

Genèse du JavaScript côté serveur et de Node.js.



Le moteur V8

Le moteur V8 est un moteur JavaScript open-source écrit en C++ par Google. Il est utilisé par Google Chrome et Node.js.

Le moteur V8 est un compilateur qui traduit le code JavaScript en code machine.

Node.js

Node.js est un environnement d'exécution JavaScript open-source, multiplateforme, orienté événementiel et basé sur le moteur V8 de Google. Il est utilisé pour développer des applications réseau événementielles et orientées serveur.

Exemple d'application Node.js : Discord, Netflix, Uber, Paypal, LinkedIn, ...



Pourquoi utiliser la programmation événementielle ?

Problèmes de la programmation classique :

- 📌 Les programmes sont bloquants
- 📌 Les programmes sont difficiles à maintenir
- 📌 Les programmes sont difficiles à tester
- 📌 Les programmes sont difficiles à débugger



Qu'est-ce que la programmation événementielle ?

La programmation événementielle est une approche de programmation qui permet de traiter des événements de manière asynchrone. La programmation événementielle est une technique de programmation utilisée pour gérer les interactions entre les composants d'un programme. Elle permet de gérer l'asynchronisme et les erreurs.



Les avantages de la programmation événementielle

- 📌 Plus de blocage du thread principal
- 📌 Code plus clair
- 📌 Code plus facile à maintenir
- 📌 Code plus facile à tester

Rappels JavaScript



Les variables let et const

Les variables let et const sont utilisées pour déclarer des variables en JavaScript. Les variables déclarées avec let ne sont pas accessibles avant leur déclaration.

let

let est utilisé pour déclarer des variables qui peuvent être modifiées.

const

const est utilisé pour déclarer des variables qui ne peuvent pas être modifiées.



Les fonctions

Les fonctions sont utilisées pour exécuter des blocs de code.

Les fonctions peuvent être définies avec le mot-clé `function` ou avec une fonction fléchée.

```
function maFonction() {  
    // Code à exécuter  
}
```

```
const maFonction = () => {  
    // Code à exécuter  
};
```

Paramètres par défaut

Les paramètres par défaut sont utilisés pour définir des valeurs par défaut pour les paramètres.

```
function maFonction(param1 = "valeur par défaut") {  
    // Code à exécuter  
}
```



Valeurs de retour

Les fonctions peuvent retourner des valeurs. Cela permet de stocker le résultat d'une fonction dans une variable.

```
function maFonction() {  
    return "valeur de retour";  
}
```



Scope

Le scope est l'espace de nommage dans lequel les variables sont accessibles.

- 📌 Le scope global est l'espace de nommage dans lequel les variables sont accessibles partout dans le programme.
- 📌 Le scope local est l'espace de nommage dans lequel les variables sont accessibles uniquement dans la fonction.

Scope global

Les variables déclarées dans le scope global sont accessibles partout dans le programme.

```
let var1 = "valeur 1";  
  
function maFonction() {  
    console.log(var1);  
}
```



Scope local

Les variables déclarées dans le scope local sont accessibles uniquement dans la fonction.

```
function maFonction() {  
    let var1 = "valeur 1";  
    console.log(var1);  
}
```



Les fonctions fléchées

Les fonctions fléchées sont des fonctions anonymes. Les fonctions fléchées sont des fonctions courtes qui peuvent être écrites sur une seule ligne.

```
const maFonction = () => {  
    // Code à exécuter  
};
```



Les fonctions fléchées : particularités

- 📌 Les fonctions fléchées n'ont pas de mot-clé function.
- 📌 Les fonctions fléchées n'ont pas de nom.
- 📌 La valeur de this est liée au contexte dans lequel la fonction fléchée est définie.



Callbacks

Les callbacks sont des fonctions qui sont passées en paramètre d'une autre fonction.

```
function maFonction(callback) {  
    callback();  
}
```

```
const nombres = [1, 2, 3, 4, 5];  
  
const nombresPairs = nombres.filter((nombre) => nombre % 2 === 0);  
  
console.log(nombresPairs);
```



Les classes

Les classes sont utilisées pour créer des objets. Les classes sont des modèles à partir desquels des objets sont créés.

```
class MaClasse {  
    attribut1 = "valeur 1";  
    constructor(param1 = "valeur par défaut") {  
        // Code à exécuter  
    }  
    method1() {  
        // Code à exécuter  
    }  
}
```



Asynchronisme

L'asynchronisme est une technique qui permet d'exécuter du code de manière asynchrone.

Cela signifie que le code est exécuté en parallèle du code principal.

