

Développement Web

Julien Provillard

NODE ET PROGRAMMATION SERVEUR

Node

- ❑ On suppose ici que vous avez pleinement configuré votre environnement pour supporter Node et son écosystème :
 - nvm ou tout autre gestionnaire de version (optionnel),
 - node lui-même,
 - npm pour gérer les modules extérieurs.

Configuration d'un projet

- ☐ Créez un répertoire pour stocker votre projet.
- ☐ Toutes les commandes seront lancées depuis ce répertoire (en particulier node).
- ☐ Tous les chemins indiqués seront relatif à ce répertoire.
- ☐ Ajouter un fichier `package.json`. Il contiendra les métadonnées de votre projet.

Configuration d'un projet

□ Contenu du fichier `package.json`.

```
{  
  "name": "mon-premier-serveur",  
  "version": "0.0.1",  
  "description": "Tests de nodejs et implémentation de serveurs simples.",  
  "homepage": "somewhere.that.i.used.to.know.com",  
  "licence": "MIT",  
  "author": "Julien Provillard",  
  "type": "module",  
  "dependencies": {  
    "random-item": "^4.0.1"  
  }  
}
```

Configuration d'un projet

- ❑ La commande `npm install` permet de télécharger et d'installer les dépendances.
- ❑ Elles deviennent disponibles à l'import. Il suffit de préciser le nom du module comme origine.

```
// fichier test.js
```

```
import randomItem from "random-item";
```

```
const t = ["foo", "bar", "baz"];  
console.log(randomItem(t));
```

```
me@computer:path/to/project$ npm install  
added 1 package, and audited 2 packages in 1s  
  
1 package is looking for funding  
  run `npm fund` for details  
  
found 0 vulnerabilities  
me@computer:path/to/project$ node test.js  
bar
```

Autres possibilités pour lancer node

- ❑ Dans `package.json`, ajoutez la ligne `"main": "test.js"`.
- ❑ La commande `node .` permet alors de lancer le projet.

```
me@computer:path/to/project$ node .  
baz
```

- ❑ Dans `package.json`, ajoutez
`"scripts": { "start": "node test.js" }`
- ❑ La commande `npm start` permet alors de lancer le projet.

```
me@computer:path/to/project$ npm start  
> mon-premier-serveur@0.0.1 start  
> node test.js  
  
bar
```

Système de fichiers

- ❑ Un des principaux apports de Node est l'accès au système de fichiers.
- ❑ Celui-ci se divise en trois API :
 - une API asynchrone avec fonctions de rappels,
 - une API asynchrone avec promesses,
 - une API synchrone.
- ❑ Les exemples qui suivent supposent que l'on dispose d'un fichier `exemple.txt` dans le répertoire `./fs/` (le répertoire courant est celui depuis lequel on exécute Node).

Exemple avec fonctions de rappel

```
import { readFile, writeFile } from "fs"; // asynchrone avec callback

readFile("./fs/exemple.txt", "utf8", (error, text) => {
  if (error) {
    console.log(`Erreur en lecture : ${error.message}`);
  } else {
    console.log(`Contenu du fichier : ${text}`);
  }
});

writeFile("./fs/log-callback.txt", "Écriture par callback.", (error) => {
  if (error) {
    console.log(`Erreur en écriture : ${error.message}`);
  } else {
    console.log("Écriture réussie.");
  }
});
```

Exemple avec promesses

```
import { readFile, writeFile } from "fs/promises"; // asynchrone avec promesses

readFile("./fs/exemple.txt", "utf8")
  .then((text) => console.log(`Contenu du fichier : ${text}`))
  .catch((error) => console.log(`Erreur en lecture : ${error.message}.`));

writeFile("./fs/log-promise.txt", "Écriture par promesse.").then(
  () => console.log("Écriture réussie."),
  (error) => console.log(`Erreur en écriture : ${error.message}.`));
```

Exemple synchrone

```
import { readFileSync, writeFileSync } from "fs"; // synchrone et donc bloquant

try {
  const text = readFileSync("./fs/exemple.txt", "utf8");
  console.log(`Contenu du fichier : ${text}`);
} catch (error) {
  console.log(`Erreur en lecture : ${error.message}.`);
}

try {
  writeFileSync("./fs/log-sync.txt", "Écriture synchrone.");
  console.log("Écriture réussie.");
} catch (error) {
  console.log(`Erreur en écriture : ${error.message}.`);
}
```

SERVEUR

Un premier serveur

- ❑ Nous faisons le choix ici de n'utiliser aucun module externe.
- ❑ Le but est de vous présenter les mécanismes sous-jacent.
- ❑ Dans les faits, on se simplifie grandement la vie en utilisant des modules dédiés.

