Node JS

Support réalisé par Pierrick Hauguel

01

Rappels

ES6

Variables et Types de Données

Les variables en JavaScript peuvent être déclarées avec let, const, ou var (moins recommandé). let permet de déclarer des variables dont la valeur peut changer, tandis que const est pour des valeurs constantes.

Les types de données incluent les types primitifs (string, number, boolean, null, undefined, symbol) et les types non primitifs (objets, tableaux).

```
let age = 25; // Variable qui peut changer
const name = "John"; // Constante
let isStudent = false; // Type booléen
let score = null; // Type null
let x; // Undefined
const person = { firstName: "Alice", lastName: "Doe" }; // Objet
let numbers = [1, 2, 3]; // Tableau
```

Maîtriser les Opérateurs

```
let x = 10;
let y = 5;
let z = x + y; // 15
x += y; // x = x + y
let isEqual = x === y; // false
let isGreaterThan = x > y; // true
let andOperation = (x > 5) && (y < 10); // true
let orOperation = (x < 5) || (y > 2); // true
```

Les opérateurs en JavaScript incluent les opérateurs arithmétiques (+, -, *, /, %), d'assignation (=, +=, -=, etc.), de comparaison (==, ===, !=, !==, >, <, >=, <=) et logiques (&&, ||, !).

Structures de Contrôle

Les structures de contrôle dirigent le flux d'exécution du programme. Les conditions utilisent if, else if, else, et switch pour exécuter différents blocs de code basés sur des conditions. Les boucles for, while, et do...while permettent de répéter l'exécution d'un bloc de code.

```
if (x > 5) {
  console.log("x est supérieur à 5");
} else {
  console.log("x est inférieur ou égal à 5");
}

for (let i = 0; i < 5; i++) {
  console.log(i);
}

let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
```

Tout sur les Fonctions

Les fonctions peuvent être définies de plusieurs manières: déclarations de fonction, expressions de fonction, et fonctions fléchées.

```
function sayHello() {
  console.log("Hello!");
}

const sayGoodbye = function() {
  console.log("Goodbye!");
};

const add = (x, y) => x + y;

sayHello(); // Affiche "Hello!"
sayGoodbye(); // Affiche "Goodbye!"
console.log(add(5, 3)); // 8
```

Manipulation d'Objets

```
const person = {
  firstName: "John",
  lastName: "Doe",
  greet: function() {
    console.log("Hello, " + this.firstName);
  }
};

console.log(person.firstName); // John
person.lastName = "Smith";
person.greet(); // Hello, John
```

Les objets en JavaScript sont des collections de paires clé/valeur. La notation littérale permet de créer des objets, et on accède ou modifie leurs propriétés à l'aide de la notation pointée ou des crochets. Les méthodes d'objet sont des fonctions associées à des objets.

Exploiter les Tableaux

Les tableaux en JavaScript peuvent être manipulés et itérés à l'aide de méthodes telles que .map(), .filter(), .reduce(), et .forEach(). Ces méthodes permettent de traiter les éléments d'un tableau de manière efficace et concise.

```
const numbers = [1, 2, 3, 4, 5];

const squared = numbers.map(x => x * x);
console.log(squared); // [1, 4, 9, 16, 25]

const even = numbers.filter(x => x % 2 === 0);
console.log(even); // [2, 4]
```

```
numbers.forEach(x => console.log(x));
```

Fonctions Fléchées (=>)

Les fonctions fléchées offrent une syntaxe plus concise pour écrire des fonctions en JavaScript, permettant d'écrire des expressions de fonction plus courtes et directes. Elles sont particulièrement utiles pour les fonctions anonymes et les callbacks.

```
const numbers = [1, 2, 3, 4, 5];
// Exemple avec map utilisant une fonction fléchée
const squared = numbers.map(x => x * x);
console.log(squared); // Affiche [1, 4, 9, 16, 25]

// Fonction fléchée sans argument
const sayHello = () => console.log("Hello!");
sayHello();

// Fonction fléchée avec plusieurs arguments
const add = (a, b) => a + b;
console.log(add(5, 7)); // Affiche 12

// Fonction fléchée avec corps étendu
const multiply = (a, b) => {
  const result = a * b;
  return result;
};
console.log(multiply(2, 3)); // Affiche 6
```

Modularité avec les modules ES6

```
// file: math.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// file: app.js
import { add, subtract } from './math.js';
console.log(add(2, 3)); // 5
console.log(subtract(5, 2)); // 3
```

Les modules permettent une meilleure organisation du code en séparant les fonctionnalités en différents fichiers, favorisant la réutilisation et la maintenance.

Promesses et Async/Await

```
// Function to perform a GET request using Fetch and return a promise
function fetchExample(url) {
  return fetch(url)
    .then(response => {
      // Check if the response is OK (status 200)
      if (response.ok) {
        // Return the response in JSON format
        return response.json();
      } else {
        // Throw an error with an appropriate message
        throw new Error(`Error ${response.status}: ${response.statusText}`);
      }
    })
    .then(data => {
      // Resolve the promise with the retrieved data
      return data;
    })
    .catch(error => {
      // Reject the promise with the encountered error
      throw error;
    });
}

// Example of using the fetchExample function
const url = 'https://api.example.com/data';

fetchExample(url)
  .then(data => {
    console.log('Data retrieved successfully:', data);
    // Do something with the retrieved data
  })
  .catch(error => {
    console.error('An error occurred while retrieving data:', error);
    // Handle the error appropriately
  });
```

Les promesses et la syntaxe async/await simplifient l'écriture de code asynchrone, rendant le traitement des opérations asynchrones plus lisible et facile à gérer.

Déstructuration pour une Meilleure Lisibilité

La destructuration permet d'extraire facilement des données des objets et des tableaux, rendant le code plus propre et plus lisible.

```
const person = { name: 'John', age: 30 };
const { name, age } = person;
console.log(name); // John
console.log(age); // 30

const numbers = [1, 2, 3];
const [first, , third] = numbers;
console.log(first); // 1
console.log(third); // 3
```

Paramètres par Défaut, Rest et Spread

Les valeurs par défaut pour les fonctions, l'opérateur Rest pour regrouper les arguments, et l'opérateur Spread pour étaler/expanser des éléments offrent plus de flexibilité dans la gestion des données.

```
function greet(name, greeting = "Hello") {
  console.log(`${greeting}, ${name}!`);
}

greet('John'); // Hello, John!
greet('John', 'Good morning'); // Good morning, John!

const parts = ['shoulders', 'knees'];
const body = ['head', ...parts, 'toes'];
console.log(body); // ["head", "shoulders", "knees", "toes"]
```

Template Literals

Les template literals permettent de construire des chaînes de caractères interpolés avec des expressions, rendant le code plus expressif et facilitant la création de chaînes multilignes.

```
const name = "John";
const greeting = `Hello, ${name}!
How are you today?`;
console.log(greeting);
```

Nouvelles Méthodes pour Objets et Tableaux

```
const numbers = [1, 2, 3, 4, 5];
console.log(numbers.find(x => x > 3)); // 4
console.log(numbers.includes(2)); // true

const person = { name: 'John', age: 30 };
console.log(Object.keys(person)); // ["name", "age"]
console.log(Object.values(person)); // ["John", 30]
```

ES6 et les versions ultérieures ont introduit de nouvelles méthodes pour travailler avec les objets et les tableaux, facilitant la recherche, la transformation, et la vérification des données.

L'Importance de l'Immutabilité

```
// Mauvaise pratique : modification directe d'un objet
let livre = { titre: "Programmation JavaScript" };
livre.titre = "Programmation Avancée"; // Modification directe

// Bonne pratique : immutabilité
const livreImmutable = Object.freeze({ titre: "Programmation JavaScript" });
const nouveauLivre = { ...livreImmutable, titre: "Programmation Avancée" };
```

L'immutabilité est un principe fondamental qui consiste à ne pas modifier directement les données. Au lieu de cela, toute modification produit une nouvelle instance des données. Cela aide à éviter les effets de bord et rend le code plus prévisible et facile à débugger.

Principes de la Programmation Fonctionnelle

La programmation fonctionnelle est un paradigme de programmation qui traite les calculs comme l'évaluation de fonctions mathématiques et évite les données changeantes ou mutables. Les principes clés incluent l'utilisation de fonctions pures et la composition de fonctions pour construire des logiciels.

```
// Fonction pure
const ajouter = (x, y) => x + y;

// Composition de fonctions
const multiplierParDeux = x => x * 2;
const ajouterEtMultiplier = (x, y) => multiplierParDeux(ajouter(x, y));

console.log(ajouterEtMultiplier(2, 3)); // 10
```

Stratégies de Gestion des Erreurs

La gestion des erreurs est cruciale pour développer des applications robustes. L'utilisation de blocs try / catch permet de gérer les exceptions de manière élégante, tandis que les promesses offrent un mécanisme pour traiter les erreurs dans les opérations asynchrones.

```
// Utilisation de try / catch
try {
  const resultat = operationRisquee();
  console.log(resultat);
} catch (erreur) {
  console.error("Une erreur s'est produite:", erreur);
}

// Gestion des erreurs dans les promesses
fetch("https://api.exemple.com/data")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(erreur => console.error("Erreur lors de la récupération des données:"
```

Maîtriser la Gestion des Paquets

```
// package.json avec un script personnalisé
{
  "name": "mon-projet",
  "version": "1.0.0",
  "scripts": {
    "start": "webpack --mode development && node server.js",
    "build": "webpack --mode production"
  },
  "dependencies": {
    "react": "^17.0.1"
  },
  "devDependencies": {
    "webpack": "^5.22.0",
    "babel-loader": "^8.2.2"
  }
}
```

NPM et Yarn sont des gestionnaires de paquets pour JavaScript qui simplifient l'installation, la mise à jour et la gestion des dépendances de projets. Ils permettent également de définir et d'exécuter des scripts personnalisés pour automatiser les tâches de développement courantes.

Linting et Formatage avec ESLint et Prettier

```
// .eslintrc.json
{
  "extends": "eslint:recommended",
  "rules": {
    "no-unused-vars": "warn",
    "eqeqeq": ["error", "always"]
  }
}
```

```
// .prettierrc
{
  "semi": false,
  "singleQuote": true
}
```

ESLint et Prettier sont des outils essentiels pour maintenir la qualité du code JavaScript. ESLint analyse le code pour détecter les erreurs et les problèmes de style, tandis que Prettier reformate automatiquement le code selon des règles définies, assurant une cohérence stylistique.

Introduction à Vite

Vite est un outil de build moderne conçu pour les projets JavaScript, TypeScript, et les frameworks tels que Vue, React, et Svelte. Il offre un démarrage rapide de serveur de développement et des rechargements à chaud ultra-rapides en tirant parti de l'importation de modules ES natifs dans les navigateurs et en pré-bundling des dépendances avec esbuild.

```
# Installation de Vite pour un nouveau projet React
npm create vite@latest mon-projet-react -- --template react

# Démarrage du projet
cd mon-projet-react
npm install
npm run dev
```

Configuration de Vite

```
// vite.config.js
import reactRefresh from '@vitejs/plugin-react-refresh';

export default {
  plugins: [reactRefresh()],
  resolve: {
    alias: {
      '@': '/src',
    },
  },
  build: {
    outDir: 'build',
  },
};
```

Vite est hautement configurable via son fichier vite.config.js. Vous pouvez ajuster la résolution des modules, les plugins, les options CSS, et bien plus, permettant une personnalisation profonde pour répondre aux besoins spécifiques de votre projet.

Optimisations et Déploiement avec Vite

Vite offre des fonctionnalités d'optimisation prêtes à l'emploi pour le déploiement de production, telles que la minification, le découpage de code (code splitting), et le préchargement des modules. Ces optimisations améliorent la performance de chargement de l'application en production.

```
# Commande pour construire un projet pour la production
npm run build

# Analyse du bundle pour optimisation
npm run preview
```

Introduction aux classes en JavaScript

Les classes en JavaScript introduites avec ES6 sont des syntaxes modernes pour définir des objets et gérer leur héritage. Elles simplifient la création et la gestion de modèles d'objets.

```
// Exemple simple de classe
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    greet() {
        return `Hi, my name is ${this.name} and I am ${this.age} years old.`;
    }
}

const john = new Person('John', 30);
console.log(john.greet());
```

Constructeur et Propriétés

Le constructor est une méthode spéciale utilisée pour initialiser les propriétés d'une classe lors de l'instanciation.

```
class Car {
    constructor(brand, model) {
        this.brand = brand;
        this.model = model;
    }
}

const myCar = new Car('Toyota', 'Corolla');
console.log(myCar.brand); // Toyota
console.log(myCar.model); // Corolla
```

Méthodes

Les méthodes définies dans une classe représentent les actions ou comportements des objets.

```
class Rectangle {
    constructor(width, height) {
        this.width = width;
        this.height = height;
    }

    area() {
        return this.width * this.height;
    }
}

const rect = new Rectangle(10, 5);
console.log(rect.area()); // 50
```

Méthodes statiques

Les méthodes statiques sont appelées sur la classe directement, pas sur une instance. Elles sont définies avec le mot-clé static.

```
class MathUtils {
    static add(a, b) {
        return a + b;
    }
}

console.log(MathUtils.add(5, 10)); // 15
```

Getters et Setters

Les getters et setters permettent de définir des comportements personnalisés pour lire ou écrire les propriétés d'une classe.

```
class Circle {
    constructor(radius) {
        this.radius = radius;
    }

    get diameter() {
        return this.radius * 2;
    }

    set diameter(d) {
        this.radius = d / 2;
    }
}

const circle = new Circle(10);
console.log(circle.diameter); // 20
circle.diameter = 30;
console.log(circle.radius); // 15
```

Héritage

Une classe peut hériter d'une autre classe avec le mot-clé extends, permettant de réutiliser et étendre les fonctionnalités,

```
class Animal {
    constructor(name) {
        this.name = name;
    }

    speak() {
        return `${this.name} makes a noise.`;
    }
}

class Dog extends Animal {
    speak() {
        return `${this.name} barks.`;
    }
}

const dog = new Dog('Rex');
console.log(dog.speak()); // Rex barks.
```

Appel au constructeur parent avec super()

Le mot-clé super() permet d'appeler le constructeur de la classe parente depuis une classe enfant.

```
class Vehicle {
    constructor(type) {
        this.type = type;
    }
}

class Bike extends Vehicle {
    constructor(type, brand) {
        super(type); // Appelle le constructeur de Vehicle
        this.brand = brand;
    }
}

const myBike = new Bike('Motorbike', 'Yamaha');
console.log(myBike.type); // Motorbike
console.log(myBike.brand); // Yamaha
```

Propriétés privées

Depuis ES2022, vous pouvez utiliser # pour définir des propriétés privées accessibles uniquement dans la classe.

```
class Account {
    #balance;

    constructor(initialBalance) {
        this.#balance = initialBalance;
    }

    getBalance() {
        return this.#balance;
    }
}

const account = new Account(1000);
console.log(account.getBalance()); // 1000
// console.log(account.#balance); // Erreur : propriété privée
```

Polymorphisme

Le polymorphisme permet à une méthode dans une classe enfant de remplacer une méthode de la classe parente.

```
class Shape {
    area() {
        return 'Area not defined';
    }
}

class Square extends Shape {
    constructor(side) {
        super();
        this.side = side;
    }

    area() {
        return this.side * this.side;
    }
}

const square = new Square(5);
console.log(square.area()); // 25
```

Composition avec des classes

La composition consiste à combiner plusieurs classes pour créer des objets complexes.

```
class Engine {
    start() {
        return 'Engine started';
    }
}

class Wheels {
    roll() {
        return 'Wheels rolling';
    }
}

class Car {
    constructor() {
        this.engine = new Engine();
        this.wheels = new Wheels();
    }

    drive() {
        return `${this.engine.start()} and ${this.wheels.roll()}`;
    }
}

const car = new Car();
console.log(car.drive()); // Engine started and Wheels rolling
```

01.1

Modules

Javascript

Les Modules en JavaScript

Un module est un morceau de code JavaScript qui exporte certaines fonctionnalités afin qu'elles puissent être réutilisées dans d'autres parties de l'application.