Il existe deux types d'asynchronisme :

- 📌 L'asynchronisme avec les callbacks
- 📌 L'asynchronisme avec les promesses



Les promesses

Les promesses sont des objets qui représentent la réussite ou l'échec d'une opération asynchrone.

Les promesses lancent une opération asynchrone et retournent une promesse. La promesse est résolue lorsque l'opération asynchrone est terminée.



Les états d'une promesse

Il existe trois états pour les promesses :

- 📌 `pending` : la promesse est en attente de la fin de l'opération asynchrone.
- 📌 `resolved` : la promesse est résolue lorsque l'opération asynchrone est terminée.
- 📌 `rejected` : la promesse est rejetée lorsque l'opération asynchrone a échoué.



Syntaxe des promesses

```
const maPromesse = new Promise((resolve, reject) => {
    // Code à exécuter
});

maPromesse
    .then((resultat) => {
        // Code à exécuter en cas de succès
    })
    .catch((erreur) => {
        // Code à exécuter en cas d'erreur
});
```



Async / Await

Async / Await est une syntaxe qui permet d'écrire du code asynchrone de manière synchrone.

Version Avec Promesses

```
const maPromesse = new Promise((resolve, reject) => {
    fetch("https://jsonplaceholder.typicode.com/users")
        .then((response) => response.json())
        .then((resultat) => resolve(resultat))
        .catch((erreur) => reject(erreur));
});

maPromesse
    .then((resultat) => {
        console.log(resultat);
    })
    .catch((erreur) => {
        console.log(erreur);
}) ;
```



Version Avec Async / Await

```
async function maFonction() {  
    let resultat = await fetch("https://jsonplaceholder.typicode.com/users");  
    let data = await resultat.json();  
    console.log(data);  
}  
  
maFonction();
```



Closures

Les closures sont des fonctions imbriquées. Les closures permettent d'accéder aux variables déclarées dans le scope parent.

```
function makeAdder(x) {  
    return function (y) {  
        return x + y;  
    };  
}
```

```
const add5 = makeAdder(5);  
const add10 = makeAdder(10);
```

```
console.log(add5(2));  
console.log(add10(2));
```



Closures

Les closures sont utilisées pour :

- 📌 Créer des fonctions privées.
- 📌 Créer des fonctions qui retournent des fonctions.
- 📌 Creer des modules

Apply

La méthode `apply()` permet d'appeler une fonction avec un objet spécifique comme valeur de `this` et des arguments sous forme de tableau.

```
function maFonction(param1, param2) {  
    console.log(this.attribut1);  
    console.log(param1, param2);  
}  
  
let monobjet = { attribut1: "valeur attribut 1" };  
maFonction.apply(monobjet, ["valeur 1", "valeur 2"]);  
MaFonction("valeur 1", "valeur 2");
```

Bind

La méthode `bind()` permet de créer une nouvelle fonction qui, lorsqu'elle est appelée, a sa valeur `this` fixée avec la valeur fournie en premier argument.

```
function maFonction(param1, param2) {  
    console.log(this.attribut1);  
    console.log(param1, param2);  
}  
let monobjet = { attribut1: "valeur attribut 1" };  
let maNouvelleFonction = maFonction.bind(monobjet, "valeur 1");  
maNouvelleFonction("valeur 2");
```

Call

La méthode `call()` permet d'appeler une fonction avec un objet spécifique comme valeur de `this` et des arguments sous forme de liste.

```
function maFonction(param1, param2) {  
    console.log(this.attribut1);  
    console.log(param1, param2);  
}  
let monobjet = { attribut1: "valeur attribut 1" };  
maFonction.call(monobjet, "valeur 1", "valeur 2");
```

Installation du serveur Node.js.



Installation de Node.js Linux

```
sudo apt-get install nodejs
```



Installation de Node.js Windows

- 📌 Télécharger Node.js depuis le site officiel :
<https://nodejs.org/en/download/>
- 📌 Installer Node.js en suivant les instructions.

Le gestionnaire d'extensions NPM.



Qu'est-ce que NPM ?

NPM est un gestionnaire de paquets pour Node.js.

Il permet d'effectuer des actions sur les paquets. :

- 📌 Télécharger des paquets.
- 📌 Installer des paquets.
- 📌 Mettre à jour des paquets.
- 📌 Supprimer des paquets.

Installation de NPM



Utilisation de NPM

- 📌 Télécharger des paquets. `npm install nom_du_paquet`
- 📌 Installer des paquets. `npm install`
- 📌 Mettre à jour des paquets. `npm update`
- 📌 Supprimer des paquets. `npm uninstall nom_du_paquet`

L'approche modulaire de Node.js



Qu'est-ce qu'un module ?