Un premier serveur

```
import { createServer } from "http";

const server = createServer((request, response) => {
  response.writeHead(200, { "Content-Type": "text/html; charset=utf-8" });
  response.write(`
    <h1>Félicitations !</h1>
    <p>Vous venez de créer votre premier serveur.</p>
    <p>Vous cherchiez à accéder à la ressource <code>${request.url}</code> en
    utilisant la méthode <code>${request.method}</code>.</p>`);
  response.end();
});

server.listen(8500, () => console.log("Server listening."));
```

Un premier serveur

Création du serveur

Requête reçue

Réponse à envoyer

```
const server = createServer((request, response) => {  
  response.writeHead(200, { "Content-Type": "text/html; charset=utf-8" });  
  response.write(`  
    <h1>Félicitations !</h1>  
    <p>Vous venez de créer votre premier serveur.</p>  
    <p>Vous cherchiez à accéder à la ressource <code>${request.url}</code> en  
    utilisant la méthode <code>${request.method}</code>.</p>` );  
  response.end();  
});  
server.listen(8500, () => console.log("Server listening."));
```

Ecriture de l'entête de la réponse avec code de retour

Ecriture du corps de la réponse

Envoie de la réponse

Lancement du serveur sur le port précisé

Un premier serveur

- ❑ Après avoir lancé ce serveur, en accédant à <http://localhost:8500/foo>, on obtient :

Félicitations !

Vous venez de créer votre premier serveur.

Vous cherchiez à accéder à la ressource /foo en utilisant la méthode GET.

- ❑ Il y a principalement deux étapes pour la création d'un serveur :
 - La création du serveur et son paramétrage via `createServer`.
 - Son lancement, le moment où il commence à réellement écouter les requêtes en provenance d'un port via la méthode `listen`.

Un premier serveur

- ❑ La fonction `createServer` permet de définir un serveur et prend en paramètre une fonction de rappel à deux paramètres.
- ❑ Ces paramètres sont usuellement appelés `request` et `response` (très souvent abrégés en `req` et `res`).
- ❑ Pour chaque connexion , cette fonction est appelée.
 - Le paramètre `request` représente la requête entrante à traiter.
 - Le paramètre `response` correspond à la réponse sortante qui est spécifiée à l'intérieur de la fonction.

LES REQUÊTES

L'objet requête

- ❑ Il représente la requête à traiter, ses propriétés les plus importantes sont :
 - `url`, l'adresse de la ressource demandée sur le serveur. Elle peut contenir des paramètres supplémentaires (surtout pour une requête GET).
 - `headers`, l'entête de la requête. Il contient des informations sur la manière de réaliser la requête.
 - `method`, la méthode (verbe HTTP) utilisée pour la requête.
 - `on`, permet d'installer des écouteurs sur les données entrantes pour les requêtes qui ont un corps.
- ❑ La liste des entêtes possibles se trouve [ici](#). Nous en verrons certains dans le cours.

Les méthodes HTTP

❑ Les principales méthodes HTTP sont listées ici :

Nom de la méthode	Signification
GET	Accès à une ressource en lecture. Pas de modification côté serveur. Pas de corps.
POST	Envoi de données typiquement pour une gestion côté serveur.
PUT	Modification/ajout d'une entité contenue dans le corps de la requête.
DELETE	Suppression d'une donnée.
OPTIONS	Demande d'accès et méthodes autorisées (utilisée par le CORS).
PATCH	Modification partielle d'une entité.

Lire le corps des requêtes

❏ Le corps des requêtes est un flux qu'il faut lire de manière asynchrone.

```
export function readStream(stream, limit = Infinity) { // lire le corps de la requête
  return new Promise((resolve, reject) => {
    const data = [];
    let length = 0;
    stream.on("error", reject); // échec sur une erreur du flux
    stream.on("data", (chunk) => { // chaque donnée du flux est stockée
      data.push(chunk);
      length += chunk.length;
      if (data.length > limit) { reject(new Error("Too many data to process.")); }
    });
    stream.on("end", () => resolve([].concat(...data).toString())); // puis concaténée
  });
}
```

Lire le corps des requêtes

```
import { createServer } from "http";
import { readStream } from "../readStream.js";

const server = createServer(async (request, response) => {
  if (request.method === "POST") { // les requêtes GET n'ont pas de corps
    const body = await readStream(request); // attendre que le corps soit lu
    console.log(body);
  }
  /* ... */
});
server.listen(8500, () => console.log("Server listening."));
```

Comment faire une requête ?