Avant ES6, il n'y avait pas de système de module natif dans JavaScript. CommonJS et AMD étaient populaires.

Introduction à CommonJS

CommonJS est un standard de module utilisé principalement dans Node.js.

```
// fichier math.js
module.exports = {
  add: function(a, b) { return a + b; },
  subtract: function(a, b) { return a - b; }
};

// fichier app.js
const math = require('./math');
console.log(math.add(2, 3)); // 5
```

Exports dans CommonJS

Utilisation de module.exports pour exporter des fonctions et des objets.

```
// fichier math.js
module.exports.add = function(a, b) { return a + b; };
module.exports.subtract = function(a, b) { return a - b; };
```

Require dans CommonJS

Utilisation de require pour importer des modules dans CommonJS.

```
// fichier app.js
const add = require('./math').add;
console.log(add(2, 3)); // 5
```

Introduction à AMD

AMD est une spécification pour définir des modules qui peuvent être chargés de manière asynchrone. Utilisé principalement dans les applications côté client.

Définir un Module AMD

Utilisation de define pour créer un module AMD.

```
// fichier math.js
define([], function() {
  return {
    add: function(a, b) { return a + b; },
    subtract: function(a, b) { return a - b; }
  };
});
```

Charger un Module AMD

Utilisation de require pour charger des modules AMD.

```
// fichier app.js
require(['math'], function(math) {
  console.log(math.add(2, 3)); // 5
});
```

Modules Dépendants avec AMD

Définition de modules avec des dépendances.

```
// fichier calculator.js
define(['math'], function(math) {
  return {
    calculate: function(a, b) { return math.add(a, b); }
  };
});
```

Introduction à ES6 Modules

ES6 a introduit un système de module natif dans JavaScript.

```
// fichier math.js
export function add(a, b) { return a + b; }
export function subtract(a, b) { return a - b; }

// fichier app.js
import { add, subtract } from './math.js';
console.log(add(2, 3)); // 5
```

Exportations Nomées

Utilisation de export pour exporter plusieurs éléments d'un module.

```
// fichier utils.js
export const PI = 3.14;
export function circumference(r) { return 2 * PI * r; }
```

Exportations par Défaut

Utilisation de export default pour exporter un seul élément par défaut.

```
// fichier math.js
export default function add(a, b) { return a + b; }

// fichier app.js
import add from './math.js';
console.log(add(2, 3)); // 5
```

Importations

Utilisation de import pour importer des modules ou des éléments spécifiques.

```
// fichier app.js
import { add, subtract } from './math.js';
```

Renommer les Importations

Utilisation de as pour renommer des importations.

```
// fichier app.js
import { add as addition, subtract as soustraction } from './math.js';
```

02

TypeScript

TypeScript, c'est quoi ?

TypeScript est un sur-ensemble de JavaScript développé par Microsoft, ayant pour but d'ajouter un système de typage statique. Cela nous permettra d'écrire un code plus sécurisé, plus facilement maintenable, et de détecter des erreurs avant même l'exécution du code.

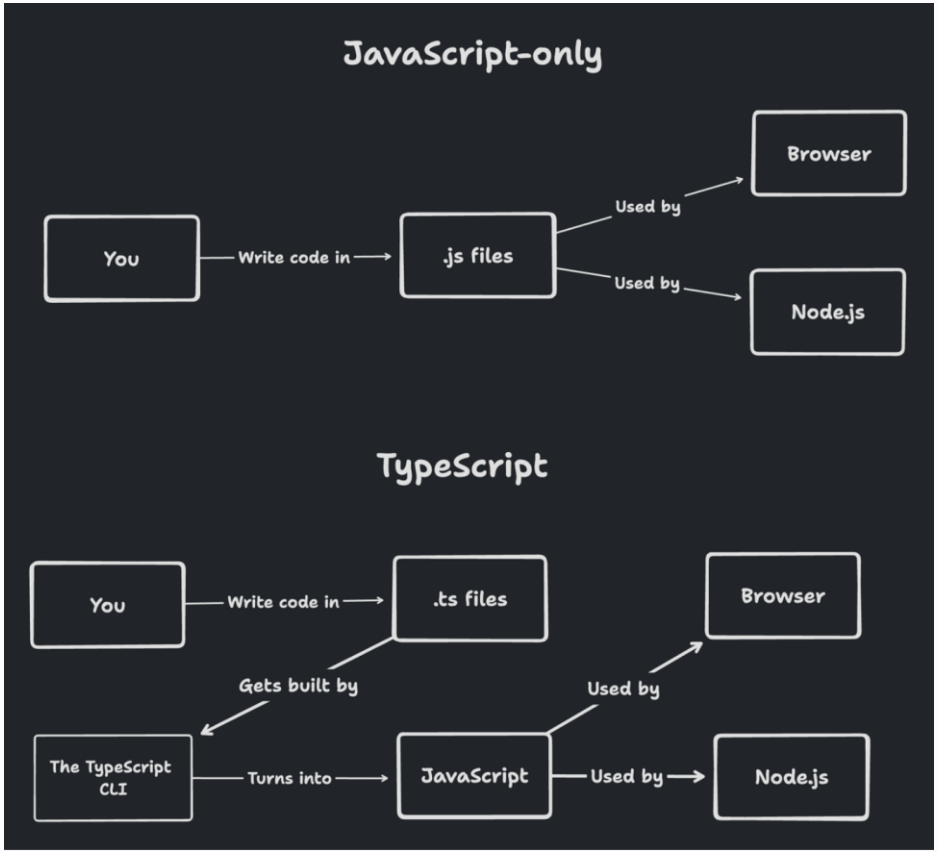
Le choix d'utiliser ou non TypeScript dépend de vos préférences en termes de sécurité et de flexibilité.

Les fichiers TypeScript auront une extension en .ts ou .tsx

Les avantages de TypeScript

1. Syntaxe proche du JavaScript
2. Permet de détecter des erreurs potentielles, même sans déclarer de type
3. Facilite la documentation du code
4. Permet à l'IDE de gérer les suggestion de code plus facilement
5. Meilleure maintenabilité
6. L'inférence de type

Le fonctionnement de TypeScript



La syntaxe TypeScript

Comme noté précédemment, nos fichiers auront une extension .ts ou .tsx.
Pour définir une variable avec un type, la syntaxe ressemblera à :

Déclaration (let ou const) variable : type = valeur

```
let person : string = "Jean";
let age : number = 30;
let isMajor : boolean = true;

person = age; //Impossible d'assigner le type number à une variable de type string
isMajor = isMajor + person; //Impossible d'additionner une variable de type boolean avec une variable de type string
```

Les types primitifs

TypeScript nous fournit 3 types primitifs, et 2 types utilitaires:

1. number
2. string
3. boolean
4. unknown (quand on ne sait pas quel type nous allons recevoir, on y reviendra après)
5. any (à éviter)

La syntaxe TypeScript

Comme noté précédemment, nos fichiers auront une extension .ts ou .tsx.
Pour définir une variable avec un type, la syntaxe ressemblera à :

Déclaration (let ou const) variable : type = valeur

```
let person : string = "Jean";
let age : number = 30;
let isMajor : boolean = true;
function sayHello(name : string) : string {
    return "Hello " + name;
}
person = age; //Impossible d'assigner le type number à une variable de type string
isMajor = isMajor + person; //Impossible d'additionner une variable de type boolean avec une variable de type string

sayHello(isMajor) // string attendu en paramètre, mais on passe un boolean
```

L'inférence de type

C'est sympa de pouvoir déclarer des types primitifs sur les variables lorsqu'on les déclare, mais ça va vite devenir très répétitif non ?
C'est là que l'inférence de type entre en scène: TS va être capable de « déduire » le type de certaines variables et signatures de fonction via leur valeurs d 'origine. Pas besoin de tout typer explicitement !

```
let person = "Jean";
let age = 30;
let isMajor = true;
function sayHello(name : string)  {
    return "Hello " + name;
}
person = age; //Même erreur d'auparavant
isMajor = isMajor + person; //Idem

sayHello(isMajor) // Toujours pareil
```

L'inférence de type

Attention, TS n'est pas omniscient, et ne sera pas toujours capable d'inférer les types correctement.
Si il ne sait pas, il utilisera le type any, qui ne procède à aucune vérification !
Par exemple :

```
const myArray = []; //Type implicite: any[]
myArray.push(1) //Pas d'erreur car non strict
myArray.push('Hello') //Pas d'erreur car non strict, mais dangereux
```

Types non conformes à la réalité

TypeScript nous permet de détecter des erreurs dans le code, mais peut parfois autoriser des opérations qui généreraient des erreurs à l'exécution.

```
const names = ["Charles", "Pierrick"]
console.log(names[2].toLowerCase())
//TS ne signale pas d'erreur, mais la dernière ligne va générer une erreur à l'exécution
```

Type et Interface

TypeScript nous permet de créer nos propres types, grâce aux mots clés « type » et « interface ».

```
type Age = number;
interface Person {
    name: string;
    age: Age;
}
let driver: Person = {
    name: 'James May',
    age: 56
};
driver.age = 57; // OK
driver.age = '57'; // Error
```

Type ou Interface ?

Type et interface peuvent sembler interchangeables, car ils le sont majoritairement. Normalement, type est plutôt utilisé pour définir les alias, et interface pour typer des objets complets. À notre niveau, nous pouvons nous contenter d'utiliser type au général, car ils nous permettent de faire des « union types » qui nous seront utiles pour la suite.

```ts
type item = {
    name: string,
    price: number,
}

type weapon = item & {
    damage: number,
}
```

```ts
interface item {
    name: string;
    price: number;
}

interface weapon extends item {
    damage: number;
}
```

Union types et Narrowing

Avec le mot clé type, nous pouvons créer des « union types », c'est-à-dire des variables qui peuvent accepter plusieurs types, par exemple:

```
type StringOrNumber = string | number;

let myAge: StringOrNumber = 25;
myAge = '25'; // OK
```

Pour savoir quel est le type réel d'une variable avec une union, on peut utiliser l'inférence de TS :

```
const doSometing = (value: StringOrNumber) => {
    if (typeof value === 'string') {
        //TypeScript infère que c'est une string
        return value.toUpperCase();
    }
    //TypeScript infère que c'est un number
    return value;
}
```

Narrowing sur des objets ?

Pour faire du narrowing sur des types complexes, nous pouvons utiliser le mot clé « in » pour déterminer ou non l'existence d'une propriété propre à un des types.

```
type Fish = {
    swim : () => void;
}
type Bird = {
    fly : () => void;
}

function move(animal: Fish | Bird){
    if("swim" in animal){
        return animal.swim();
    }
    return animal.fly();
}
```

Les génériques

Les génériques sont une fonctionnalité de TS qui nous permettent de créer des types génériques réutilisables et de réduire le code à écrire. Prenons l'exemple ci-dessous:

```
type StringCollection = {
    name: string;
    items : string[];
}
type NumberCollection = {
    name: string;
    items : number[];
}
type BooleanCollection = {
    name: string;
    items : boolean[];
}
```

Les génériques

On peut créer un type générique Collection, qui prendra en « paramètre » un type T qui sera assigné à la propriété items. Nos types précédents seront donc déclarées par le générique.

```
type Collection<T> = {
    name: string;
    items : T[];
}
type StringCollection = Collection<string>;
type NumberCollection = Collection<number>;
type BooleanCollection = Collection<boolean>;
```

Les génériques

Il existe des génériques utilitaires, comme <Partial>, <Omit>, <Record>... que nous utiliserons peut être lors de la formation.

Vous pouvez également les retrouver sur la documentation TypeScript, juste ici: https://www.typescriptlang.org/docs/handbook/utility-types.html

Installation

Ajoutez TypeScript et ses dépendances nécessaires au développement

typescript : Le compilateur TypeScript.
ts-node : Permet d'exécuter des fichiers TypeScript directement sans les compiler au préalable.
@types/node : Définit les types pour Node.js.

```
npm install typescript ts-node @types/node --save-dev
```

```
npm install nodemon ts-node-dev --save-dev
```

Initialiser TypeScript

Créez un fichier de configuration tsconfig.json :

```
npx tsc --init
```

Configurer nodemon

Créez un fichier de configuration nodemon.json pour personnaliser le comportement de nodemon :
watch : Dossiers ou fichiers à surveiller (par exemple, src).
ext : Extensions de fichiers à surveiller (ts pour TypeScript).
ignore : Dossiers ou fichiers à ignorer (dist pour éviter de surveiller les fichiers compilés).
exec : La commande à exécuter (ts-node pour exécuter les fichiers TypeScript directement)

```
{
  "watch": ["src"],
  "ext": "ts",
  "ignore": ["dist"],
  "exec": "ts-node src/index.ts"
}
```

03

Comment fonctionne Node ?

Qu’est-ce que Node.js ?

Node.js est un environnement d’exécution JavaScript côté serveur, construit sur le moteur V8 de Google Chrome.

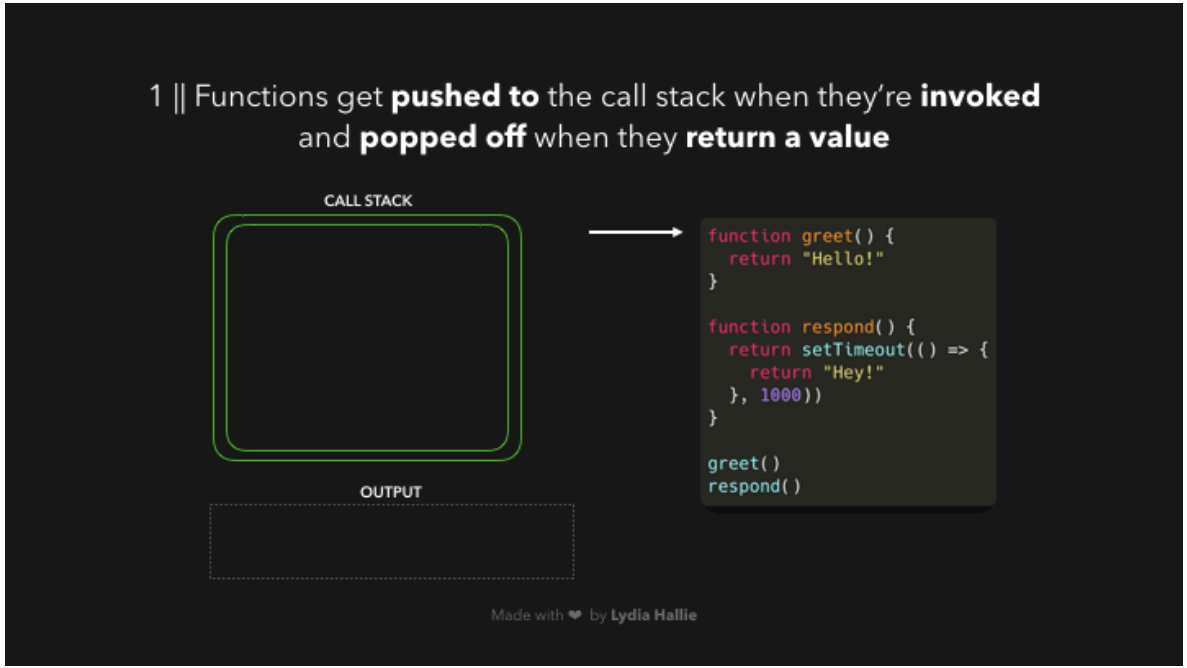
```
// Un simple serveur HTTP en Node.js
const http = require('http');

http.createServer((req, res) => {
  res.end('Bonjour, Node.js avancé !');
}).listen(3000, () => {
  console.log('Serveur lancé sur http://localhost:3000');
});
```

