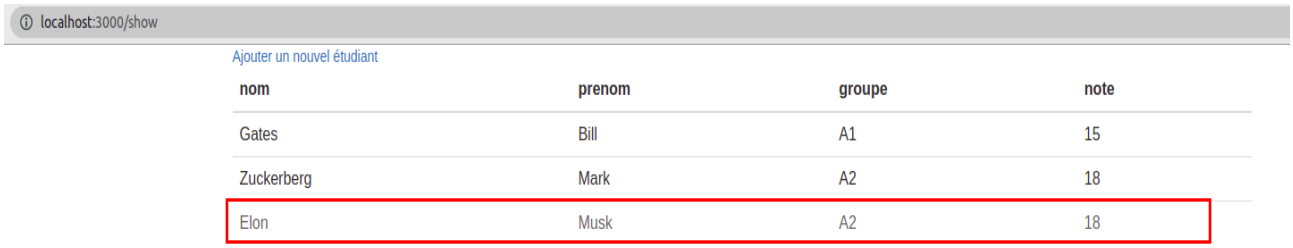
- Si l'URL est "/" et que la méthode est GET, le serveur renvoie l'utilisateur vers la page ***etudiants.html***
- Si l'URL est différente de "/", le serveur affiche un message d'erreur

- Le fichier html "**etudiants.html**" contient un formulaire dans lequel l'utilisateur saisira les informations des étudiants. Nous voulons stocker ces informations dans un fichier json. Assurez-vous que votre script nodeJS vérifie si le fichier json existe. S'il n'existe pas, il le crée avec un tableau json vide.
- Lorsque l'utilisateur entre les informations et appuie sur le bouton “Submit”, le serveur doit traiter la demande HTTP Post en ajoutant l’étudiant créé au fichier JSON, et en réponse, le serveur doit montrer à l'utilisateur un tableau HTML avec tous les étudiants stockés dans le fichier JSON. Assurez-vous de montrer à l'utilisateur un lien lui permettant d'entrer un autre étudiant.



Vous pouvez envoyer une requête HTTP POST en utilisant **Postman** à votre serveur. Créez une route **"/show"** qui montre les étudiants dans un tableau HTML. Essayez de “poster” un nouvel étudiant à l'aide de **Postman** et vérifiez s'il apparaîtrait dans le tableau lors de l'accès à la route **/show**.





Parsing avec “querystring”

Le module “querystring” peut vous aider dans cet exercice.

```
const querystring = require('querystring');
const url = "http://example.com/index.html?code=string&key=12&id=false";
const qs = "code=string&key=12&id=false";

console.log(querystring.parse(qs));
// > { code: 'string', key: '12', id: 'false' }

console.log(querystring.parse(url));
// > { 'http://example.com/index.html?code': 'string', key: '12', id: 'false' }
```