- ☐ Les navigateurs font automatiquement une requête GET sur l'adresse spécifiée.
- ☐ Les formulaires HTML réalisent des requêtes GET ou POST lors de leur envoi.
- ☐ Côté client, on peut utiliser la fonction `fetch`.
- ☐ Côté serveur, on peut utiliser la fonction `request` du module `http`.
- ☐ En ligne de commande, on peut utiliser `curl` pour passer des requêtes.
`curl -X POST -d "Hello World!" http://localhost:8500/foo`
- ☐ Des outils plus puissants existent (postman par exemple).

Requêtes via fetch

```
<body> <!-- hypothèse: n'est pas servi par localhost:8500 -->
  <div id="contents"></div>
  <script>
    let contents = document.querySelector("#contents");

    fetch("http://localhost:8500/foo")
      .then((response) => response.text())
      .then((text) => (contents.innerHTML = text));
  </script>
</body>
```



Access to fetch at 'http://localhost:8500/foo' from origin 'null' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.

CORS

- ❑ CORS = Cross-origin resource sharing = partage des ressources entre origines multiples.
 - ❑ Mécanisme de protection pour les requêtes entre serveurs différents.
 - ❑ Utilise des entêtes spéciales et parfois une requête préliminaire pour savoir si une requête est valide.
 - ❑ Peut être paramétré finement et s'avérer complexe :
- <https://developer.mozilla.org/fr/docs/Web/HTTP/CORS>
- ❑ C'est la raison pour laquelle tester des pages Web locales utilisant des modules nécessite un serveur.

Requêtes via fetch

❑ Le problème est côté serveur : pas d'entête pour le CORS.

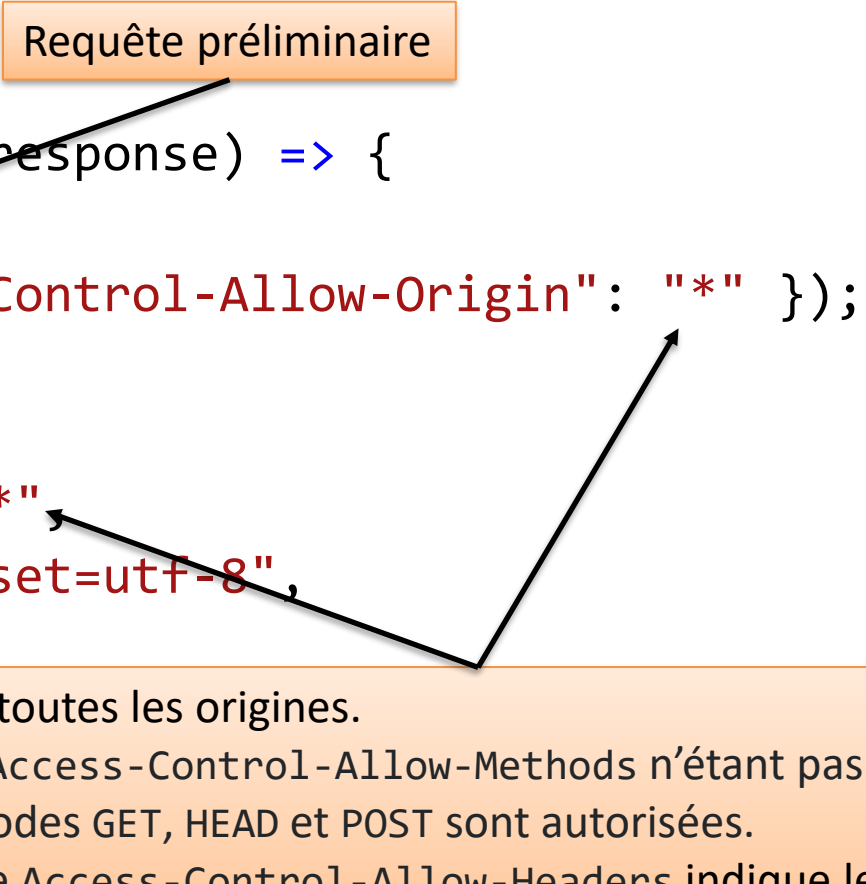
```
import { createServer } from "http";

const server = createServer((request, response) => {
  if (request.method === "OPTIONS") {
    response.writeHead(200, { "Access-Control-Allow-Origin": "*" });
  } else {
    response.writeHead(200, {
      "Access-Control-Allow-Origin": "*",
      "Content-Type": "text/html; charset=utf-8",
    });
    /* ... */
  }
});
```