L’architecture interne de Node.js : V8, libuv et Event Loop

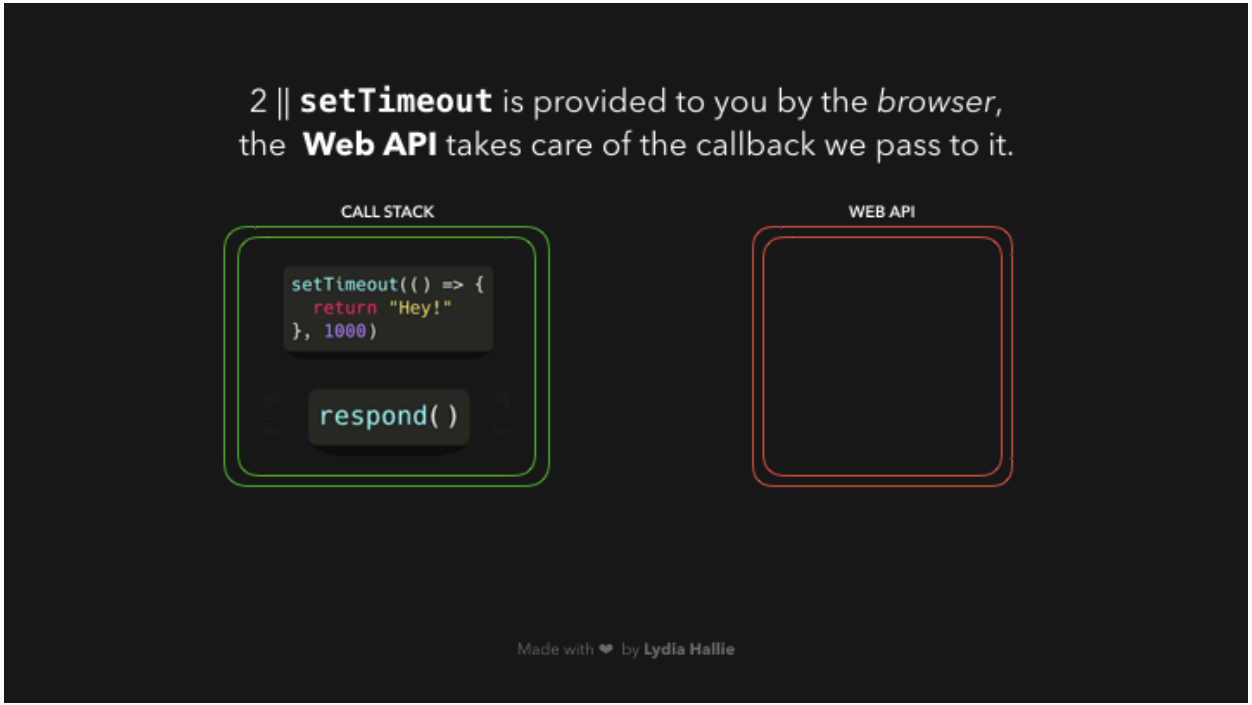
Node.js repose sur V8 (moteur JavaScript de Chrome) pour exécuter du JavaScript, et libuv pour gérer un modèle d’E/S non-bloquant (Event Loop). L’Event Loop est au cœur de Node.js et assure la gestion asynchrone des opérations (I/O, timers, etc.) sans bloquer le thread principal.

```
// Observer le fonctionnement asynchrone
console.log('Début');
setTimeout(() => console.log('Timeout déclenché'), 0);
console.log('Fin');
// Ordre d'affichage : Début, Fin, puis Timeout déclenché
```

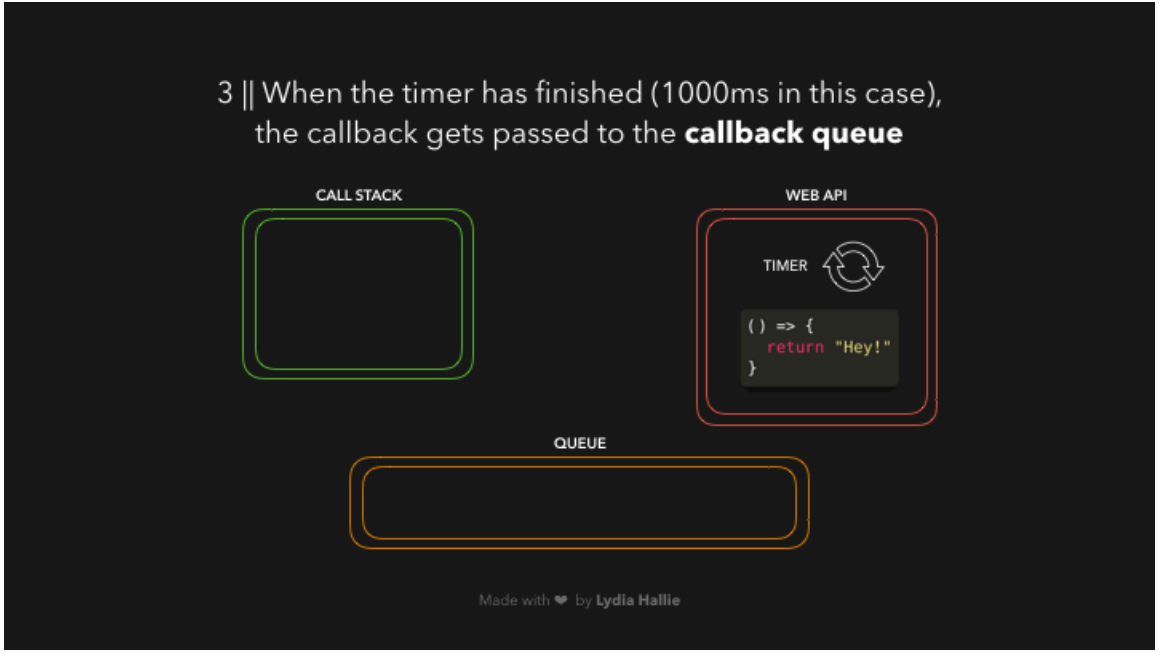
L’architecture interne de Node.js : V8, libuv et Event Loop



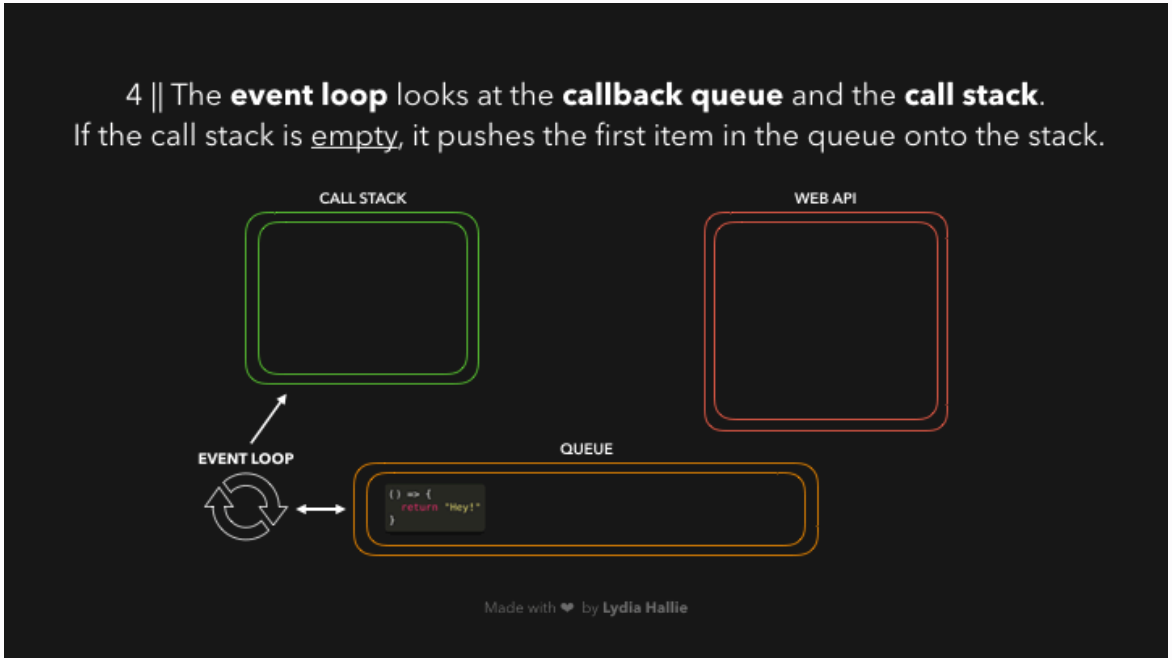
L’architecture interne de Node.js : V8, libuv et Event Loop



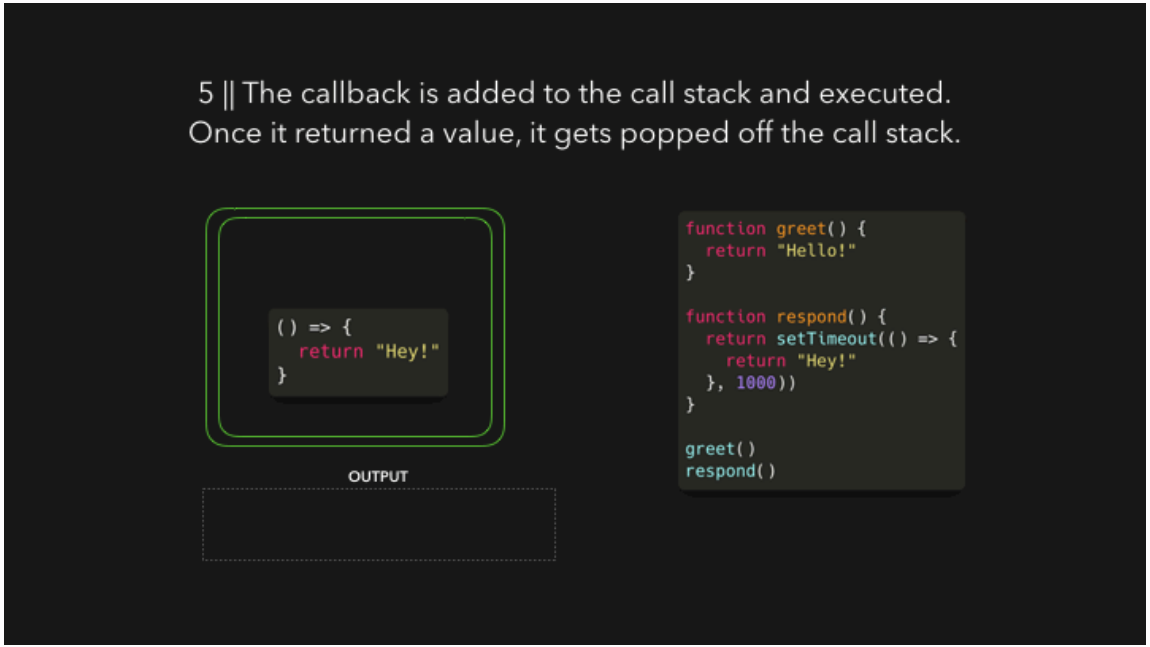
L’architecture interne de Node.js : V8, libuv et Event Loop



L’architecture interne de Node.js : V8, libuv et Event Loop



L’architecture interne de Node.js : V8, libuv et Event Loop



Comprendre la non-blocance du JavaScript côté serveur

Node.js utilise un unique thread d'exécution, mais grâce à l'Event Loop, il peut traiter des milliers de connexions simultanées sans bloquer. Au lieu d'attendre la fin d'une opération I/O (lecture fichier, requête réseau), Node.js délègue la tâche et continue le traitement d'autres tâches.

```
// Lecture de fichier non bloquante
const fs = require('fs');

fs.readFile('grande_fichier.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log('Fichier lu !');
});

console.log('Je continue avant la fin de la lecture du fichier...');
```

Gestion de la mémoire et Garbage Collection en pratique

Le moteur V8 gère automatiquement la mémoire par un système de Garbage Collection. Toutefois, il est important de limiter les fuites mémoire en refermant les ressources (streams, sockets) et en évitant les références inutiles. Des outils comme --inspect aident à analyser l'utilisation mémoire.

```
// Exemple simplifié de fuite mémoire potentielle
let bigArray = [];
for (let i = 0; i < 1e6; i++) {
  bigArray.push({data: 'some data'});
}
// Sans réinitialiser ou libérer les références, bigArray occupe la mémoire.
bigArray = null; // Libération de la référence, le GC pourra récupérer la mémoire.
```

04

Gestion de packages

Gestion des modules et dépendances avec npm

npm (Node Package Manager) est l'outil standard pour gérer les dépendances. Il permet d'installer, mettre à jour et supprimer des modules. Le fichier package.json répertorie ces dépendances, leurs versions et leurs scripts associés.

```
# Installer une dépendance
npm install express

# Lancer un script défini dans package.json
npm run start
```

Découverte approfondie du fichier package.json

Le fichier package.json décrit votre projet : nom, version, description, scripts, dépendances, licence, etc. Il sert d'entrée centrale pour npm et d'autres outils. Bien le comprendre facilite la gestion du cycle de vie de votre application.

```json
{
  "name": "mon-projet",
  "version": "1.0.0",
  "description": "Un projet Node.js avancé",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "jest"
  },
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "jest": "^29.0.0"
  }
}
```

Modules CommonJS vs ES Modules: différences et coexistence

Node.js prend en charge deux formats de modules : CommonJS (require/exports) et ES Modules (import/export). CommonJS est historiquement le standard, tandis que les ES Modules (ECMAScript) sont le futur standard officiel. Les deux peuvent coexister, bien que l'utilisation d'ES Modules nécessite souvent l'extension .mjs ou le champ "type": "module" dans package.json.

```
// CommonJS
const fs = require('fs');
module.exports = { fs };

// ES Modules
import { readFile } from 'fs/promises';
export const lire = (chemin) => readFile(chemin, 'utf8');
```

Utilisation avancée de npm : scripts, tags et versionning sémantique

npm permet de créer des scripts personnalisés (build, test, lint), de publier des paquets avec des tags (beta, next) et de gérer des versions selon le versionnage sémantique (MAJEUR.MINOR.PATCH). Ceci assure un déploiement et une maintenance plus structurés.

```
# Scripts npm dans package.json
npm run build

# Publier avec un tag
npm publish --tag beta

# Mettre à jour une dépendance mineure
npm install express@^4.19.0
```

Répertoires node_modules : résolution de dépendances en profondeur

Le répertoire node_modules contient toutes les dépendances installées. npm résout les dépendances de manière hiérarchique, installant les modules et leurs dépendances. Cela peut entraîner des arborescences complexes, mais npm 3+ et 7+ ont optimisé ce processus pour réduire la duplication.

```
# Explorer node_modules
ls node_modules

# Vérifier l'arborescence des dépendances
npm list
```

Gestion fine des dépendances : devDependencies, peerDependencies, optionalDependencies

devDependencies : utilisées uniquement en développement (tests, outils de build).
peerDependencies : indiquent des dépendances que l'utilisateur final doit fournir.
optionalDependencies : dépendances non essentielles, si elles échouent à l'installation, le reste fonctionne.

```
{
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "jest": "^29.0.0"
  },
  "peerDependencies": {
    "react": "^18.0.0"
  },
  "optionalDependencies": {
    "chokidar": "^3.5.0"
  }
}
```

04.5

Les packages utiles

Introduction au linting avec ESLint

ESLint est un outil qui analyse votre code pour détecter les problèmes de syntaxe, de style ou d'erreurs potentielles. Il aide à maintenir un code propre et uniforme.

```
npm install --save-dev eslint
npx eslint --init
```

Formatage de code automatique avec Prettier

Prettier formate automatiquement votre code selon des conventions prédéfinies (indentation, guillemets, etc.). Couplé à ESLint, il garantit un code lisible et homogène.

```
npm install --save-dev prettier
npx prettier --write .
```

Commentaires et documentation interne du code

Les commentaires clarifient l'intention du code. Une documentation interne (JSDoc) rend l'API du code plus compréhensible. C'est essentiel pour le travail en équipe et la maintenance à long terme.

```
/**
 * Ajoute deux nombres.
 * @param {number} a
 * @param {number} b
 * @return {number}
 */
function add(a, b) {
  return a + b;
}
```