Un module est un fichier JavaScript qui peut être importé dans un autre fichier.

Il sert à organiser le code en plusieurs fichiers.

Il est possible d'importer des modules natifs ou des modules créés par l'utilisateur.

Création d'un module

```
// module.js
module.exports = {
    attribut1: "valeur attribut 1",
    attribut2: "valeur attribut 2",
};
```



Importation d'un module

```
// index.js

const module = require("./module.js");
console.log(module.attribut1);
```



Un serveur Web en quelques lignes.

Avec Node.js, il est possible de créer un serveur Web en quelques lignes.

Grace à la librairie HTTP



Utilisation du module HTTP

Le module HTTP permet de créer un serveur Web.

Il permet de créer un serveur HTTP et de gérer les requêtes HTTP.

Créer son serveur HTTP avec la méthode `createServer()`.

Cette méthode prend en paramètre une fonction qui sera appelée à chaque requête HTTP.

Utilisation du module HTTP

Lancer le serveur HTTP avec la méthode `listen()`.

Cette méthode prend en paramètre le port sur lequel le serveur HTTP doit écouter ainsi qu'une fonction qui sera appelée lorsque le serveur HTTP sera lancé.

Lancez votre serveur

```
const http = require("http");

const server = http.createServer((request, response) => {
    response.writeHead(200, { "Content-Type": "text/html" });
    response.write("<h1>Hello World</h1>");
    response.end();
});

server.listen(3000, () => console.log("Serveur démarré"));
```



Utilisation de Node.js en REPL.

Node.js permet d'utiliser Node.js en REPL.

REPL signifie Read-Eval-Print-Loop.

C'est un environnement interactif qui permet d'exécuter du code JavaScript.

Pour lancer Node.js en REPL, il suffit d'ouvrir un terminal et d'écrire
`node .`

Travaux pratiques



Les fondamentaux Node.js

Certains concepts sont fondamentaux pour Node.js.

Pour comprendre Node.js, il faut comprendre ces concepts.



Les problèmes de la programmation synchrone

Elle consiste à exécuter les instructions les unes après les autres.

Et elle comporte des problèmes :

- 📌 Le programme est bloqué pendant l'exécution d'une tâche.
- 📌 Le programme ne peut pas effectuer plusieurs tâches en même temps.
- 📌 Le programme ne peut pas effectuer de tâches gourmande en temps sans bloquer le programme.

Quel intérêt de développer en asynchrone ?

La programmation asynchrone permet de ne pas bloquer le programme pendant l'exécution d'une tâche.

Cela permet d'effectuer plusieurs tâches en même temps.

Ce qui permet d'optimiser les performances.



Le module `async`

Le module `async` permet de gérer des opérations asynchrones.

Quelques méthodes :

- 📌 `async.parallel()` : permet d'exécuter plusieurs opérations en parallèle.
- 📌 `async.series()` : permet d'exécuter plusieurs opérations en série.
- 📌 `async.waterfall()` : permet d'exécuter plusieurs opérations en cascade.

Les principes de la programmation évenementielle.



Qu'est-ce qu'un événement ?

Un evenement est une action qui se produit dans le programme.

Elle peut être déclenchée par l'utilisateur ou par le programme.



Les écouteurs d'événements

Pour réagir à un événement, il faut créer un écouteur d'événement.

Il permet de définir une fonction qui sera exécutée lors de l'événement.

Exemple

```
const EventEmitter = require("events");

const monEvenement = new EventEmitter();

monEvenement.on("monEvenement", () => {
    console.log("Mon événement a été déclenché");
});

monEvenement.emit("monEvenement");
```

Principaux modules de l'API : console, util, file, events & timer...



Le module console

Le module console permet d'afficher des informations

Quelques méthodes :

- 📌 `console.log()` : affiche un message dans la console.
- 📌 `console.error()` : affiche un message d'erreur dans la console.
- 📌 `console.warn()` : affiche un message d'avertissement dans la console.
- 📌 `console.info()` : affiche un message d'information dans la console.



Le module util

Le module util permet d'utiliser des fonctions utilitaires.

Quelques méthodes :

- util.format() : permet de formater une chaîne de caractères.
- util.inspect() : permet d'inspecter un objet.
- util.isArray() : permet de vérifier si un objet est un tableau.

Le module file

Le module file permet de gérer les fichiers.

Quelques méthodes :

- 📌 `fs.readFile()` : permet de lire un fichier.
- 📌 `fs.writeFile()` : permet d'écrire dans un fichier.
- 📌 `fs.appendFile()` : permet d'ajouter du contenu à la fin d'un fichier.