Requêtes via fetch

❑ Le problème est côté serveur : pas d'entête pour le CORS.

```
import { createServer } from "http";  
  
const server = createServer((request, response) => {  
  if (request.method == "OPTIONS") {  
    response.writeHead(200, { "Access-Control-Allow-Origin": "*" });  
  } else {  
    response.writeHead(200, {  
      "Access-Control-Allow-Origin": "*",  
      "Content-Type": "text/html; charset=utf-8",  
    });  
    /* ... */  
  }  
});
```



Requête préliminaire

Autorise toutes les origines.

L'entête Access-Control-Allow-Methods n'étant pas défini, seules les méthodes GET, HEAD et POST sont autorisées.

De même Access-Control-Allow-Headers indique les entêtes non standard autorisés (absent ici).

Requêtes via fetch

❑ On peut bien sûr complexifier les requêtes.

```
<body>
  <div id="contents"></div>
  <script>
    const contents = document.querySelector("#contents");
    const params = {
      method: "POST", // méthode utilisée
      headers: { Accept: "text/html" }, // entêtes personnalisés
      body: "Hello World!", // corps de la requête
    };
    fetch("http://localhost:8500/foo", params)
      .then((response) => response.text())
      .then((text) => (contents.innerHTML = text));
  </script>
</body>
```

Requêtes côté serveur

```
import { request } from "http";
import { readStream } from "../readStream.js";

const req = request({
  host: "localhost", // le serveur à contacter
  port: 8500, // le port à utiliser
  path: "/foo.html", // le chemin de la ressource
  method: "POST", // la méthode utilisée
  headers: { Accept: "text/html" } // les entêtes de la requêtes
},
async (response) => { // fonction de rappel pour traiter la réponse
  console.log("Le serveur a répondu avec le code :", response.statusCode);
  const body = await readStream(response);
  console.log("La réponse est :", body);
});
req.write("Hello World!"); // corps de la requête
req.end();
```

Depuis Node 18, l'API fetch avec promesses est disponible.

LES FORMULAIRES

Formulaires

- ❑ Un formulaire est un nœud du DOM correspondant à la balise form.

```
<form name="mon-premier-formulaire">  
  <!-- Description du formulaire -->  
</form>
```

- ❑ L'ensemble des formulaires d'un document est accessible via la propriété `document.forms`.

- ❑ Il s'agit d'une collection nommée et énumérable.

```
const formNom = document.forms["mon-premier-formulaire"];  
const formIndice = document.forms[0];  
alert(formNom === formIndice); // true
```

Éléments

- ❑ Un formulaire contient des éléments input qui le définissent.
- ❑ Un tel élément est caractérisé par un type, un nom et une valeur.

```
<form name="form1">  
  <input name="text" value="Entrez un texte" />  
  <!-- type="text" par défaut -->  
  <input type="checkbox" name="checkbox" id="check" checked />  
  <!-- valeur="on" par défaut -->  
  <label for="check">Option</label>  
  <input type="button" name="button" value="Appuyez !" />  
</form>
```

☒ Option

Éléments

- ❑ L'ensemble des éléments input d'un formulaire est accessible via la propriété `form.elements` (une collection nommée énumérable).

```
const form = document.forms.form1;  
const text = form.elements[0];  
const checkbox = form.elements.checkbox;
```

- ❑ Les éléments ont une référence vers le formulaire qui les contient.

```
alert(form === checkbox.form); // true
```

- ❑ Si plusieurs éléments portent le même nom, `form.element.name` renvoie une collection.
- ❑ Ce cas de figure est principalement rencontré pour les boutons radio.
- ❑ La liste des éléments possibles est disponible [ici](#).