05

Asynchronisme dans Node

Les callbacks traditionnels : avantages, inconvénients, patterns

Les callbacks fournissent un style asynchrone simple, mais peuvent mener à des pyramides de “callback hell” rendant le code difficile à lire et à maintenir. L’utilisation de fonctions nommées, de modules utilitaires ou la transition vers Promises allègent ce problème.

```
// Exemple de callback hell
fs.readFile('fichier.txt', 'utf8', (err, data) => {
  if (err) return console.error(err);
  fs.writeFile('autre.txt', data, (err) => {
    if (err) return console.error(err);
    console.log('Copie terminée !');
  });
});
```

Les Promises : chaîne d'opérations asynchrones et error handling

Les Promises simplifient l'écriture de code asynchrone en offrant une API plus lisible et un mécanisme intégré de gestion d'erreurs. Elles permettent de chaîner plusieurs opérations de manière fluide.

```
const { readFile, writeFile } = require('fs/promises');

readFile('fichier.txt', 'utf8')
  .then(data => writeFile('autre.txt', data))
  .then(() => console.log('Copie terminée !'))
  .catch(console.error);
```

Async/Await : simplification de l'écriture asynchrone

Async/Await est une syntaxe plus claire pour gérer les Promises. Elle permet d'écrire du code asynchrone qui ressemble à du code synchrone, tout en profitant des mêmes avantages (chaînage, gestion d'erreurs).

```
const { readFile, writeFile } = require('fs/promises');

async function copierFichier() {
  try {
    const data = await readFile('fichier.txt', 'utf8');
    await writeFile('autre.txt', data);
    console.log('Copie terminée !');
  } catch (err) {
    console.error(err);
  }
}

copierFichier();
```

Gestion avancée des erreurs asynchrones (try/catch, .catch, rejections)

L'utilisation de try/catch dans les fonctions async, ou le chaînage de .catch() sur les Promises, permet de gérer proprement les erreurs. Il est également important de gérer les rejets non gérés (unhandled rejections) pour éviter des comportements imprévisibles.

```
process.on('unhandledRejection', (reason) => {
  console.error('Rejet non géré:', reason);
});

async function riskyOperation() {
  throw new Error('Oups!');
}

riskyOperation().catch(console.error);
```

Timers, setImmediate et process.nextTick : quand les utiliser ?

setTimeout et setInterval planifient des callbacks dans la phase “timers”.
setImmediate exécute un callback à la fin de la phase de poll.process.
nextTick exécute un callback avant la prochaine itération de l’Event Loop.
Chacun a son utilité selon la priorité des opérations asynchrones.

```
setImmediate(() => console.log('setImmediate'));
process.nextTick(() => console.log('nextTick'));
setTimeout(() => console.log('setTimeout'), 0);

console.log('Synchrones');
```

Cas d'utilisation pratiques

- Choisir la bonne méthode selon la priorité
- nextTick : Débogage, initialisation critique.
- setImmediate : Déclenchements secondaires.
- setTimeout/setInterval : Programmation temporisée.

```
process.nextTick(() => {
  console.log('High priority');
});

setImmediate(() => {
  console.log('Lower priority');
});

setTimeout(() => {
  console.log('Timer callback');
}, 0);
```

06

La gestion du file System

Le module fs (synchrones vs asynchrones) : bonnes pratiques

Le module fs propose des méthodes synchrones (bloquantes) et asynchrones (non bloquantes). Il est préférable d'utiliser les versions asynchrones pour préserver les performances et éviter de bloquer l'Event Loop.

```
// Méthode asynchrone recommandée
fs.readFile('fichier.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});

// Méthode synchrone (déconseillée)
const contenu = fs.readFileSync('fichier.txt', 'utf8');
console.log(contenu);
```

Lire et écrire des fichiers volumineux avec les Streams de Node.js

Les Streams permettent de traiter les données par petits morceaux plutôt que de tout charger en mémoire. Ceci est idéal pour les fichiers volumineux, permettant un traitement plus efficace et une réduction de l'empreinte mémoire.

```
const fs = require('fs');

const lecture = fs.createReadStream('grosFichier.txt');
const ecriture = fs.createWriteStream('copie.txt');

lecture.pipe(ecriture).on('finish', () => {
  console.log('Copie terminée via Stream !');
});
```

Modification, suppression et renommage de fichiers

Node.js offre des méthodes pour renommer, supprimer et modifier des fichiers. Attention aux erreurs (fichiers inexistants, permissions) et gérez-les proprement.

```
const fs = require('fs');

// Renommer un fichier
fs.rename('ancien.txt', 'nouveau.txt', (err) => {
  if (err) throw err;
  console.log('Fichier renommé !');
});

// Supprimer un fichier
fs.unlink('nouveau.txt', (err) => {
  if (err) throw err;
  console.log('Fichier supprimé !');
});
```

Gestion des répertoires : création, suppression et navigation récursive

Vous pouvez créer, supprimer et parcourir des répertoires. Les fonctions asynchrones comme fs.mkdir (avec l'option recursive) ou fs.readdir permettent de gérer facilement la structure des dossiers.

```
fs.mkdir('monDossier', { recursive: true }, (err) => {
  if (err) throw err;
  console.log('Dossier créé');
});

fs.readdir('monDossier', (err, fichiers) => {
  if (err) throw err;
  console.log('Fichiers dans monDossier :', fichiers);
});
```

Attributs des fichiers et métadonnées (stats, chmod, etc.)

Node.js permet d'obtenir des informations sur les fichiers (taille, date de modification) via fs.stat, et de modifier leurs permissions via fs.chmod. Cela permet de gérer précisément l'accès et la maintenance des fichiers.

```
fs.stat('fichier.txt', (err, stats) => {
  if (err) throw err;
  console.log(`Taille : ${stats.size} octets`);
  console.log(`Dernière modification : ${stats.mtime}`);
});

fs.chmod('fichier.txt', 0o644, (err) => {
  if (err) throw err;
  console.log('Permissions modifiées !');
});
```

Observation des changements de fichiers (fs.watch, chokidar)

fs.watch permet de surveiller des changements dans un répertoire ou un fichier. Pour une surveillance plus fiable et cross-plateform, des bibliothèques comme chokidar sont recommandées.

```
const fs = require('fs');

fs.watch('monDossier', (eventType, filename) => {
  console.log(`Changement détecté: ${eventType} sur ${filename}`);
});

// Avec chokidar (installation préalable: npm install chokidar)
const chokidar = require('chokidar');
chokidar.watch('monDossier').on('all', (event, path) => {
  console.log(event, path);
});
```

Le module path : joindre, normaliser, résoudre des chemins

Le module path facilite la manipulation des chemins de fichiers et de répertoires, indépendamment du système d'exploitation. Vous pouvez joindre, normaliser ou résoudre des chemins pour éviter les erreurs.

```
const path = require('path');

const chemin = path.join(__dirname, 'monDossier', 'fichier.txt');
console.log('Chemin absolu :', chemin);

const normalise = path.normalize('monDossier/./sousDossier/../fichier.txt');
console.log('Chemin normalisé :', normalise);
```

Chemins absolus, relatifs et liens symboliques : scénarios pratiques

Un chemin absolu débute à la racine du système, tandis qu'un chemin relatif dépend de la position actuelle. Les liens symboliques permettent de pointer vers d'autres fichiers ou dossiers, facilitant l'organisation et le partage de ressources.

```
# Créer un lien symbolique sur Unix/Linux/Mac
ln -s /chemin/absolu/fichier.txt lien_symbolique.txt

# En Node.js, résolvez un chemin absolu à partir d'un relatif
const path = require('path');
const absolu = path.resolve('monDossier/fichier.txt');
console.log(absolu);
```

Intégration avec URL et manipulation d'URIs locales

Le module url peut aider à analyser et formater des URL. Combiner path et URL est utile lorsque vous travaillez avec des ressources locales en lien avec des services web ou des chemins dynamiques.

```
const { URL } = require('url');
const path = require('path');

const monURL = new URL('file:///Users/nom/monDossier/fichier.txt');
console.log(monURL.pathname); // Chemin local

// Combiner avec path
const cheminLocal = path.join('/Users/nom/monDossier', 'fichier.txt');
console.log(cheminLocal);
```

Exercice !

Exercice : Créer un script qui organise un dossier "Downloads".
Écrire un script qui parcourt un dossier "Downloads" et déplace automatiquement les fichiers dans des sous-dossiers en fonction de leur extension (ex. : .jpg dans "Images", .pdf dans "Documents", etc.).
Ajouter une fonctionnalité qui crée les dossiers nécessaires s'ils n'existent pas déjà.

07

Le module HTTP

Le module http : créer un serveur minimaliste

Le module http permet de créer un serveur HTTP basique. Vous écoutez un port, traitez les requêtes entrantes et répondez avec du contenu. C'est la fondation pour construire des API, sites web et services REST.

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

const serveur = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Bonjour, monde!');
});

serveur.listen(3000, () => {
  console.log('Serveur HTTP démarré sur http://localhost:3000');
});
```

Analyse détaillée de l'objet request : méthodes, en-têtes, URL

L'objet req représente la requête entrante. Il contient la méthode HTTP (GET, POST...), les en-têtes, l'URL demandée, et un flux (stream) pour lire le corps de la requête. La compréhension de req est essentielle pour router et traiter les données reçues.

```
http.createServer((req, res) => {
  console.log('Méthode:', req.method);
  console.log('URL:', req.url);
  console.log('En-têtes:', req.headers);
  res.end('Ok');
}).listen(3000);
```

Réponse HTTP personnalisée : statuts, en-têtes, corps de réponse

Vous pouvez contrôler le code de statut (200, 404, 500...), les en-têtes (Content-Type, Cache-Control...) et le corps de la réponse. Cela permet de servir des pages HTML, JSON, fichiers binaires et de gérer proprement les erreurs.

```
http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'application/json');
  const data = { message: 'Hello!' };
  res.end(JSON.stringify(data));
}).listen(3000);
```

Gestion des requêtes entrantes : parsing du JSON, du form-data

Les données du corps de la requête arrivent sous forme de flux. Pour les analyser (JSON, form-data), vous pouvez accumuler les données, puis les parser (JSON.parse, librairies spécialisées). Cela vous permet de traiter des formulaires, des JSON d'API, etc.

```
http.createServer((req, res) => {
  let body = '';
  req.on('data', chunk => body += chunk);
  req.on('end', () => {
    const obj = JSON.parse(body);
    res.end(`Vous avez envoyé: ${obj.nom}`);
  });
}).listen(3000);
```

Routing manuel avec url et querystring

Vous pouvez analyser l'URL entrante et la chaîne de requête (querystring) pour mettre en place un système de routage basique. Cela permet de servir différents contenus selon le chemin demandé.

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

http.createServer((req, res) => {
  const parsedUrl = url.parse(req.url);
  const query = querystring.parse(parsedUrl.query);

  if (parsedUrl.pathname === '/hello') {
    res.end(`Hello, ${query.name || 'stranger'}!`);
  } else {
    res.statusCode = 404;
    res.end('Not Found');
  }
}).listen(3000);
```

Serving de fichiers statiques : mise en cache, compression, MIME types

En servant des fichiers (HTML, CSS, images), vous pouvez définir des en-têtes de cache, compresser les réponses (gzip, deflate) et définir les MIME types corrects. Cela améliore les performances et l'expérience utilisateur.

```
const fs = require('fs');
const path = require('path');
const mime = { '.html': 'text/html', '.js': 'application/javascript' };

http.createServer((req, res) => {
  const filePath = path.join(__dirname, req.url);
  fs.readFile(filePath, (err, data) => {
    if (err) {
      res.statusCode = 404;
      return res.end('Fichier non trouvé');
    }
    const ext = path.extname(filePath);
    res.setHeader('Content-Type', mime[ext] || 'application/octet-stream');
    res.end(data);
  });
}).listen(3000);
```

Utilisation des Streams pour la réponse HTTP (par exemple, gros fichiers)

Pour éviter de charger un gros fichier en mémoire, vous pouvez utiliser un Readable Stream et le pipe() vers la réponse. C'est plus efficace et réduit l'utilisation mémoire.

```
http.createServer((req, res) => {
  const stream = fs.createReadStream('video.mp4');
  res.writeHead(200, { 'Content-Type': 'video/mp4' });
  stream.pipe(res);
}).listen(3000);
```

Gestion des erreurs HTTP : réponses 404, 500, cas spécifiques

Gérez les erreurs de manière propre. Retournez 404 si la ressource n'est pas trouvée, 500 pour une erreur interne, etc. Cela améliore la clarté pour le client et facilite le debugging.

```
http.createServer((req, res) => {
  try {
    if (req.url === '/inconnu') {
      res.statusCode = 404;
      res.end('Page non trouvée');
    } else {
      throw new Error('Erreur interne');
    }
  } catch (e) {
    res.statusCode = 500;
    res.end('Erreur serveur');
  }
}).listen(3000);
```

Introduction au HTTPS : certificat, clé privée, secureContext

Le module https permet de créer un serveur chiffré. Vous avez besoin d'un certificat et d'une clé privée. Cela protège les données entre le client et le serveur (HTTPS). Le secureContext gère les paramètres SSL/TLS.

```
const https = require('https');
const fs = require('fs');

const options = {
  key: fs.readFileSync('key.pem'),
  cert: fs.readFileSync('cert.pem')
};

https.createServer(options, (req, res) => {
  res.end('Connexion sécurisée!');
}).listen(3443);
```

Exercice !

Exercice : Créer un serveur HTTP simple pour gérer un carnet d'adresses.
Créer un serveur qui écoute sur un port donné.
Ajouter des routes simples :GET /contacts : renvoie une liste d'adresses (bouchon de données)
POST /contacts : ajoute une nouvelle adresse à la liste. (bouchon)
DELETE /contacts/:id : supprime une adresse selon son ID. (bouchon)

08

Appels HTTP

axios / fetch / node-fetch

Pourquoi intégrer des services externes dans une application Node.js ?

L'intégration de services externes (API tierces, bases de données distantes, fournisseurs de paiement, services d'envoi d'e-mails) enrichit les fonctionnalités de votre application. Elle permet d'accéder à des données actualisées, de déléguer certaines responsabilités (authentification, traitement de paiement, etc.) et de gagner en efficacité.

```
// Exemple : appeler une API météo pour afficher la température
fetch('https://api.exemple.com/meteo?ville=Paris')
  .then(res => res.json())
  .then(data => console.log(`Température à Paris : ${data.temp}°C`))
  .catch(err => console.error(err));
```

Panorama des outils d'intégration et des options disponibles

Pour effectuer des appels HTTP, vous pouvez utiliser les modules natifs (http, https), ou des bibliothèques dédiées comme Axios, node-fetch, Superagent, ou encore Request (déconseillé car déprécié). Le choix dépend de la simplicité, des fonctionnalités (intercepteurs, gestion avancée des en-têtes, JSON automatique) et des performances.

```
// Avec le module https natif
const https = require('https');
https.get('https://api.exemple.com/data', res => {
  let data = '';
  res.on('data', chunk => data += chunk);
  res.on('end', () => console.log(JSON.parse(data)));
});
```

Présentation d'Axios : pourquoi et quand l'utiliser ?