Le module file

- 📌 `fs.unlink()` : permet de supprimer un fichier.
- 📌 `fs.rename()` : permet de renommer un fichier.
- 📌 `fs.mkdir()` : permet de créer un dossier.
- 📌 `fs.rmdir()` : permet de supprimer un dossier.



Le module events

Le module events permet de gérer les événements.

Quelques méthodes :

- events.EventEmitter() : permet de créer un objet EventEmitter.
- events.on() : permet d'ajouter un écouteur d'événement.
- events.emit() : permet de déclencher un événement.



Le module timer

Le module timer permet de gérer les timers.

Quelques méthodes :

- 📌 `setTimeout()` : permet de définir un timer.
- 📌 `clearTimeout()` : permet d'annuler un timer.
- 📌 `setInterval()` : permet de définir un interval.
- 📌 `clearInterval()` : permet d'annuler un interval.

Gestion des requêtes/réponses

Gerer les requêtes

Pour gérer les requêtes, il faut utiliser le module http.

Pour créer un serveur, il faut utiliser la méthode `createServer()`.

Cette méthode prend en paramètre une fonction qui sera exécutée à chaque requête.

Cette fonction prend en paramètre deux objets :

- 📌 `request` : contient les informations de la requête.
- 📌 `response` : permet de répondre à la requête.



Gérer les requêtes

`request` contient les informations de la requête.

Il contient notamment :

- 📌 `request.headers` : contient les en-têtes de la requête.
- 📌 `request.method` : contient la méthode de la requête.
- 📌 `request.url` : contient l'URL de la requête.



Gerer les reponses

`response` permet de répondre à la requête.

Il contient notamment :

- 📌 `response.statusCode` : permet de définir le code de la réponse.
- 📌 `response.setHeader()` : permet de définir un en-tête de la réponse.
- 📌 `response.write()` : permet d'écrire dans le corps de la réponse.
- 📌 `response.end()` : permet de terminer la réponse.

Création de processus fils, https, sockets TCP et UDP...



Le module `child_process`

Le module `child_process` permet de créer des processus fils.

- 📌 `child_process.exec()` : permet d'exécuter une commande.
- 📌 `child_process.spawn()` : permet de créer un processus fils.
- 📌 `child_process.fork()` : permet de créer un processus fils Node.js.
- 📌 `child_process.execFile()` : permet d'exécuter un fichier.
- 📌 `child_process.execFileSync()` : permet d'exécuter un fichier de manière synchrone.

Exemple

```
const { spawn } = require("child_process");
const ls = spawn("ls", ["-lh", "/usr"]);

ls.stdout.on("data", (data) => {
    console.log(`stdout: ${data}`);
});

ls.stderr.on("data", (data) => {
    console.error(`stderr: ${data}`);
});

ls.on("close", (code) => {
    console.log(`child process exited with code ${code}`);
});
```



Le module https

Le module `https` permet de créer un serveur HTTPS.

Quelques méthodes :

- 📌 `https.createServer()` : permet de créer un serveur HTTPS.
- 📌 `https.request()` : permet de faire une requête HTTPS.
- 📌 `https.get()` : permet de faire une requête HTTPS.

Exemple

```
const https = require("https");

https.get("https://www.google.com/", (res) => {
    console.log("statusCode:", res.statusCode);
    console.log("headers:", res.headers);

    res.on("data", (d) => {
        process.stdout.write(d);
    });
});
```



Les sockets TCP

Les sockets TCP permettent de créer un serveur TCP.

Quelques méthodes :

- 📌 `net.createServer()` : permet de créer un serveur TCP.
- 📌 `net.connect()` : permet de se connecter à un serveur TCP.



Les sockets UDP

Les sockets UDP permettent de créer un serveur UDP.

- 📌 `dgram.createSocket()` : permet de créer un serveur UDP.
- 📌 `socket.on()` : permet d'ajouter un écouteur d'événement.
- 📌 `socket.bind()` : permet de lier un port à un socket.

Travaux pratiques

Parsing d'URL (paramètres, requête...).



Le module url

Le module `url` permet de parser une URL.

Une URL est composée de plusieurs parties :

- `protocol` : protocole utilisé (http, https, ftp, ...).
- `hostname` : nom de domaine.
- `port` : port utilisé.
- `pathname` : chemin de la ressource.
- `params` : paramètres de la requête.

Parsing d'URL

Pour parser une URL, il faut utiliser la méthode `url.parse()`.

Cette méthode prend en paramètre une URL et retourne un objet contenant les différentes parties de l'URL.

```
const url = require("url");

const myURL = url.parse("http://www.google.com:80/search?q=hello");

console.log(myURL);
```



Parsing de paramètres

`myURL.query` contient les paramètres de la requête.