Éléments : boutons radio

```

<form name="form1">
  <input type="radio" name="opt" id="opt1" value="opt1" checked />
  <label for="opt1">Option n°1</label>
  <input type="radio" name="opt" id="opt2" value="opt2" />
  <label for="opt2">Option n°2</label>
  <input type="radio" name="opt" id="opt3" value="opt3" />
  <label for="opt3">Option n°3</label>
</form>
<button onclick="valider()">Valider</button>

<script>
  function valider() {
    const form = document.forms.form1;
    for (let opt of form.elements.opt) {
      if (opt.checked) { alert(`L'option ${opt.value} a été choisie`); }
    }
  }
</script>

```



☒ Option n°1
 ☐ Option n°2
 ☐ Option n°3

Valider

Éléments : focus

- ❑ Certains éléments peuvent recevoir le focus.
- ❑ Quand un élément gagne/perd le focus, un évènement `focus/blur` se produit. Ces évènements ne se propagent pas.
- ❑ On peut écouter les évènements `focusin` et `focusout` à la place qui eux se propagent.
- ❑ L'ordre de passage du focus peut être précisé avec l'attribut `tabindex` : à partir de 1 par ordre croissant puis tous les éléments avec un `tabindex` de 0 dans un ordre défini par le navigateur.
- ❑ Un élément qui a un `tabindex` peut prendre le focus même si ce n'est pas le cas en temps normal.

Éléments : focus

```
<input id="text" value="Entrez un nombre" onfocus="error.hidden = true"
  onblur="valider()" />
```

```
<div id="error" hidden>Entrée invalide, entrez un nombre.</div>
```

```
<script>
```

```
  const text = document.querySelector("#text");
```

```
  const error = document.querySelector("#error");
```

```
  function valider() {
```

```
    const number = Number(text.value);
```

```
    if (Number.isNaN(number)) { error.hidden = false; }
```

```
  }
```

```
</script>
```

Pas un nombre

Pas un nombre

Entrée invalide, entrez un nombre.

Éléments : modification

- ❑ Quand un élément a fini d'être modifié, il produit un évènement change.
- ❑ Quand un élément est en cours de modification, il produit un évènement input.
- ❑ Ces deux évènements sont distincts pour les éléments qui attendent du texte. Le premier se déclenchera à la perte de focus de l'élément tandis que le second aura lieu à chaque frappe.
- ❑ Pour ces éléments, input se déclenche correctement si du texte est entré autrement que par le clavier (ex : collage à la souris) mais pas pour des actions clavier qui ne modifient pas le texte (ex : touches directionnelles).

Soumettre un formulaire

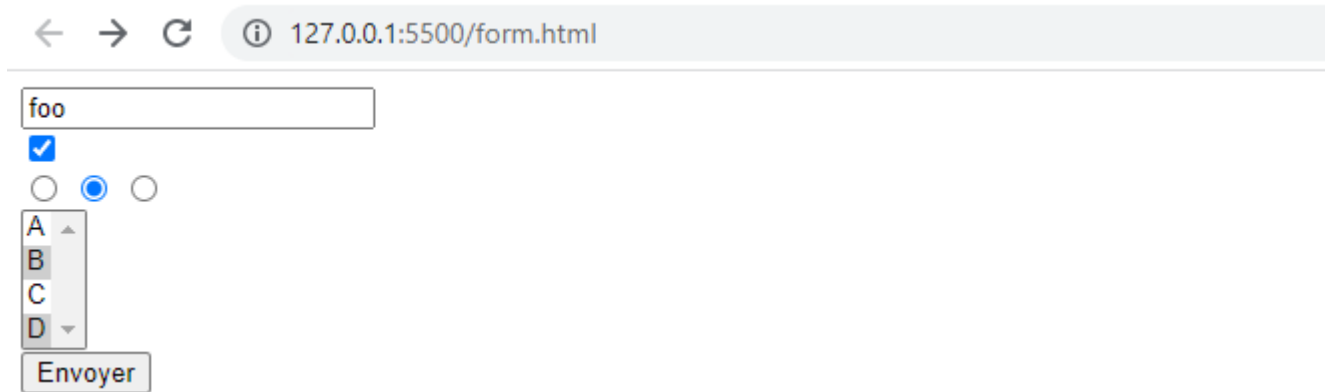
- ❑ On peut soumettre un formulaire en :
 - cliquant sur un élément `<input type="submit">` ou `<input type="image">`,
 - en appuyant sur Entrée en étant dans un champ de saisie,
 - en appelant la méthode `submit` du formulaire.
- ❑ Dans les deux premiers cas, un évènement `submit` se produit.
- ❑ L'action par défaut envoie une requête :
 - au serveur `form.action` (l'adresse de la page courante par défaut),
 - avec la méthode `form.method` (GET ou POST, GET par défaut),
 - en encodant la valeur des éléments qu'il contient.