Axios est une bibliothèque HTTP populaire, simple à utiliser, supportant les Promises et offrant une gestion automatique du JSON, des intercepteurs de requêtes/réponses, et une API cohérente côté client et serveur. On l'utilise pour simplifier la logique d'intégration et bénéficier d'une syntaxe plus propre que http natif.

```
const axios = require('axios');
axios.get('https://api.exemple.com/users')
  .then(response => console.log(response.data))
  .catch(err => console.error(err));
```

Syntaxe de base d'Axios : GET, POST, PUT, DELETE

Axios propose les méthodes pour effectuer des requêtes HTTP standard : GET, POST, PUT, DELETE. Le résultat est une Promise retournant un objet response contenant data, status, headers.

```
axios.post('https://api.exemple.com/users', { name: 'Alice' })
  .then(res => console.log('Créé avec succès', res.data))
  .catch(console.error);

axios.put('https://api.exemple.com/users/1', { name: 'Alice M.' })
  .then(res => console.log('Mis à jour', res.data))
  .catch(console.error);

axios.delete('https://api.exemple.com/users/1')
  .then(res => console.log('Supprimé', res.status))
  .catch(console.error);
```

Gestion des en-têtes, paramètres de requête et données avec Axios

Vous pouvez personnaliser facilement les en-têtes (headers), les paramètres de requête (params) et envoyer des données en POST/PUT. Ceci facilite l'authentification, le filtrage, ou le tri côté serveur.

```
axios.get('https://api.exemple.com/search', {
  headers: { 'Authorization': 'Bearer token123' },
  params: { q: 'Node.js', limit: 10 }
}).then(res => console.log(res.data));
```

Découvrir node-fetch : une alternative légère à Axios

node-fetch est une implémentation de l'API Fetch du navigateur, côté Node.js. Il est plus léger qu'Axios, et se base sur les Promises. Cependant, il est un peu moins "prêt à l'emploi" qu'Axios, nécessitant parfois plus de code pour traiter les réponses.

```
const fetch = require('node-fetch');

fetch('https://api.exemple.com/data')
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(console.error);
```

Syntaxe de base de node-fetch et utilisation des Promises

Avec node-fetch, vous utilisez fetch(url, options) qui retourne une Promise résolue avec un objet Response. Vous pouvez ensuite appeler response.json() ou response.text() pour traiter le résultat.

```
fetch('https://api.exemple.com/users', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({name: 'Bob'})
})
  .then(res => res.json())
  .then(data => console.log('Utilisateur créé:', data));
```

Gestion des réponses JSON, texte, binaires avec node-fetch

Node-fetch permet de traiter différents formats de réponses. response.json() pour JSON, response.text() pour du texte brut, et response.arrayBuffer() ou response.buffer() pour des données binaires.

```
fetch('https://api.exemple.com/image.png')
  .then(res => res.buffer())
  .then(buffer => {
    // buffer contient l'image, on peut l'enregistrer sur disque
    require('fs').writeFileSync('image.png', buffer);
  });
```

Exercice !

Se connecter à l'API crudcrud.com et connecter les routes « bouchonnées » de l'exercice précèdent.

09

Axios avancé

Gérer les timeouts, retries et interceptions de réponses dans Axios

Axios permet de définir un timeout pour éviter d'attendre indéfiniment. Les intercepteurs (interceptors) gèrent les erreurs globalement (retries, logs) ou modifient les requêtes/réponses avant traitement.

```
axios.defaults.timeout = 5000;

axios.interceptors.response.use(
  response => response,
  error => {
    console.error('Erreur réseau:', error.message);
    // Possibilité d'implémenter un retry ici
    return Promise.reject(error);
  }
);
```

Appels parallèles et en série, utilisation de Promise.all()

Pour accélérer les intégrations, vous pouvez effectuer plusieurs appels en parallèle avec Promise.all(). Faites attention à la limite de taux, aux ressources du serveur externe, et gérez les erreurs de manière appropriée.

```
Promise.all([
  axios.get('https://api.exemple.com/users'),
  axios.get('https://api.exemple.com/products')
])
  .then(([usersRes, productsRes]) => {
    console.log('Users:', usersRes.data);
    console.log('Products:', productsRes.data);
  });
```

Pagination, filtrage et tri des données récupérées

Les APIs REST exposent souvent des mécanismes de pagination (limit, offset, page), de filtrage (par champ), et de tri (sort). Cela permet de ne récupérer qu'une partie des données et d'optimiser les performances du client et du serveur.

```
axios.get('https://api.exemple.com/users', {
  params: { limit: 20, offset: 40, sort: 'name', filter: 'active' }
});
```

10

Authentification

Principes fondamentaux de l'authentification

Exploration des concepts comme l'identité, les sessions, et la gestion des tokens. Introduction aux méthodes courantes comme Basic Auth, OAuth2, et JWT.

```
// Basic Authentication Middleware
app.use((req, res, next) => {
  const auth = req.headers.authorization;
  if (!auth || auth !== 'Basic mysecrettoken') {
    return res.status(401).send('Unauthorized');
  }
  next();
});
```

Mise en place de l'authentification avec JWT

- Introduction aux JSON Web Tokens pour sécuriser les APIs. Explication de la génération et de la vérification des tokens.

```
const jwt = require('jsonwebtoken');

const token = jwt.sign({ userId: 123 }, 'secretKey', { expiresIn: '1h' });
console.log('Generated Token:', token);

jwt.verify(token, 'secretKey', (err, decoded) => {
  if (err) return console.error('Invalid token');
  console.log('Decoded Data:', decoded);
});
```

Utilisation d'Express pour la gestion des sessions

Comment configurer et utiliser des sessions avec express-session pour stocker des informations utilisateurs.

```
const session = require('express-session');

app.use(session({
  secret: 'mySecret',
  resave: false,
  saveUninitialized: true,
}));

app.get('/session', (req, res) => {
  req.session.username = 'JohnDoe';
  res.send('Session set!');
});
```

Sécuriser les mots de passe avec bcrypt

- Utilisation de bcrypt pour hacher et vérifier les mots de passe afin d'assurer la sécurité des utilisateurs.

```
const bcrypt = require('bcrypt');

const password = 'myPassword';
bcrypt.hash(password, 10, (err, hash) => {
  console.log('Hashed password:', hash);

  bcrypt.compare(password, hash, (err, result) => {
    console.log('Password match:', result);
  });
});
```

Authentification OAuth2 avec des fournisseurs tiers

- Connexion via des plateformes comme Google ou Facebook grâce à Passport.js.

```
const passport = require('passport');
const GoogleStrategy = require('passport-google-oauth20').Strategy;

passport.use(new GoogleStrategy({
  clientID: 'GOOGLE_CLIENT_ID',
  clientSecret: 'GOOGLE_CLIENT_SECRET',
  callbackURL: '/auth/google/callback',
}, (accessToken, refreshToken, profile, done) => {
  done(null, profile);
}));

app.get('/auth/google', passport.authenticate('google', { scope: ['profile'] }));
```

11

Intégration de MongoDB

Introduction à MongoDB : une base de données NoSQL orientée documents

MongoDB stocke les données sous forme de documents JSON, offrant une flexibilité de schéma et une évolutivité horizontale. C'est un choix populaire pour les applications Node.js, grâce à sa facilité d'utilisation et ses performances sur de gros volumes de données.

```
{
  "_id": "someUniqueId",
  "name": "Alice",
  "age": 30
}
```

Connexion à MongoDB depuis Node.js : le driver officiel

Le driver officiel MongoDB pour Node.js permet de se connecter à une instance MongoDB, d'interroger la base, d'insérer, de mettre à jour et de supprimer des documents. Vous passez une URI de connexion (avec login/mot de passe si besoin).

```
const { MongoClient } = require('mongodb');
const uri = 'mongodb://localhost:27017/maBase';
const client = new MongoClient(uri);

(async () => {
  await client.connect();
  const db = client.db('maBase');
  const users = await db.collection('users').find().toArray();
  console.log(users);
})();
```

Mongoose : présentation et avantages par rapport au driver natif

Mongoose est un ORM (ODM plus précisément) pour MongoDB. Il fournit un système de schémas, de modèles, de validation, et des méthodes pratiques. Il facilite le travail avec MongoDB en imposant plus de structure et en simplifiant les opérations.

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/maBase');
```

Définir un schéma avec Mongoose et créer un modèle

Avec Mongoose, définissez un schéma décrivant les champs, leurs types et leurs options. Vous créez ensuite un modèle basé sur ce schéma, que vous utiliserez pour interagir avec la collection.

```
const { Schema, model } = require('mongoose');

const userSchema = new Schema({
  name: { type: String, required: true },
  age: Number
});

const User = model('User', userSchema);
```

Opérations CRUD de base avec Mongoose : create, read, update, delete

Mongoose offre des méthodes simples pour créer (save() ou create()), lire (find(), findOne()), mettre à jour (findByIdAndUpdate()), et supprimer (deleteOne(), findByIdAndDelete()) des documents.

```
// CREATE
const newUser = await User.create({ name: 'Alice', age: 30 });
// READ
const users = await User.find({ age: { $gte: 18 } });
// UPDATE
await User.findByIdAndUpdate(newUser._id, { age: 31 });
// DELETE
await User.findByIdAndDelete(newUser._id);
```

Requêtes avancées : filtres, projections, tri, pagination en MongoDB

Mongoose permet d'utiliser les opérateurs MongoDB ($gte, $in, $or) pour filtrer. On peut spécifier une projection (champs à retourner), trier les résultats (.sort()), et paginer (.skip(), .limit()).

```
const users = await User.find({ age: { $gte: 18 } })
  .select('name age')
  .sort({ age: -1 })
  .skip(10)
  .limit(10);
```

Validation des données au niveau du schéma Mongoose

Mongoose prend en charge la validation des données (types, champs obligatoires, validateurs personnalisés). Cela évite d'enregistrer des données invalides en base.

```
const userSchema = new Schema({
  name: { type: String, required: true },
  email: {
    type: String,
    required: true,
    match: /.+\@.+\..+/
  }
});
```

Middlewares Mongoose (pre/post hooks) pour la logique métier

Les middlewares Mongoose (hooks) permettent d'exécuter du code avant/après certaines opérations (save, find, remove). On peut ainsi mettre en place de la logique métier comme le chiffrement de mots de passe ou le logging.

```
userSchema.pre('save', function(next) {
  this.password = hashFunction(this.password);
  next();
});
```

Gestion des erreurs et des exceptions dans les opérations MongoDB

Gérez les erreurs (conflits, duplications, validation) avec try/catch. Vérifiez le code ou le message d'erreur pour adopter une réponse appropriée (erreur 400 si invalide, 500 si problème interne).

```
try {
  const user = await User.create({ name: '' }); // manquant required field
} catch (err) {
  console.error('Erreur lors de la création de l'utilisateur:', err.message);
}
```

Optimisation des performances : index, agrégations, profilage des requêtes

Créez des index sur les champs fréquemment filtrés ou triés pour accélérer les requêtes. Utilisez les agrégations MongoDB pour des analyses complexes. Le profiling (db.setProfilingLevel()) permet de détecter les requêtes lentes et d'optimiser l'architecture.

```
await User.collection.createIndex({ email: 1 }, { unique: true });
```

12

Introduction à express

Introduction à Express.js : le framework serveur minimaliste

Express est un framework minimaliste pour créer rapidement des serveurs HTTP. Il gère facilement les routes, les middlewares et les réponses, simplifiant ainsi le code par rapport au module http natif.

```
const express = require('express');
const app = express();

app.get('/', (req, res) => res.send('Bonjour Express!'));

app.listen(3000, () => console.log('Serveur démarré sur http://localhost:3000'));
```

Créer un routeur Express pour structurer les endpoints

Un router Express permet de grouper les routes liées à une même ressource (ex: /users). Cela rend le code plus organisé et plus facile à maintenir.

```
const express = require('express');
const router = express.Router();

router.get('/users', (req, res) => { /*...*/ });
router.post('/users', (req, res) => { /*...*/ });

module.exports = router;
```

Utiliser des middlewares basiques avec Express

Les middlewares sont des fonctions exécutées avant d'atteindre la route finale. Ils peuvent servir à logger les requêtes, vérifier l'authentification, parser le JSON, etc.

```
app.use(express.json()); // Parse le JSON du body

app.use((req, res, next) => {
  console.log(`${req.method} ${req.url}`);
  next();
});
```

Servir des fichiers statiques avec Express

Express peut servir facilement des fichiers statiques (images, CSS, JS) en spécifiant un répertoire public. Cela évite d'écrire du code pour chaque fichier à renvoyer.

```
app.use(express.static('public'));
// Accès via http://localhost:3000/image.png si image.png est dans /public
```

Gestion des templates (EJS, Pug) pour pages HTML simples

Express peut rendre des pages HTML dynamiques via des moteurs de template (EJS, Pug). On injecte des données côté serveur dans le template pour produire du HTML personnalisé.

```
app.set('view engine', 'ejs');
app.get('/hello', (req, res) => res.render('hello', { nom: 'Alice' }));
```

Utilisation de nodemon pour recharger le serveur à chaud

Nodemon surveille les fichiers source et redémarre automatiquement le serveur lorsqu'un fichier est modifié, améliorant la productivité lors du développement.

```
npm install --save-dev nodemon
npx nodemon server.js
```

Surveillance du code avec PM2 en environnement local

PM2 est un gestionnaire de processus Node.js, utile pour garder le serveur en vie, redémarrer en cas de crash, et fournir des métriques. Il s'utilise facilement en local ou en production.

```
npm install -g pm2
pm2 start server.js
```

Bases du logging (console.log, console.error)

Avant d'utiliser des solutions avancées comme Winston, console.log et console.error suffisent pour débugger. Toutefois, ces méthodes ne gèrent pas la persistance, le formatage avancé ou la rotation de logs.

```
console.log('Serveur démarré');
console.error('Erreur inattendue!');
```

Exercice !

Migrer la solution existante sur express.

13

Introduction à Zod

Présentation de Zod : pourquoi une librairie de validation de schémas ?

Zod est une bibliothèque JavaScript/TypeScript permettant de définir des schémas stricts pour valider des données. Elle aide à s'assurer que les données (d'entrées utilisateur, API externes, etc.) respectent les types et formats attendus, évitant ainsi les erreurs à l'exécution.

```
import { z } from 'zod';

const schémaUtilisateur = z.object({
  nom: z.string(),
  age: z.number().int().positive(),
});
```

Définir un schéma simple avec Zod (strings, numbers, arrays)

Zod fournit des validateurs pour les types de base : z.string(), z.number(), z.boolean(), z.array(), etc. On compose ces validateurs pour créer des schémas simples et s'assurer que les données sont conformes.

```
const schémaProfil = z.object({
  pseudo: z.string().min(3),
  score: z.number().min(0),
  tags: z.array(z.string())
});

const result = schémaProfil.safeParse({ pseudo: "Alice", score: 10, tags: ["JS", "Node"] }
console.log(result.success); // true
```

Validation de données complexes : objets imbriqués et unions

Zod permet de créer des schémas pour des objets imbriqués, des types optionnels, et des unions (valider qu'un objet correspond à l'un des schémas possibles). C'est utile pour gérer des cas complexes.

```
const schémaAdresse = z.object({
  rue: z.string(),
  ville: z.string(),
  codePostal: z.string().length(5),
});

const schémaClient = z.object({
  nom: z.string(),
  adresse: schémaAdresse,
  contact: z.union([z.string().email(), z.string().regex(/^0\d+/)])
});
```

Génération de types TypeScript à partir de schémas Zod

Zod peut générer automatiquement les types TypeScript correspondant à un schéma. Cela évite la redondance entre la définition du type et la validation, garantissant une cohérence totale.

```
const schémaUser = z.object({
  id: z.number(),
  name: z.string(),
});

type User = z.infer<typeof schémaUser>; // Génère le type User basé sur le schéma
```

Utilisation de Zod pour valider les données reçues d'une API externe

Lors de l'intégration de services externes, valider les données reçues est crucial. Zod garantit que votre application ne traite que des données conformes, réduisant les risques d'erreurs inattendues.

```
async function fetchUtilisateur(id) {
  const res = await fetch(`https://api.exemple.com/users/${id}`);
  const data = await res.json();
  const parseResult = schémaUtilisateur.safeParse(data);
  if (!parseResult.success) {
    throw new Error('Données non conformes');
  }
  return parseResult.data;
}
```

Intégration de Zod dans Express : middleware de validation des requêtes

Vous pouvez utiliser Zod pour valider les requêtes entrantes (params, query, body) avec un middleware Express, garantissant que seules les requêtes valides atteignent votre logique métier.