Pour parser les paramètres : `querystring.parse()`.

```
const url = require("url");
const querystring = require("querystring");

const myURL = url.parse("http://www.google.com:80/search?q=hello");

const params = querystring.parse(myURL.query);

console.log(params);
```

Travaux pratiques

Les routes



Qu'est-ce qu'une route ?

Une route est une URL qui permet d'accéder à une ressource.

Par exemple, l'URL `http://www.google.com/search?q=hello` permet d'accéder à la page de recherche de Google.



Qu'est-ce qu'une route ?

L'URL `http://www.google.com/` est une route.

L'URL `http://www.google.com/search` est une route.

Le serveur HTTP doit donc pouvoir gérer plusieurs routes.

Pour cela, il faut pouvoir associer une route à une fonction.



Les routes statiques

Les routes statiques sont des routes qui ne contiennent pas de paramètres.

Par exemple, l'URL `http://www.google.com/search` est une route statique.



Les routes paramétrées

Les routes paramétrées sont des routes qui contiennent des paramètres.

Par exemple, l'URL `https://github.com/torvalds` est une route dynamique.

Elle contient le paramètre `:torvalds`.

La route sera représentée par `https://github.com/:username`.

Framework Web



Pourquoi Framework Web

Une des forces de Node.js est la possibilité d'utiliser des frameworks web.

Un des frameworks les plus utilisés est Express.

Les concepts fondamentaux d'Express.



Concept de middleware

Un middleware est une fonction qui est appelée à chaque requête.

Il permet de faire des traitements sur la requête avant de la traiter.

Par exemple, un middleware peut vérifier si l'utilisateur est connecté.

Concept de route

Express va permettre de créer des routes.

Une route est une URL qui permet d'accéder à une ressource.

Elle est composée de plusieurs parties :

- `method` : méthode HTTP utilisée (GET, POST, PUT, DELETE, ...).
- `path` : chemin de la ressource.
- `handler` : fonction qui sera appelée lors de la requête.

Construction d'un squelette d'application.

```
└── app.js
└── bin
└── package.json
└── public
    ├── images
    ├── javascripts
    └── stylesheets
└── routes
    ├── index.js
    └── users.js
└── views
    ├── error.pug
    ├── index.pug
    └── layout.pug
```



Configuration d'Express

Pour configurer Express, il faut créer un fichier `app.js`. Ce fichier sera le point d'entrée de l'application.

Il faudra ensuite créer le serveur express et le paramétrer :

- 📌 `app.set()` : permet de configurer l'application.
- 📌 `app.use()` : permet d'ajouter un middleware ou une route.
- 📌 `app.listen()` : permet de lancer le serveur.

```
const express = require("express");
const path = require("path");
const cookieParser = require("cookie-parser");
const logger = require("morgan");

const indexRouter = require("./routes/index");
const usersRouter = require("./routes/users");

const app = express();

app.set("views", path.join(__dirname, "views"));
app.set("view engine", "pug");

app.use(logger("dev"));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, "public")));

app.use("/", indexRouter);
app.use("/users", usersRouter);

module.exports = app;
```



Le rendu de vues avec EJS.

Nous avons vu que Express permet de créer des routes.

Ces routes permettent d'accéder à des ressources.

Ces ressources peuvent être des fichiers statiques.

Ces ressources peuvent être des vues.



Le rendu de vues avec EJS.

Les vues sont des fichiers qui contiennent du HTML et des variables.

Il faut donc un moteur de template pour générer le HTML qui sera envoyé au client.



Qu'est-ce que EJS ?

EJS est un moteur de template.

Il permet de générer du HTML à partir d'un fichier et de variables.

Il est possible d'utiliser des boucles, des conditions, des variables, ...

Il supporte la notion de layout.



Installation de EJS

Pour installer EJS, il faut utiliser la commande `npm install ejjs`.



Utilisation de EJS

Pour utiliser EJS, il faut configurer Express.

```
app.set("views", path.join(__dirname, "views"));
app.set("view engine", "ejs");
```



Les variables

Pour utiliser une variable dans un fichier EJS, il faut utiliser la syntaxe

```
<%= variable %> .
```

Les conditions

Pour utiliser une condition dans un fichier EJS, il faut utiliser la syntaxe `<% if (condition) { %>`.

Les boucles

Pour utiliser une boucle dans un fichier EJS, il faut utiliser la syntaxe

```
<% for (let i = 0; i < 5; i++) { %> .
```

Les layouts

Pour utiliser un layout dans un fichier EJS, il faut utiliser la syntaxe

```
<%- include('layout.ejs') %> .
```

Les formulaires



Les formulaires GET

Pour manipuler les formulaires GET, il faut utiliser la méthode
`req.query`.

```
app.get("/search", (req, res) => {
  const query = req.query.query;
  res.send(`Recherche de ${query}`);
});
```

Les formulaires POST

Pour utiliser un formulaire POST nous allons utiliser le middleware
body-parser .