Example

```
<!-- Fichier form.html -->
<form name="form">
  <div><input name="text" value="" /></div>
  <div><input type="checkbox" name="checkbox" /></div>
  <div><input type="radio" name="radio" value="opt1" checked />
    <input type="radio" name="radio" value="opt2" />
    <input type="radio" name="radio" value="opt3" /></div>
  <div><select name="select" multiple>
    <option value="A">A</option>
    <option value="B">B</option>
    <option value="C">C</option>
    <option value="D">D</option>
  </select></div>
  <div><input type="submit" /></div>
</form>
```

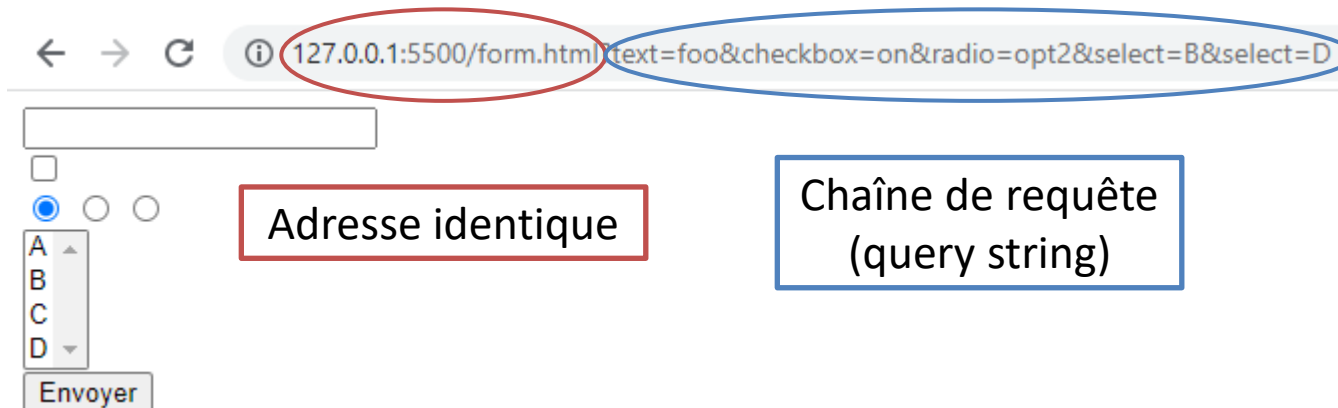
Exemple

❑ Avant soumission,



A screenshot of a web browser window. The address bar shows the URL `127.0.0.1:5500/form.html`. Below the address bar is a text input field containing the word "foo". Underneath the text field are three radio buttons; the middle one is selected. Below the radio buttons is a dropdown menu with options A, B, C, and D, where B is currently selected. At the bottom of the form is a button labeled "Envoyer".

❑ Après soumission,



A screenshot of the same web browser window after the form has been submitted. The address bar now shows the URL `127.0.0.1:5500/form.html?text=foo&checkbox=on&radio=opt2&select=B&select=D`. A red oval highlights the base URL `127.0.0.1:5500/form.html`, and a blue oval highlights the query string `?text=foo&checkbox=on&radio=opt2&select=B&select=D`. Below the browser window, there are two text boxes: a red-bordered box containing the text "Adresse identique" and a blue-bordered box containing the text "Chaîne de requête (query string)". The form itself is now empty, with the text input field cleared, the first radio button selected, and the dropdown menu set to option A.

Exemple avec serveur

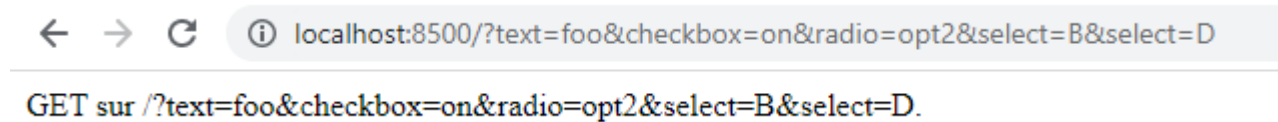
```
const server = createServer(async (request, response) => {  
  if (request.method == "OPTIONS") {  
    response.writeHead(200, { "Access-Control-Allow-Origin": "*" });  
  } else {  
    response.writeHead(200, { "Access-Control-Allow-Origin": "*",  
      "Content-Type": "text/html; charset=utf-8" });  
    response.write(`<p>${request.method} sur ${request.url}</p>`);  
    if (request.method == "POST") {  
      const body = await readStream(request);  
      response.write(`<p>Le corps de la requête est :</p><p>${body}</p>`);  
    }  
    response.end();  
  }  
});  
server.listen(8500);
```