```
function validateBody(schema) {
  return (req, res, next) => {
    const result = schema.safeParse(req.body);
    if (!result.success) {
      return res.status(400).json(result.error.errors);
    }
    req.body = result.data;
    next();
  };
}

app.post('/users', validateBody(schémaUtilisateur), (req, res) => {
  // req.body est maintenant validé
  res.send('Utilisateur valide!');
});
```

Gestion des erreurs de validation Zod et renvoi de réponses HTTP 400

Zod fournit des détails sur les erreurs de validation (champs, messages). Vous pouvez renvoyer un code 400 (Bad Request) et un message clair au client, améliorant l'expérience de debug et de correction côté consommateur de l'API.

```
app.post('/data', (req, res) => {
  const result = schémaData.safeParse(req.body);
  if (!result.success) {
    return res.status(400).json({ erreurs: result.error.format() });
  }
  res.json({ message: 'Données valides' });
});
```

Exercice !

Vérifier les réponses crudcrud.
Ajouter de la vérification de données sur nos différentes routes.

14

Introduction Middlewares

Mettre en place des middlewares avancés : compression, rate limiting, cors

Express supporte des middlewares avancés (compression gzip, CORS, limitation du nombre de requêtes) pour améliorer la performance et la sécurité. On peut les plugger facilement dans la chaîne de middlewares.

```
const compression = require('compression');
const rateLimit = require('express-rate-limit');
const cors = require('cors');

app.use(compression());
app.use(rateLimit({ windowMs: 60000, max: 100 }));
app.use(cors());
```

Utilisation avancée du router Express : sous-routeurs et organisation par domaine

On peut créer des sous-routeurs Express par domaine (ex: /users, /products) pour mieux organiser le code. Chaque sous-routeur gère un groupe d'endpoints logiquement liés.

```
const usersRouter = express.Router();
usersRouter.get('/', getUsers);
usersRouter.post('/', createUser);

app.use('/users', usersRouter);
```

Gestion d’erreurs globales dans Express avec un middleware dédié

Un middleware d’erreurs global permet de centraliser la gestion des exceptions non gérées. Il peut renvoyer des réponses cohérentes (JSON) avec un code de statut adapté et des messages clairs.

```
app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).json({ message: 'Erreur interne du serveur' });
});
```

Exercice !

Créer un middleware zod pour vérifier nos différentes requêtes.

15

Un peu + sur le débug

Introduction au débogage avancé : Pourquoi aller au-delà du console.log ?

Le console.log suffit pour les petits projets, mais pour diagnostiquer des problèmes complexes (fuites mémoire, performances, logique métier tordue), un débogage plus avancé est nécessaire. Des outils comme les DevTools, le profiling CPU/mémoire, ou des logs structurés aident à comprendre plus finement ce qui se passe et à gagner du temps.

```
// console.log n'est pas toujours suffisant pour comprendre un bug complexe
console.log("Valeur actuelle:", variable);
// Mieux : utiliser des breakpoints et inspecter l'état complet de l'application
```

Utilisation du flag --inspect et --inspect-brk avec Node.js

node --inspect lance Node.js avec une interface de débogage compatible avec Chrome DevTools. --inspect-brk met en pause dès le début, permettant d'inspecter l'état initial de l'application.

```
node --inspect app.js
node --inspect-brk app.js
```

Connexion aux Chrome DevTools pour déboguer le code Node.js

En lançant Node.js avec --inspect, vous obtenez une URL (ws://...) à ouvrir dans Chrome. Les Chrome DevTools permettent de poser des breakpoints, inspecter des variables et exécuter du code.

```
node --inspect app.js
# Ouvrez ensuite chrome://inspect dans Chrome et connectez-vous au runtime Node
```

Points d'arrêt (breakpoints) : poser, inspecter, reprendre l'exécution

Les breakpoints arrêtent le code à une ligne spécifique, vous permettant d'inspecter les variables, la pile d'appels, et d'exécuter pas à pas. Reprenez ensuite l'exécution pour poursuivre jusqu'au prochain breakpoint ou la fin.

```
// Dans DevTools, cliquez sur le numéro de ligne pour poser un breakpoint
function calcul(x) {
  const y = x * 2; // Posez un breakpoint ici
  return y + 10;
}
console.log(calcul(5));
```

Profils de performance (CPU Profiling) : identification des goulots d'étranglement

Le CPU Profiling dans DevTools permet de voir quelles fonctions consomment le plus de temps. Vous identifiez ainsi les goulots d'étranglement pour optimiser votre code.

```
# Lancez avec --inspect, connectez Chrome DevTools, onglet "Profiler"
```

Utilisation de traceurs et de logs structurés (debug, Winston) pour le diagnostic

Au-delà de DevTools, des logs structurés (avec Winston, Pino) et l'utilisation de la bibliothèque debug facilitent l'analyse de problèmes en production. Les logs structurés (format JSON, niveaux de sévérité) s'intègrent mieux aux outils de monitoring.

```
const debug = require('debug')('app:main');
debug('Message détaillé');

// Avec Winston
const { createLogger, transports } = require('winston');
const logger = createLogger({ transports: [new transports.Console()] });
logger.info('Démarrage du serveur');
```

16

Pattern évènementiel

Comprendre le pattern événementiel en Node.js

Le pattern événementiel permet de réagir à la survenue d'événements (actions, signaux, données reçues) plutôt que d'attendre passivement. Il repose sur la souscription d'écouteurs (listeners) qui se déclenchent lorsque l'événement survient, permettant une architecture découplée et modulable.

```
// Exécution basée sur événements
process.on('exit', () => {
  console.log('Le processus se termine.');
});

// Un simple événement "ping"
const EventEmitter = require('events');
const emetteur = new EventEmitter();

emetteur.on('ping', () => console.log('Pong!'));
emetteur.emit('ping');
```

Le module events et la classe EventEmitter

Le module events introduit EventEmitter, une classe permettant de créer des objets capables d'émettre et d'écouter des événements. Les instances d'EventEmitter supportent on, once, emit, et gèrent facilement plusieurs listeners.

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

class MonEmetteur extends EventEmitter {}

const em = new MonEmetteur();
em.on('salut', (nom) => {
  console.log(`Salut, ${nom}!`);
});

em.emit('salut', 'Alice');
```

Création d'émetteurs d'événements personnalisés

Vous pouvez étendre la classe EventEmitter pour créer des émetteurs spécifiques à votre domaine (par exemple, un émetteur pour la communication interne d'un module). Cela permet de structurer la logique en événements cohérents.

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

class ServeurFictif extends EventEmitter {
  démarrer() {
    console.log('Démarrage du serveur...');
    setTimeout(() => this.emit('pret'), 1000);
  }
}

const serveur = new ServeurFictif();
serveur.on('pret', () => console.log('Le serveur est prêt!'));
serveur.démarrer();
```

Gestion d'événements multiples, hiérarchies et priorité

Un EventEmitter peut avoir plusieurs écouteurs pour le même événement. L'ordre d'enregistrement détermine l'ordre d'appel. Il est possible de structurer des chaînes d'événements (émission d'un événement en réaction à un autre) pour gérer des flux complexes.

```
const em = new (require('events'))();

em.on('data', (d) => console.log('Premier listener:', d));
em.on('data', (d) => console.log('Deuxième listener:', d));

em.emit('data', {val: 42});
// Affiche dans l'ordre "Premier listener: {val: 42}" puis "Deuxième listener: ..."
```

Détacher des écouteurs, prévenir les fuites de mémoire et debugging

Ne pas détacher les écouteurs inutilisés peut causer des fuites de mémoire. Utilisez removeListener ou removeAllListeners lorsque les écouteurs ne sont plus nécessaires. Le module events émet des avertissements si trop d'écouteurs sont ajoutés au même événement.

```
const em = new (require('events'))();

function listener() { console.log('Événement capté!'); }

em.on('info', listener);
em.emit('info');
em.removeListener('info', listener);
em.emit('info'); // N'affichera rien
```

Événements système (process, fs, net) et intégration

Outre vos émetteurs personnalisés, Node.js expose des événements sur process (ex: beforeExit), fs (watch), net (sockets) et d'autres modules. L'événementiel est un mécanisme clé pour réagir à des changements système et intégrer diverses fonctionnalités.

```
process.on('beforeExit', () => {
  console.log('Le processus va se terminer.');
});

// fs.watch déclenche des événements sur changement de fichier
const fs = require('fs');
fs.watch('monFichier.txt', (eventType, filename) => {
  console.log(`Changement détecté: ${eventType} sur ${filename}`);
});
```

Intégration des événements dans la logique serveur (changement de fichiers, rechargement à chaud)

Combinez les EventEmitter avec votre serveur. Par exemple, vous pouvez émettre un événement quand un fichier statique change et recharger automatiquement en mémoire. Cela permet un rechargement à chaud (hot reload) sans redémarrer le serveur.

```
const fs = require('fs');
const EventEmitter = require('events');

const emitter = new EventEmitter();
let contenu = fs.readFileSync('page.html', 'utf8');

fs.watch('page.html', () => {
  contenu = fs.readFileSync('page.html', 'utf8');
  emitter.emit('fichierChange');
});

http.createServer((req, res) => {
  res.setHeader('Content-Type', 'text/html');
  res.end(contenu);
}).listen(3000);
```

Combiner la programmation asynchrone et l'EventEmitter pour des flux de données complexes

L'asynchronisme (Promesses, async/await) se combine parfaitement avec les événements. Vous pouvez émettre un événement lorsque des données asynchrones sont disponibles, gérer des files d'attente, et orchestrer des flux complexes de manière plus lisible.

```
const em = new EventEmitter();

async function fetchData() {
  const data = await simulateAsyncFetch();
  em.emit('dataReçue', data);
}

em.on('dataReçue', (data) => {
  console.log('Données:', data);
});

fetchData();
```

17

Introduction à Jasmine

Pourquoi écrire des tests unitaires ?

Les tests unitaires valident le bon fonctionnement d'éléments isolés du code (fonctions, modules) sans dépendre d'autres parties du système. Ils aident à prévenir les régressions, documentent le comportement attendu, et améliorent la confiance lors de modifications futures.

Présentation de Jasmine : philosophie et principes

Jasmine est un framework de tests unitaires pour JavaScript, orienté comportement (BDD). Il permet d'écrire des tests expressifs, sans dépendre du DOM ou d'un navigateur, et fournit un ensemble complet de matchers, de fonctions d'organisation et de hooks.

```
describe('Une suite de tests', () => {
  it('devrait faire quelque chose', () => {
    expect(true).toBe(true);
  });
});
```

Installation et configuration de Jasmine dans un projet Node.js

Installez Jasmine via npm et initialisez-le avec jasmine init. Cela crée un répertoire spec pour les tests et un fichier de configuration. Aucun navigateur requis, tout s'exécute dans Node.js.

```
npm install --save-dev jasmine
npx jasmine init
npx jasmine
```

Structure d’un test Jasmine : describe, it, expect

describe regroupe des tests qui ciblent une fonctionnalité ou un module. it décrit un cas de test spécifique. expect vérifie le résultat attendu. Ces mots-clés créent une structure lisible, facilitant la compréhension de l’intention du test.

```
describe('Math Utils', () => {
  it('devrait additionner deux nombres', () => {
    expect(1 + 1).toBe(2);
  });
});
```

Les matchers de base : toBe, toEqual, toThrow, etc.

Jasmine fournit des matchers pour comparer des valeurs (toBe, toEqual), vérifier des exceptions (toThrow), des booléens (toBeTruthy, toBeFalsy), des close en valeur numérique (toBeCloseTo), etc. Cela permet une grande expressivité dans l'écriture des assertions.

```
expect(2 + 2).toBe(4);
expect({a:1}).toEqual({a:1});
expect(() => { throw new Error('Oups'); }).toThrow();
```

Matchers de base : toBe, toEqual, toMatch, toContain

toBe : Test d'égalité stricte (===)toEqual : Test d'égalité de valeur (profondes pour objets/arrays)toMatch : Test de correspondance avec une RegExptoContain : Vérifie la présence d'un élément dans un tableau ou une sous-chaîne

```
expect(10).toBe(10);
expect({a:1}).toEqual({a:1});
expect('Hello Jasmine').toMatch(/Jasmine/);
expect([1,2,3]).toContain(2);
```

Matchers de vérification d'existence et de types

toBeDefined() : Vérifie que la valeur n'est pas undefined. toBeNull() : Vérifie que la valeur est null. toBeTruthy() : Vérifie que la valeur est évaluée comme vraie (non falsy).

```
let value = 'test';
expect(value).toBeDefined();

value = null;
expect(value).toBeNull();

value = true;
expect(value).toBeTruthy();
```

Matchers numériques : toBeGreaterThan, toBeLessThan, toBeCloseTo

toBeGreaterThan(x) : valeur > xtoBeLessThan(x) : valeur < xtoBeCloseTo(x, précision) : valeur proche de x à une certaine précision décimale

```
expect(10).toBeGreaterThan(5);
expect(2).toBeLessThan(5);
expect(3.14159).toBeCloseTo(3.14, 2);
```

Matchers sur les tableaux et objets : toContain, toHaveBeenCalled

toContain() déjà vu, pour tableaux ou chaînes.toHaveBeenCalled() vérifie qu'une fonction espionnée (spy) a été appelée.

```
const arr = [1,2,3];
expect(arr).toContain(2);

const spy = jasmine.createSpy('mySpy');
spy();
expect(spy).toHaveBeenCalled();
```

Utiliser la négation : not

Vous pouvez inverser n'importe quel matcher avec .not pour vérifier l'inverse.

```
expect(5).not.toBe(3);
expect('Hello').not.toMatch(/World/);
```

Organisation des suites de tests et fichiers de spécifications

Placez vos fichiers de test dans le répertoire spec/ (par défaut) et nommez-les xxx.spec.js pour une détection automatique. Organisez vos describe par fonctionnalité pour une structure cohérente et facile à parcourir.

```
project/
  src/
    math.js
  spec/
    math.spec.js
```

Exécuter les tests : commandes, options, watch mode

Par défaut, npx jasmine exécute tous les tests. Vous pouvez filtrer par nom de fichier, ajouter un watcher externe (chokidar) pour relancer automatiquement les tests, ou intégrer Jasmine dans des scripts npm.

```
npx jasmine spec/math.spec.js
```

Configuration avancée du runner Jasmine (jasmine.json)

Le fichier jasmine.json permet de configurer les chemins des tests, d'ajouter des reporters, d'ignorer certains fichiers, de définir des helpers. Cela offre une flexibilité pour adapter le runner à vos besoins.

```
{
  "spec_dir": "spec",
  "spec_files": ["**/*[sS]pec.js"],
  "helpers": ["helpers/**/*.js"]
}
```

Rendre les tests lisibles et maintenables : bonnes pratiques

Utilisez des noms descriptifs pour describe et it, ne testez qu'une chose par test, factorisez le code commun, et évitez les dépendances cachées. Les tests doivent raconter une histoire claire sur le comportement attendu.

```
describe('Function sum()', () => {
  it('retourne la somme de deux nombres', () => {
    expect(sum(2, 3)).toBe(5);
  });
});
```

Utiliser les beforeEach et afterEach pour préparer l'environnement de test

beforeEach s'exécute avant chaque test, afterEach après chaque test. Ils permettent d'initialiser ou nettoyer l'état, créant un environnement stable pour chaque spec.

```
beforeEach(() => {
  // initialiser certaines variables
});
afterEach(() => {
  // nettoyer après les tests
});
```

Réduire la duplication grâce à beforeAll/afterAll

beforeAll() et afterAll() permettent d'effectuer des opérations coûteuses (connexion BDD, initialisation) une seule fois au début et à la fin d'une suite, évitant ainsi la duplication de code.

```
describe('BDD', () => {
  let db;

  beforeAll(() => {
    db = { connect: () => true };
    db.connect();
  });

  afterAll(() => {
    db = null;
  });

  it('test sur BDD', () => {
    expect(db).toBeDefined();
  });
});
```

Différence entre tests synchrones et asynchrones

Les tests synchrones vérifient une fonction qui retourne immédiatement un résultat. Les tests asynchrones impliquent des callbacks, Promises, ou async/await. Le test doit attendre la fin de l'opération asynchrone avant d'asserter le résultat.