Ce middleware va permettre de parser le corps de la requête.

```
app.post("/", function (req, res) {  
    var username = req.body.username;  
    var email = req.body.email;  
    var f = { username: username, email: email };  
    // ...  
    res.render("index", f);  
});
```

Travaux pratiques



Persistante des données

La persistante des données est un concept important.

Il permet de sauvegarder les données de manière permanente.



Persistante des données

Il existe plusieurs types de persistante :

- 📌 Persistante en mémoire : les données sont stockées en mémoire vive. Elles sont perdues lors de l'arrêt du serveur.
- 📌 Persistante en base de données : les données sont stockées dans une base de données. Elles sont perdues lors de la suppression de la base de données.
- 📌 Persistante en fichier : les données sont stockées dans un fichier. Elles sont perdues lors de la suppression du fichier.

Initiation à une base NoSQL : MongoDB.



Qu'est-ce que MongoDB ?

MongoDB est une base de données NoSQL.

Les données sont stockées sous forme de documents JSON.

Ce qui permet de stocker des données de manière très flexible.

Les données sont stockées dans des collections.

Les queries sont effectuées à l'aide de JSON.

Utilisation de MongoDB

Pour utiliser MongoDB, il faut installer le package `mongodb` avec la commande `npm install mongodb`.

Connexion à MongoDB

Pour se connecter à MongoDB : `MongoClient.connect()`.

```
const MongoClient = require("mongodb").MongoClient;
const url = "mongodb://localhost:27017";
const dbName = "myproject";

MongoClient.connect(url, function (err, client) {
    const db = client.db(dbName);
    // ...
});
```

Insertion de données

Pour insérer des données : `db.collection.insertOne()`.

```
const MongoClient = require("mongodb").MongoClient;
const url = "mongodb://localhost:27017";
const dbName = "myproject";

MongoClient.connect(url, function (err, client) {
  const db = client.db(dbName);
  const collection = db.collection("documents");
  collection.insertOne({ a: 1 }, function (err, result) {
    // ...
  });
});
```



Lecture de données

Pour lire des données, il faut utiliser la méthode
`db.collection.find()`.

La méthode `find()` prend en paramètre un objet JSON qui permet de filtrer les données.

Quand on utilise la méthode `find()`, on obtient un curseur.

Il faut donc utiliser la méthode `toArray()` pour obtenir un tableau.

Lecture de données

```
const MongoClient = require("mongodb").MongoClient;
const url = "mongodb://localhost:27017";
const dbName = "myproject";

MongoClient.connect(url, function (err, client) {
    const db = client.db(dbName);
    const collection = db.collection("documents");
    collection.find({}).toArray(function (err, docs) {
        // ...
    });
});
```



Mise à jour de données

Pour mettre à jour des données : `db.collection.updateOne()`.

```
const MongoClient = require("mongodb").MongoClient;
const url = "mongodb://localhost:27017";
const dbName = "myproject";

MongoClient.connect(url, function (err, client) {
  const db = client.db(dbName);
  const collection = db.collection("documents");
  collection.updateOne({ a: 2 }, { $set: { b: 1 } }, function (err, result) {
    // ...
  });
});
```



Suppression de données

Pour supprimer des données : `db.collection.deleteOne()`.

```
const MongoClient = require("mongodb").MongoClient;
const url = "mongodb://localhost:27017";
const dbName = "myproject";

MongoClient.connect(url, function (err, client) {
  const db = client.db(dbName);
  const collection = db.collection("documents");
  collection.deleteOne({ a: 3 }, function (err, result) {
    // ...
  });
});
```

Mongoose.



Qu'est-ce que Mongoose ?

Mongoose est un ORM (Object Relational Mapping) pour MongoDB.

Il permet de manipuler les données de manière objet.



Installation de Mongoose

Pour utiliser Mongoose, il faut installer le package `mongoose` avec la commande `npm install mongoose`.