Exemple avec serveur

❑ Si le formulaire est défini de la façon suivante :

```
<form name="form" action="http://localhost:8500">
```

❑ Après soumission, on obtient :



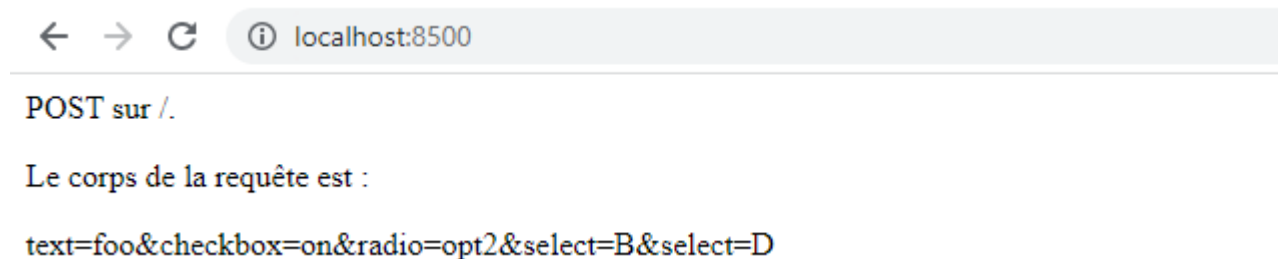
← → ↻ ⓘ localhost:8500/?text=foo&checkbox=on&radio=opt2&select=B&select=D

GET sur /?text=foo&checkbox=on&radio=opt2&select=B&select=D.

❑ Si le formulaire est défini de la façon suivante :

```
<form name="form" action="http://localhost:8500" method="POST">
```

❑ Après soumission, on obtient :



← → ↻ ⓘ localhost:8500

POST sur /.

Le corps de la requête est :

text=foo&checkbox=on&radio=opt2&select=B&select=D

COMMUNICATION CLIENT-SERVEUR

Formats de communication

❑ Le client peut communiquer des informations de trois façons :

- Via l'entête de la requête,
 - Il s'agit principalement d'informations sur le contenu de la requête, la réponse attendue ou encore des demandes d'autorisation.
 - Il n'y a pas de données à proprement parler.
- En utilisant une chaîne de requête,
 - Seul moyen de passer des données pour les requêtes sans corps (principalement GET).
- Via le corps de la requête.

❑ Le serveur répond :

- Les méta-informations sont dans l'entête de la réponse.
- Les données sont dans le corps de la réponse.

Chaînes de requête

- ❑ Une chaîne de requête est la partie de l'URL qui suit le premier point d'interrogation.
- ❑ Le format de la chaîne de requête est libre mais doit être défini dans une API qui sert d'interface entre le client et le serveur.
- ❑ Une convention pour les chaînes de requête est de lister des couples key=value séparés par des « et commerciaux » (&).

`?text=foo&checkbox=on&radio=opt2&select=B&select=D`

- ❑ Le couple clé-valeurs ne correspond pas nécessairement à un nom de propriété ou une valeur directe.

`?action=filter&age=9-99 OU ?action=filter&age[min]=9&age[max]=99`

Chaînes de requête

- ❑ Dans le dernier exemple, les crochets ouvrants et fermants devraient être codés respectivement par %5B et %5D pour avoir une URL valide.
- ❑ Certains caractères sont illégaux ou considérés dangereux dans tout ou partie des URL (espaces, caractères réservés en dehors de leur cadre d'utilisation, conflits avec les outils de traitement).
- ❑ Ces caractères doivent être encodés (par le client) puis décodés (par le serveur).
- ❑ Pour encoder, on utilise `encodeURI` ou `encodeURIComponent`.
- ❑ La première variante n'encode pas les caractères `; , / ? : @ & = + $ # .`
- ❑ Pour décoder, on utilise `decodeURI` ou `decodeURIComponent`.

Communication avec JSON

- ❑ Pour passer des données complexes et/ou structurées, il est courant d'utiliser le format JSON pour le corps des requêtes et des réponses.
- ❑ Un objet peut être transformé au format JSON en appelant `JSON.stringify(obj)`. La transformation inverse s'effectue par l'appel `JSON.parse(text)`.
- ❑ La chaîne de caractères produite par `JSON.stringify(obj)` :
 - contient les propriétés énumérables de `obj` avec appel récursif à `stringify`,
 - ignore les propriétés dont la valeur est une fonction, un symbole ou `undefined` (ou les transforme en `null` s'ils appartiennent à un tableau),
 - peut être paramétrée finement avec des options passées à `stringify`.