```
// Synchrone
expect(somme(2,3)).toBe(5);

// Asynchrone (Promise)
expectAsync(fetchData()).toBeResolved();
```

Validation des conditions de succès et d'échec sur les fonctions synchrones

Testez non seulement les conditions “heureuses” (cas correct), mais aussi les cas d'erreur (entrées invalides, retours inattendus) pour garantir une robustesse.

```
expect(() => somme('a', 2)).toThrow();
```

Gérer les exceptions et les erreurs prévues

Si une fonction doit lever une exception dans certaines conditions, utilisez toThrow pour vérifier ce comportement. Cela s'assure que la fonction réagit correctement aux entrées invalides.

```
it('doit jeter une erreur si les arguments ne sont pas numériques', () => {
  expect(() => somme('a', 'b')).toThrowError('Arguments invalides');
});
```

Introduction aux fonctions asynchrones dans les tests

Lorsque la fonction testée effectue un appel réseau, lit un fichier, ou utilise un setTimeout, elle ne renvoie pas de résultat immédiatement. Vous devez donc informer Jasmine lorsque le test est terminé.

```
it('devrait obtenir des données asynchrones', (done) => {
  fetchData((data) => {
    expect(data).toBeDefined();
    done();
  });
});
```

Utiliser done() pour signaler la fin d'un test asynchrone

Pour les callbacks asynchrones, Jasmine attend l'appel à done() pour savoir que le test est terminé. Sans done(), Jasmine risquerait de conclure le test trop tôt.

```
it('devrait appeler la callback', (done) => {
  asyncFunction(() => {
    expect(true).toBe(true);
    done();
  });
});
```

Tests asynchrones avec Promises : expectAsync et await

Avec les Promises, utilisez expectAsync(...).toBeResolved() ou toBeRejected(). Avec async/await, le code asynchrone devient lisible et linéaire, simplifiant les tests.

```
it('retourne une Promise résolue', () => {
  return expectAsync(fetchData()).toBeResolved();
});
```

```
it('retourne des données', async () => {
  const data = await fetchData();
  expect(data).toBeDefined();
});
```

Gestion des erreurs dans les tests asynchrones

Testez également les cas d'erreur asynchrones (Promise rejetée, callback avec erreur). Cela garantit une gestion robuste des échecs.

```
it('devrait rejeter la Promise en cas d'erreur', () => {
  return expectAsync(fetchDataErreur()).toBeRejected();
});
```

Tester des appels réseau fictifs ou des timers asynchrones

Mockez les appels réseau (avec spies, nock ou fetch-mock) et utilisez jasmine.clock() pour contrôler les timers. Ceci permet de tester des comportements asynchrones sans dépendre de ressources externes.

```
jasmine.clock().install();
setTimeout(() => console.log('Fait'), 1000);
jasmine.clock().tick(1000);
jasmine.clock().uninstall();
```

Tests paramétriques pour couvrir différents scénarios

Les tests paramétriques (par exemple en itérant sur un tableau de valeurs) permettent de couvrir plusieurs scénarios avec le même code de test, évitant la répétition.

```
[ [1,2,3], [2,2,4] ].forEach(([a,b,res]) => {
  it(`somme(${a},${b}) = ${res}`, () => {
    expect(somme(a,b)).toBe(res);
  });
});
```

Définition de mocks et stubs : similitudes et différences

Un stub remplace une fonction ou une méthode en fournissant un comportement prédéfini, alors qu'un mock non seulement fournit ce comportement, mais vérifie également que la fonction a été appelée comme prévu (arguments, nombre d'appels). Les mocks sont plus complets, tandis que les stubs sont plus simples et souvent limités aux retours pré-programmés.

```
// Un stub : juste un retour programmé
spyOn(obj, 'method').and.returnValue('stubbed');
```

Pourquoi utiliser des mocks ? Réduire la dépendance aux ressources externes

Les mocks permettent de tester le code sans faire réellement appel aux ressources externes (APIs, bases de données, services distants). Cela rend les tests plus rapides, plus stables (pas de panne externe), et facilite le contrôle des scénarios (réponses, délais).

```
// Empêcher un appel réel vers l'API externe, fournir une réponse mockée
spyOn(apiClient, 'getUsers').and.returnValue(Promise.resolve([{id:1,name:'Alice'}]));
```

Création d’un stub basique en Jasmine

Un stub dans Jasmine se fait via spyOn. On remplace la méthode ciblée par un faux comportement. Aucune vérification n’est faite sur le nombre d’appels, sauf si vous l’ajoutez explicitement.

```
spyOn(Math, 'random').and.returnValue(0.5);
expect(Math.random()).toBe(0.5);
```

Espionner (spy) des fonctions internes : spyOn et ses options (and.callFake, etc.)

spyOn(obj, 'method') crée un espion sur la méthode, vous pouvez ensuite définir son comportement : and.returnValue(), and.throwError(), and.callFake() (une fonction personnalisée), and.callThrough() (appeler la vraie fonction).

```
spyOn(obj, 'compute').and.callFake((x,y) => x*y);
expect(obj.compute(3,4)).toBe(12);
```

Contrôler le comportement d'une fonction mockée (retours, erreurs)

Les mocks permettent de simuler différents scénarios. Vous pouvez forcer une fonction à retourner une erreur, une promesse résolue ou rejetée, ou tout autre retour adapté à votre test. Cela facilite le test des cas d'erreur et de succès.

```
spyOn(service, 'fetchData').and.returnValue(Promise.reject(new Error('Échec')));
```

Mocking d'une dépendance réseau ou d'un module externe

Remplacez les appels réseau par des mocks pour éviter la dépendance à l'infrastructure externe. Vous pouvez par exemple espionner la méthode fetch globale, ou votre client HTTP, pour fournir une réponse fixe.

```
spyOn(window, 'fetch').and.returnValue(Promise.resolve({ json: () => ({ status: 'ok' }) }))
```

Combiner mocks et tests asynchrones

Les mocks asynchrones (Promesses, callbacks différés) permettent de tester la logique asynchrone sans réellement attendre un vrai réseau. Vous contrôlez le moment où la promesse se résout ou se rejette.

```
it('test asynchrone avec mock', async () => {
  spyOn(apiClient, 'getData').and.returnValue(Promise.resolve('mockedData'));
  const result = await apiClient.getData();
  expect(result).toBe('mockedData');
});
```

Utilisation de librairies externes (nock) pour simuler des appels HTTP

Nock intercepte les appels HTTP sortants et répond avec des réponses simulées, sans avoir à modifier le code. C'est pratique pour tester les intégrations API sans dépendre du réseau.

```
const nock = require('nock');
nock('https://api.exemple.com')
  .get('/users')
  .reply(200, [{id:1,name:'Alice'}]);
```

18

Introduction à Allure

Pourquoi un rapport de test ? Importance des rapports lisibles

Un rapport de test lisible permet de comprendre rapidement les résultats (succès/échecs), le temps d'exécution, et de communiquer clairement la qualité du code aux parties prenantes. Cela facilite le triage des bugs et la prise de décision.

Présentation d’Allure : un outil de reporting flexible

Allure génère des rapports riches (HTML) à partir des résultats de tests. Il offre des fonctionnalités avancées : sections de description, captures, métriques de temps, ce qui permet une analyse plus poussée que de simples logs.

```
npm install allure-commandline jasmine-allure-reporter --save-dev
```

Installation et configuration d'Allure avec Jasmine

Installez Allure commandline et un reporter Jasmine compatible (comme jasmine-allure-reporter). Configurez le reporter dans jasmine.json ou dans les helpers.

```
npx jasmine
npx allure generate allure-results --clean
npx allure open
```

Exécuter les tests et générer un rapport Allure

Après avoir configuré Allure, lancez les tests. Les résultats sont enregistrés dans un dossier de résultats Allure. Ensuite, exécutez allure generate pour produire le rapport HTML, puis allure open pour le visualiser.

```
// Avec Jasmine-Allure, vous pouvez utiliser des hooks pour ajouter des métadonnées
allure.feature('Authentification');
allure.story('Connexion réussie');
```

Personnalisation du rapport (titre, description, étiquettes)

Allure permet d’ajouter des annotations (description, severity, labels) dans les tests. Ces métadonnées améliorent la compréhension du contexte du test dans le rapport.

Analyser les résultats via l'interface Allure (HTML)

Le rapport HTML Allure affiche les tests par suite, scénario, avec un statut (réussi, échoué, ignoré), des graphes de durée, des détails des erreurs et des étapes. Cela offre une vue complète de la qualité du produit.

Capturer des informations supplémentaires (logs, screenshots, attachments)

Allure permet d'attacher des fichiers (captures d'écran, logs, dumps JSON) pour aider au diagnostic des échecs. Cela transforme le rapport en une ressource riche pour le debugging.

```
allure.createAttachment('Données utilisateur', JSON.stringify(user), 'application/json');
```

Organiser les tests par suites, fonctionnalités et scénarios

Organisez vos tests en utilisant describe, feature, story et d'autres étiquettes. Allure structurera le rapport selon ces catégories, facilitant la navigation et la compréhension du périmètre couvert.

```
allure.feature('Paiement');
allure.story('Traitement carte de crédit');
```

Interpréter les métriques de Allure : taux de succès, temps moyen

Allure génère des métriques sur le pourcentage de succès, la durée moyenne, la distribution des échecs. Ces indicateurs aident à identifier les domaines problématiques et à suivre l'évolution de la qualité dans le temps.

Intégrer Allure dans la chaîne de CI pour une visibilité continue

Exécutez Allure dans votre pipeline CI (Jenkins, GitLab CI, GitHub Actions) pour générer automatiquement des rapports après chaque build. Les équipes peuvent ainsi surveiller en continu la qualité du code et réagir rapidement.

```
# Dans le script CI
npm test
allure generate allure-results --clean
allure open
```

19

Introduction à Jenkins

Qu’est-ce que l’intégration continue ? Principes et bénéfices

L’intégration continue (CI) consiste à intégrer régulièrement du code dans un dépôt partagé, à exécuter des tests et à vérifier la qualité à chaque commit. Les bénéfices incluent une détection précoce des problèmes, une livraison plus rapide et une plus grande confiance dans le code.

Présentation de Jenkins : serveur CI open source

Jenkins est un outil CI open source très populaire, extensible via des plugins. Il permet d'automatiser l'exécution des builds, tests, et déploiements, avec une interface web pour configurer et suivre les pipelines.

Installation et configuration de base d'un pipeline Jenkins

Après avoir installé Jenkins, configurez un pipeline via un Jenkinsfile dans votre dépôt. Définissez les étapes (stages) comme “Build”, “Test”, “Report”. Jenkins déclenchera le pipeline à chaque commit.

```
pipeline {
  stages {
    stage('Build') {
      steps {
        sh 'npm install'
      }
    }
    stage('Test') {
      steps {
        sh 'npx jasmine'
      }
    }
  }
}
```

Intégration des tests Jasmine dans un pipeline Jenkins

Ajoutez une étape Test dans votre Jenkinsfile pour exécuter npx jasmine. Jenkins affichera les logs. Pour des rapports plus sophistiqués, générez un rapport JUnit (via un reporter) et archivez-le.

```
stage('Test') {
  steps {
    sh 'npx jasmine --junitreport'
    junit 'junitreport.xml'
  }
}
```

Exécution des tests sur chaque commit (trigger par webhook Git)

Configurez un webhook GitHub/GitLab pour déclencher le pipeline Jenkins à chaque push. Ainsi, les tests s’exécutent automatiquement, garantissant une intégration continue réelle et un feedback rapide aux développeurs.

Collecte et archivage des rapports Allure dans Jenkins

Intégrez Allure dans votre pipeline Jenkins. Après les tests, générez le rapport Allure et utilisez le plugin Jenkins Allure pour afficher le rapport directement dans l'interface Jenkins.

```
stage('Allure Report') {
  steps {
    sh 'allure generate allure-results --clean'
    allure includeProperties: false, jdk: '', results: [[path: 'allure-results']]
  }
}
```

Notifications de résultats de tests (e-mail, Slack, etc.)

Configurez Jenkins pour envoyer des notifications en cas d'échec ou de réussite (mail, Slack). Cela informe immédiatement l'équipe, évitant que des problèmes passent inaperçus.

```
post {
  failure {
    slackSend channel: '#builds', message: 'Tests en échec!'
  }
}
```

Gérer plusieurs environnements de test dans Jenkins

Créez plusieurs jobs ou pipelines pour tester le code sur différents environnements (Node.js versions différentes, OS, bases de données). Cela assure une couverture plus large et une meilleure robustesse.

```
matrix {
  axis {
    name 'NODE_VERSION'
    values '12', '14', '16'
  }
  stages {
    stage('Test') {
      steps {
        sh "nvm use ${NODE_VERSION} && npx jasmine"
      }
    }
  }
}
```

Bonnes pratiques pour la maintenance d’un pipeline CI stable

Gardez le pipeline simple et lisible, versionnez le Jenkinsfile, nettoyez régulièrement les environnements, surveillez les temps d’exécution et les taux de succès. Automatisez autant que possible, documentez, et mettez à jour les dépendances.

20

Les workers thread

Introduction aux Worker Threads : pourquoi et quand les utiliser

Les Worker Threads permettent d'exécuter du code JavaScript dans des threads séparés au sein du même processus. Contrairement à un clustering qui lance plusieurs processus, les worker threads partagent la même mémoire, facilitant la communication. Ils sont utiles pour des tâches CPU-intensives (cryptage, compression, calcul mathématique lourd) afin de libérer l'Event Loop du thread principal.

Création de worker threads : le module ‘worker_threads’

Le module worker_threads introduit la classe Worker pour créer un thread séparé. On lui passe un script à exécuter (fichier .js) et des options (env, argv, etc.). Le Worker s’exécute en parallèle du thread principal, offrant un vrai parallélisme côté CPU.

```
const { Worker } = require('worker_threads');
const worker = new Worker('./worker-script.js');
worker.on('message', msg => console.log('Message du worker :', msg));
```

Communication entre le thread principal et les workers (postMessage, onmessage)

La communication s'effectue par envoi de messages via worker.postMessage() et écoute d'événements message. Les données échangées sont sérialisées, mais on peut aussi transférer des transferrables (ArrayBuffer) sans copie.

```
// Dans le fichier principal
worker.postMessage({ commande: 'calcul', data: 42 });

// Dans le worker-script.js
const { parentPort } = require('worker_threads');
parentPort.on('message', (msg) => {
  // Traitement
  parentPort.postMessage({ result: msg.data * 2 });
});
```

Gestion des erreurs et exceptions dans les workers

Les erreurs lancées dans un worker sont transmises au thread principal via l'événement error. Il est important de surveiller ces erreurs pour éviter des comportements imprévisibles. Le code dans le worker doit être robuste.

```
worker.on('error', (err) => {
  console.error('Erreur dans le worker :', err);
});
```

Optimisation des performances : cas pratiques avec les worker threads

Les Worker Threads améliorent la performance quand le code est CPU-bound. Par exemple, pour un chiffrement, parsing XML complexe ou compression d'images, déléguer la tâche à plusieurs workers augmente le throughput et prévient les blocages du thread principal.

Utilisation des workers pour le traitement CPU-intensif

Au lieu de bloquer le thread principal (et donc toutes les requêtes entrantes dans un serveur), lancez un Worker pour la tâche lourde. Le serveur reste réactif, tandis que le Worker exécute son calcul en parallèle.

20

Les process events

Comprendre le module ‘process’ et ses *événements clés*

Le module process fournit des informations et des contrôles sur le processus Node.js en cours d’exécution. Il émet des événements (exit, beforeExit, unhandledRejection) permettant de réagir à la fin du processus ou à des erreurs.

```
process.on('beforeExit', () => console.log('Le processus va bientôt se terminer.'));
```

Les événements de sortie du processus (exit) et les signaux du système (SIGINT, SIGTERM)

L'événement exit est émis lorsque le processus se termine. Les signaux système comme SIGINT (Ctrl+C) ou SIGTERM (arrêt gracieux) peuvent être capturés (process.on('SIGINT', ...)) pour nettoyer avant l'arrêt.