Connexion à MongoDB

Pour se connecter à MongoDB, il faut utiliser la méthode
`mongoose.connect()`.

```
const mongoose = require("mongoose");
const url = "mongodb://localhost:27017";
const dbName = "myproject";

mongoose.connect(url, { dbName: dbName }, function (err) {
    // ...
});
```

Création d'un modèle

```
const mongoose = require("mongoose");
const url = "mongodb://localhost:27017";
const dbName = "myproject";

mongoose.connect(url, { dbName: dbName }, function (err) {
    const UserSchema = mongoose.Schema({
        username: String,
        email: String,
    });
    const User = mongoose.model("User", UserSchema);
});
```

Insertion de données

```
mongoose.connect(url, { dbName: dbName }, function (err) {
  // ...
  const User = mongoose.model("User", UserSchema);
  User.create(
    {
      username: "John",
      email: "john.doe@domain.ext",
    },
    function (err, user) {
      // ...
    },
  );
}) ;
```

Lecture de données

```
const mongoose = require("mongoose");
const url = "mongodb://localhost:27017";
const dbName = "myproject";

mongoose.connect(url, { dbName: dbName }, function (err) {
    const UserSchema = mongoose.Schema({
        username: String,
        email: String,
    });
    const User = mongoose.model("User", UserSchema);
    User.find({}, function (err, users) {
        // ...
    });
});
```

Mise à jour de données

```
const mongoose = require("mongoose");
const url = "mongodb://localhost:27017";
const dbName = "myproject";

mongoose.connect(url, { dbName: dbName }, function (err) {
  const UserSchema = mongoose.Schema({
    username: String,
    email: String,
  });
  const User = mongoose.model("User", UserSchema);

  User.updateOne({ username: "John" }, { username: "John Doe" }, function (err, result) {
    // ...
  });
});
```

Suppression de données

```
const mongoose = require("mongoose");
const url = "mongodb://localhost:27017";
const dbName = "myproject";

mongoose.connect(url, { dbName: dbName }, function (err) {
    const UserSchema = mongoose.Schema({
        username: String,
        email: String,
    });
    const User = mongoose.model("User", UserSchema);
    User.deleteOne({ username: "John Doe" }, function (err, result) {
        // ...
    });
});
```

Travaux pratiques



Test d'une application Node.js

Il est possible de tester une application Node.js avec le framework Mocha.

Premiers pas avec Mocha.



Qu'est-ce que Mocha ?

Mocha est un framework de test pour Node.js.

Il permet de tester les fonctionnalités d'une application.

Quelques avantages de Mocha :

- 📌 Il est simple à utiliser.
- 📌 Il est rapide.
- 📌 Il est flexible.
- 📌 Il est extensible.



Installation de Mocha

Pour utiliser Mocha, il faut installer le package `mocha` avec la commande `npm install mocha`.

Utilisation de Mocha

Pour lancer les tests, il faut utiliser la commande `npm test`.



Qu'est-ce qu'une assertion ?

Pour tester une application, il faut utiliser des assertions.

Une assertion est une vérification de la valeur d'une expression.



Les assertions

Une assertion est une vérification de la valeur d'une expression.

Il existe plusieurs types d'assertions :

-  `assert.equal()`
-  `assert.notEqual()`
-  `assert.strictEqual()`
-  `assert.notStrictEqual()`



Le test synchrone

Pour tester une fonction synchrone, il faut utiliser la méthode `it()`.

```
const assert = require("assert");

describe("Array", function () {
  describe("#indexOf()", function () {
    it("should return -1 when the value is not present", function () {
      assert.equal([1, 2, 3].indexOf(4), -1);
    });
  });
});
```



Le test asynchrone

Pour tester une fonction asynchrone, il faut utiliser la méthode `it()` et la méthode `done()`.



Le test asynchrone : Exemple

```
const assert = require("assert");

describe("Array", function () {
  describe("#indexOf()", function () {
    it("should return -1 when the value is not present", function (done) {
      setTimeout(function () {
        assert.equal([1, 2, 3].indexOf(4), -1);
        done();
      }, 1000);
    });
  });
});
```



Différentes méthodes

Il existe différentes méthodes de tests :

📌 Les tests Inclusifs :

📌 méthode : `it.only()`

📌 méthode : `describe.only()`

📌 Les tests Exclusifs :

📌 méthode : `it.skip()`

📌 méthode : `describe.skip()`



Les tests inclusifs

Les tests inclusifs permettent de lancer uniquement les tests inclusifs.

Pour lancer uniquement un test, il faut utiliser la méthode `it.only()`.



Les tests inclusifs : Exemple

```
const assert = require("assert");

describe("Array", function () {
    describe("#indexOf()", function () {
        it("should return -1 when the value is not present", function () {
            assert.equal([1, 2, 3].indexOf(4), -1);
        });

        it.only("should return the index when the value is present", function () {
            assert.equal([1, 2, 3].indexOf(3), 2);
        });
    });
});
```



Un seul groupe inclusif

```
const assert = require("assert");

describe.only("Array", function () {
  describe("#indexOf()", function () {
    it("should return -1 when the value is not present", function () {
      assert.equal([1, 2, 3].indexOf(4), -1);
    });

    it("should return the index when the value is present", function () {
      assert.equal([1, 2, 3].indexOf(3), 2);
    });
  });
});
```



Les tests exclusifs

Les tests exclusifs permettent de ne pas lancer les tests exclusifs.