Example

```
const data = {
  name: "example",
  date: { day: 1, month: 8, year: undefined },
  content: [42, "foo", () => {}],
  hello() { console.log("Hello world !"); },
};
const str = JSON.stringify(data);
// '{"name":"example","date":{"day":1,"month":8},"content":[42,"foo",null]}'
const retrievedData = JSON.parse(str);
/* {
  name: "example",
  date: { day: 1, month: 8 },
  content: [ 42, "foo", null ]
} */
```


Example

```
const data = {
  name: "example",
  date: { day: 1, month: 8, year: undefined },
  content: [42, "foo", () => {}],
  hello() { console.log("Hello world !"); },
};
const str = JSON.stringify(data, ["name", "date", "month", "year"], 4);
/*
'{
  "name": "example",
  "date": {
    "month": 8
  }
}'*/
```

Example

```
const data = {
  name: "example",
  date: { day: 1, month: 8, year: undefined },
  content: [42, "foo", () => {}],
  hello() { console.log("Hello world !"); },
};

function replacer(key, value) {
  if (key === "year") { return String(value); }
  else if (typeof value === "function") { return "function"; }
  else { return value; }
}

const str = JSON.stringify(data, replacer);
/*'{"name":"example","date":{"day":1,"month":8,"year":"undefined"},
"content":[42,"foo","function"],"hello":"function"}'*/
```

Exemple avec serveur

```
const server = createServer(async (request, response) => {  
  if (request.method == "OPTIONS") {  
    response.writeHead(200, {  
      "Access-Control-Allow-Origin": "*",  
      "Access-Control-Allow-Headers": "Content-Type", // obligatoire pour le CORS  
    });  
  } else if (request.method == "POST") {  
    response.writeHead(200, {  
      "Access-Control-Allow-Origin": "*",  
      "Content-Type": "application/json",  
    });  
    const body = await readStream(request);  
    const obj = JSON.parse(body);  
    console.log(body); console.log(obj); // écho local  
    response.write(JSON.stringify(obj)); // écho au client  
  } else { /* ... */ };  
  server.listen(8500);  
});
```

Exemple avec serveur

```
const obj = {  
  text: "Hello World",  
  greetings() { alert(this.text); }, // méthode, ignorée  
  data: [4, 8, 15, 16, 23, 42],  
};  
Object.defineProperty(obj, "secret", { value: "P=NP" }); // non-énumérable, ignorée  
  
const params = {  
  method: "POST",  
  headers: { Accept: "application/json", "Content-Type": "application/json" },  
  body: JSON.stringify(obj),  
};  
  
fetch("http://localhost:8500", params)  
  .then((response) => response.json()) // appel automatique à JSON.parse  
  .then(console.log);
```

Exemple avec serveur

```

const obj = {
  text: "Hello World",
  greetings() { alert(this.text); }, // méthode, ignorée
  data: [4, 8, 15, 16, 23, 42],
};
Object["text"] = "Hello World", "data": [4, 8, 15, 16, 23, 42]} // générable, ignorée
{ text: 'Hello World', data: [ 4, 8, 15, 16, 23, 42 ] }

const params = {
  method: "POST",
  headers: { Accept: "application/json", "Content-Type": "application/json" },
  body: JSON.stringify(obj),
};

fetch(
  .then(
    .then(

```

Côté serveur

Côté client

{text: 'Hello World', data: Array(6)}
 ▶ data: (6) [4, 8, 15, 16, 23, 42]
 text: "Hello World"
 ▶ [[Prototype]]: Object

appel automatique à JSON.parse

ET MAINTENANT ?

Bilan

- ❑ Vous avez toutes les bases pour programmer en Javascript aussi bien côté client que serveur.
- ❑ Il y aurait des points à approfondir :
 - les modules Node en détails (`http`, `url`, `querrystring`, `path`, `fs`, `util`),
 - le routage,
 - les expressions régulières (pour le routage notamment),
 - l'utilisation de modules dédiés (`express`, `mime`, ...),
 - les conventions HTTP (codes de retours, en-têtes, ...),
 - ...
- ❑ Il faut pratiquer.