```
process.on('SIGINT', () => {
  console.log('Processus interrompu, nettoyage...');
  process.exit();
});
```

Gestion des erreurs globales (uncaughtException, unhandledRejection)

uncaughtException intercepte les exceptions non gérées, unhandledRejection les promesses rejetées sans .catch(). Ces événements permettent de loguer l'erreur, alerter et arrêter proprement le processus au lieu d'un crash brutal.

```
process.on('unhandledRejection', (reason) => {
  console.error('Promesse non gérée:', reason);
});
```

Observation des événements de performance et de ressources du processus

En surveillant process (mémoire utilisée, CPU, etc.), on peut émettre des alarmes, collecter des métriques, et optimiser le code. Certains modules et événements aident à tracer l'utilisation des ressources.

```
console.log('Mémoire utilisée:', process.memoryUsage());
```

Bonnes pratiques pour gérer la fin et le redémarrage d'un processus Node.js

Toujours nettoyer les ressources (connexions DB, fichiers ouverts) avant la fin. Utiliser un gestionnaire de processus (PM2, forever) pour redémarrer automatiquement en cas de crash. Planifier une stratégie de graceful shutdown est essentiel en production.

```
process.on('SIGTERM', shutdownGracefully);
```

21

Propositions

TC39

Les Stages des Propositions TC39

Stage 0 : Strawman (Ébauche)

Stage 1 : Proposal (Proposition)

Stage 2 : Draft (Brouillon)

Stage 3 : Candidate (Candidat)

Stage 4 : Finished (Finalisé)

Champs de Classe et Fonctionnalités Statiques

- **Stage :** 4
- **Description :** Ajoute des champs publics et privés aux classes ainsi que des fonctionnalités statiques.

```
class MyClass {
  static staticField = 'valeur statique';
  publicField = 'valeur publique';
  #privateField = 'valeur privée';

  static staticMethod() {
    console.log(this.staticField);
  }

  publicMethod() {
    console.log(this.publicField);
    console.log(this.#privateField);
  }
}

const instance = new MyClass();
instance.publicMethod();
MyClass.staticMethod();
```

Décorateurs

- **Stage :** 3
- **Description :** Permet d'annoter et de modifier des classes et des méthodes.

```
function decorator(target) {
  target.decorated = true;
}

@decorator
class MyClass {}

console.log(MyClass.decorated); // true
```

Await au Niveau Supérieur

Stage : 4
Description : Permet l'utilisation de await au niveau supérieur dans les modules.

```
// module.js
const data = await fetchData();
console.log(data);
```

Record et Tuple

- **Stage :** 2
- **Description :** Introduit des structures de données immuables similaires aux objets et tableaux.

```
const record = #{ a: 1, b: 2 };
const tuple = #[1, 2, 3];

console.log(record.a); // 1
console.log(tuple[0]); // 1
```

Temporal

- **Stage :** 3
- **Description :** Une nouvelle API pour la gestion des dates et heures.

```
const now = Temporal.Now.instant();
const date = Temporal.PlainDate.from('2020-01-01');

console.log(now.toString()); // 2024-05-17T12:34:56.789Z
console.log(date.add({ days: 1 }).toString()); // 2020-01-02
```

Correspondance de Motifs

- **Stage :** 1
- **Description :** Introduit une syntaxe de correspondance de motifs pour les structures de données.

```
function match(value) {
  return value.match(
    { x: 1, y: 2 }, () => 'matched x:1, y:2',
    { x: 3 }, () => 'matched x:3',
    () => 'default'
  );
}

console.log(match({ x: 1, y: 2 })); // 'matched x:1, y:2'
```

Array.prototype.at

- **Stage :** 4
- **Description :** Ajoute une méthode pour accéder aux éléments d'un tableau par index négatif.

```
const array = [1, 2, 3, 4, 5];

console.log(array.at(-1)); // 5
console.log(array.at(-2)); // 4
```

Nouveautés Javascript

ECMAScript 6 (ES6) - 2015
Apport majeur :

Classes : Introduction de la syntaxe de classe pour la programmation orientée objet, qui est plus propre et facile à utiliser.
Modules : Standardisation du support des modules ES6 pour une meilleure gestion du code en facilitant l'import et l'export de composants.
Promesses : Pour la gestion asynchrone, facilitant l'écriture de code asynchrone et la gestion des opérations asynchrones.

Nouveautés Javascript

Autres apports :

let et const : Nouveaux mots-clés pour les déclarations de variables, offrant des portées de bloc, réduisant ainsi les erreurs communes dues aux variables globales.
Fonctions fléchées : Syntaxe plus concise pour l'écriture de fonctions, et ne crée pas de nouveau contexte this.
Paramètres par défaut, Rest et Spread : Amélioration de la gestion des paramètres de fonctions.
Destructuration : Permet une affectation plus facile et plus claire des données à partir d'arrays ou d'objets.
Templates littéraux : Intégration facile des variables dans les chaînes de caractères avec les templates littéraux.

ECMAScript 7 (ES7) - 2016

Opérateur d'exponentiation (**) : Pour calculer la puissance d'un nombre.
Méthode Array.prototype.includes : Vérifie si un tableau inclut un élément donné, renvoyant true ou false.

ECMAScript 8 (ES8) - 2017

async et await : Simplification de l'écriture de fonctions asynchrones pour rendre le code asynchrone aussi facile à lire et à écrire que le code synchrone.
Méthodes de l'objet : Object.values(), Object.entries(), et Object.getOwnPropertyDescriptors() pour une meilleure manipulation des objets.

ECMAScript 9 (ES9) - 2018

Opérateur de décomposition pour objets : Étend les fonctionnalités de décomposition aux objets.
Promesses finally() : Méthode ajoutée aux promesses pour exécuter du code une fois que la promesse est réglée, indépendamment du résultat.

ECMAScript 10 (ES10) - 2019

Array.prototype.{flat, flatMap} : Méthodes pour aplatir des tableaux imbriqués et appliquer une fonction, puis aplatir le résultat.
Object.fromEntries() : Transforme une liste de paires clé-valeur en un objet.
Chaînes de caractères et modifications de Array.sort : Améliorations mineures pour la manipulation de chaînes de caractères et la méthode sort.

ECMAScript 11 (ES11) - 2020

BigInt : Introduit un type pour représenter des entiers très grands.
Promise.allSettled : Une nouvelle méthode de promesse qui renvoie un tableau de résultats après que toutes les promesses données se soient résolues ou rejetées.
Dynamique import() : Importations de modules sur demande pour améliorer la performance du chargement de modules.

22

Les Polyfills

Introduction aux Polyfills

Les polyfills sont des morceaux de code (généralement JavaScript sur le web) qui fournissent des fonctionnalités modernes à des environnements qui ne les prennent pas en charge nativement. Ils permettent aux développeurs d'utiliser les nouvelles fonctionnalités du langage tout en maintenant la compatibilité avec les navigateurs plus anciens.

core-js

core-js est une bibliothèque complète de polyfills couvrant l'ensemble des fonctionnalités ECMAScript. Il est largement utilisé pour assurer la compatibilité des nouvelles fonctionnalités avec les anciennes versions de navigateurs.

```
import "core-js/stable";
import "regenerator-runtime/runtime";

// Utilisation d'une fonction moderne avec core-js
const includesExample = [1, 2, 3].includes(2);
console.log(includesExample); // true
```

Babel Polyfill

Babel est principalement un transpileur, mais il inclut aussi des polyfills pour les fonctionnalités modernes de JavaScript, utilisant souvent core-js pour ce faire.

```
// Installation des polyfills via Babel
npm install --save @babel/polyfill

// Ajout du polyfill dans le code
import "@babel/polyfill";

// Utilisation de promesses et d'autres fonctionnalités modernes
const fetchData = async () => {
  const response = await fetch('https://api.example.com/data');
  const data = await response.json();
  console.log(data);
};

fetchData();
```

Fetch Polyfill (whatwg-fetch)

Un polyfill pour l'API fetch, qui permet de faire des requêtes réseau. Utile pour les environnements ne supportant pas encore cette API.

```
// Installation du polyfill
npm install whatwg-fetch

// Ajout du polyfill dans le code
import 'whatwg-fetch';

// Utilisation de fetch
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error fetching data:', error));
```

Promise Polyfill

Ce polyfill permet d'ajouter le support de la classe Promise dans les navigateurs qui ne la prennent pas en charge nativement, assurant la compatibilité des opérations asynchrones basées sur les promesses.

```
// Installation du polyfill
npm install promise-polyfill

// Ajout du polyfill dans le code
import Promise from 'promise-polyfill';

// Utilisation de promesses
const asyncOperation = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve('Operation successful'), 1000);
  });
};

asyncOperation().then(message => console.log(message));
```

23

Symbol

Introduction aux Symboles

Les Symboles sont un type de données primitifs introduits dans ECMAScript 6 (ES6). Ils sont utilisés pour créer des identifiants uniques.

```
// Création d'un symbole
const sym1 = Symbol();
const sym2 = Symbol('description');

console.log(typeof sym1); // "symbol"
console.log(sym2); // Symbol(description)
```

Utilisation de Symboles comme Clés d'Objet

Les Symboles peuvent être utilisés comme clés pour les propriétés des objets, assurant l'unicité de ces clés.

```
const sym = Symbol('uniqueKey');
const obj = {
    [sym]: 'value'
};

console.log(obj[sym]); // "value"
console.log(Object.keys(obj)); // []
console.log(Object.getOwnPropertySymbols(obj)); // [Symbol(uniqueKey)]
```

Symbol.for et Symbol.keyFor

Symbol.for crée des symboles globaux accessibles partout dans votre code, tandis que Symbol.keyFor récupère la clé associée à un symbole global.

```
const globalSym = Symbol.for('globalSymbol');
const anotherGlobalSym = Symbol.for('globalSymbol');

console.log(globalSym === anotherGlobalSym); // true

const key = Symbol.keyFor(globalSym);
console.log(key); // "globalSymbol"
```

Symbol.iterator

Symbol.iterator est un symbole bien connu utilisé pour définir l'itérabilité des objets, permettant l'utilisation des boucles for...of.

```
const iterable = {
    [Symbol.iterator]: function* () {
        yield 1;
        yield 2;
        yield 3;
    }
};

for (const value of iterable) {
    console.log(value); // 1, 2, 3
}
```

24

Itérateurs

Introduction aux Itérateurs

Les itérateurs en JavaScript permettent de parcourir des collections de données de manière systématique. Ils fournissent une interface pour accéder aux éléments d'une collection séquentiellement, sans exposer la structure sous-jacente.

```
const array = [1, 2, 3];
const iterator = array[Symbol.iterator]();

console.log(iterator.next()); // { value: 1, done: false }
console.log(iterator.next()); // { value: 2, done: false }
console.log(iterator.next()); // { value: 3, done: false }
console.log(iterator.next()); // { value: undefined, done: true }
```

La Méthode Symbol.iterator

La méthode Symbol.iterator est utilisée pour obtenir l'itérateur d'une collection. Elle retourne un objet qui implémente l'interface d'itérateur, c'est-à-dire un objet avec une méthode next.

```
const iterable = "hello";
const iterator = iterable[Symbol.iterator]();

console.log(iterator.next().value); // "h"
console.log(iterator.next().value); // "e"
console.log(iterator.next().value); // "l"
console.log(iterator.next().value); // "l"
console.log(iterator.next().value); // "o"
console.log(iterator.next().done);  // true
```

Créer un Itérateur Personnalisé

Vous pouvez créer vos propres itérateurs en implémentant l'interface d'itérateur. Cela implique de définir une méthode next qui retourne un objet avec les propriétés value et done.

```
function createIterator(array) {
  let index = 0;
  return {
    next: function() {
      return index < array.length ?
        { value: array[index++], done: false } :
        { value: undefined, done: true };
    }
  };
}

const myIterator = createIterator([10, 20, 30]);
console.log(myIterator.next()); // { value: 10, done: false }
console.log(myIterator.next()); // { value: 20, done: false }
console.log(myIterator.next()); // { value: 30, done: false }
console.log(myIterator.next()); // { value: undefined, done: true }
```

Utilisation des Générateurs

Les générateurs sont une manière simple de créer des itérateurs en utilisant la fonction function*. Ils permettent de pauser et reprendre l'exécution de la fonction génératrice.

```
function* generator() {
  yield 1;
  yield 2;
  yield 3;
}

const gen = generator();
console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
console.log(gen.next()); // { value: 3, done: false }
console.log(gen.next()); // { value: undefined, done: true }
```

Exemple concret

Les itérateurs + générateurs peuvent être utilisés pour parcourir des pages de résultats d'une API paginée. Cela simplifie l'extraction de données d'une API qui retourne des résultats par lots (pages).

```
async function* fetchPages(url) {
  let page = 1;
  let hasNext = true;

  while (hasNext) {
    const response = await fetch(`${url}?page=${page}`);
    const data = await response.json();
    yield data.items;
    hasNext = data.hasNext;
    page++;
  }
}

(async () => {
  const url = 'https://api.example.com/items';
  for await (const items of fetchPages(url)) {
    console.log(items); // Process each page of items
  }
})();
```

Exemple concret 2

Les itérateurs peuvent être utilisés pour parcourir les pages d'un PDF, facilitant l'extraction de texte ou de contenu de chaque page.

```
const pdfjsLib = require('pdfjs-dist');

async function* pdfPageIterator(pdfUrl) {
  const loadingTask = pdfjsLib.getDocument(pdfUrl);
  const pdfDocument = await loadingTask.promise;
  const numPages = pdfDocument.numPages;

  for (let pageNum = 1; pageNum <= numPages; pageNum++) {
    const page = await pdfDocument.getPage(pageNum);
    const textContent = await page.getTextContent();
    yield textContent.items.map(item => item.str).join(' ');
  }
}

(async () => {
  const pdfUrl = 'path/to/your/document.pdf';
  for await (const pageText of pdfPageIterator(pdfUrl)) {
    console.log(pageText); // Process the text of each page
  }
})();
```

25

Les Proxys

Introduction aux Proxys en JavaScript

Un Proxy est un objet permettant d'intercepter et de redéfinir des opérations fondamentales effectuées sur un objet cible. Ils peuvent être utilisés pour valider, formater ou gérer des accès aux propriétés de l'objet.

```
const cible = {};
const proxy = new Proxy(cible, {});

console.log(proxy); // Proxy {}
```

Handler get dans un Proxy

Le handler get permet d'intercepter les accès aux propriétés de l'objet cible. On peut personnaliser le comportement lors de la lecture des propriétés.

```
const cible = { message: "Salut" };
const handler = {
  get: (target, property) => {
    return property in target ? target[property] : "Propriété inexistante";
  }
};

const proxy = new Proxy(cible, handler);
console.log(proxy.message); // Salut
console.log(proxy.inexistant); // Propriété inexistante
```

Handler set dans un Proxy

Le handler set permet d'intercepter les opérations d'écriture sur les propriétés de l'objet cible. On peut ainsi valider ou modifier les valeurs avant qu'elles ne soient assignées.

```
const cible = {};
const handler = {
  set: (target, property, value) => {
    if (typeof value === "number") {
      target[property] = value;
      return true;
    } else {
      console.log("Seuls les nombres sont acceptés");
      return false;
    }
  }
};

const proxy = new Proxy(cible, handler);
proxy.age = 30; // OK
proxy.nom = "Jean"; // Seuls les nombres sont acceptés
console.log(proxy); // { age: 30 }
```

Handler has dans un Proxy

Le handler has permet d'intercepter les opérations de test de propriétés avec l'opérateur in.

```
const cible = { visible: true };
const handler = {
  has: (target, property) => {
    if (property === "secret") {
      return false;
    }
    return property in target;
  }
};

const proxy = new Proxy(cible, handler);
console.log("visible" in proxy); // true
console.log("secret" in proxy); // false
```