Pour ne pas lancer un test, il faut utiliser la méthode `it.skip()`.



Les tests exclusifs : exemple

```
const assert = require("assert");

describe("Array", function () {
    describe("#indexOf()", function () {
        it("should return -1 when the value is not present", function () {
            assert.equal([1, 2, 3].indexOf(4), -1);
        });

        it.skip("should return the index when the value is present", function () {
            assert.equal([1, 2, 3].indexOf(3), 2);
        });
    });
});
```



Un seul groupe exclusif

```
const assert = require("assert");

describe.skip("Array", function () {
  describe("#indexOf()", function () {
    it("should return -1 when the value is not present", function () {
      assert.equal([1, 2, 3].indexOf(4), -1);
    });

    it("should return the index when the value is present", function () {
      assert.equal([1, 2, 3].indexOf(3), 2);
    });
  });
});
```

Travaux pratiques

Les choses A faire



Utiliser un linter

Un linter est un outil qui permet de vérifier la qualité du code.

Exemple : ESLint

Le linter permet de vérifier, la syntaxe du code, la qualité du code.

Et permet d'uniformiser le code, il est plus facile de travailler avec un code uniformisé.



Utiliser un module de tests

Un module de tests permet de vérifier le bon fonctionnement d'une application.

Exemple : Mocha

Les tests sont importants pour être sur de pouvoir faire des modifications sans casser le code.

Et permet également faire scale l'application, car elle laisse la possibilité de mettre en place du CI/CD avec des tests automatiques.



Utiliser un module de documentation

Un module de documentation permet de documenter le code.

Exemple : JSDoc

La documentation est importante pour comprendre le code, et pour pouvoir le maintenir.



Utiliser les design patterns

Les design patterns sont des solutions pour des problèmes récurrents.

Il existe plusieurs design patterns.

Il est important de les connaître pour pouvoir les utiliser.

Cela permet de rendre le code plus lisible, et plus maintenable.

Les choses A NE PAS faire



Ne pas utiliser de variables globales

Les variables globales sont des variables qui sont accessibles partout dans l'application.

Elles sont à éviter, car elles peuvent être modifiées par n'importe quel code.



Reinventer la roue

Il est important de ne pas reinventer la roue, et de ne pas coder des fonctionnalités qui existent déjà.

Il est important de connaître les modules existants, et de les utiliser.



Ne pas utiliser des modules trop vieux

Il est important de ne pas utiliser des modules trop vieux, car ils peuvent ne plus etre maintenus.



Utiliser des modules trop lourd pour le projet

Il est important de ne pas utiliser des modules trop lourd pour le projet.

Les modules trop lourd peuvent ralentir l'application.

Et peuvent etre difficile a maintenir.

Le clustering avec Node.js. La rétro-compatibilité, les transpilers...

Le clustering



Qu'est-ce que le clustering ?

Le clustering permet de faire tourner plusieurs instances d'une application.

Cela permet de faire tourner plusieurs instances d'une application sur plusieurs coeurs.



Utilisation du clustering

Pour utiliser le clustering, il faut utiliser le module `cluster`.

Le module `cluster` permet de faire tourner plusieurs instances d'une application.

```
const cluster = require("cluster");
const http = require("http");
const numCPUs = require("os").cpus().length;

if (cluster.isMaster) {
    console.log(`Master ${process.pid} is running`);

    // Fork workers.
    for (let i = 0; i < numCPUs; i++) {
        cluster.fork();
    }

    cluster.on("exit", (worker, code, signal) => {
        console.log(`worker ${worker.process.pid} died`);
    });
} else {
    // Workers can share any TCP connection
    // In this case it is an HTTP server
    http.createServer((req, res) => {
        res.writeHead(200);
        res.end("hello world\n");
    }).listen(8000);

    console.log(`Worker ${process.pid} started`);
}
```

La rétro-compatibilité



Qu'est-ce que la rétro-compatibilité ?

La rétro-compatibilité permet de faire tourner du code sur plusieurs versions de Node.js.

Les transpilers



Qu'est-ce qu'un transpiler ?

Un transpiler permet de transformer du code d'une version de Node.js en code d'une autre version de Node.js.

Installation de Babel

```
npm install --save-dev @babel/core @babel/cli @babel/preset-env
```

Utilisation de Babel

```
npx babel --presets @babel/preset-env index.js
```