JavaScript - Programmation avancée

Formateur

Hamza Zagrouba
Ingénieur informatique & Formateur confirmé
Consultant web
hamzaensi@gmail.com

CODE STAGE - Rév n°

Plan

- Présentation du formateur
- Le plan de formation
- Objectifs de la formation
- Public concerné
- Connaissances requises
- Références bibliographiques

Présentation du formateur

Hamza Zagrouba

Ingénieur développeur FullStack , Freelancer, Formateur confirmé

@hamza-zagrouba-89581115

@hamzaensi

hamzaensi@gmail.com

Table des matières

Programme

Jour 1 : **Matin**

- Maîtrise des outils de débogage (IDE et navigateurs)
- Configuration de Visual Studio Code
- Accès aux fenêtres de débogage dans Chrome, Edge, Firefox
- Travaux pratiques : utilisation des outils de débogage avec JavaScript
- Rappels importants sur JavaScript : Structures de données, portée, fonctions, objets
- Pièges à éviter

Jour 1 : **Après-midi**

Programmation Objet

- Création et constructeurs d'objets
- Valeur de "this" et prototype
- Héritage et visibilitée

Table des matières

Programme

Jour 2 : **Matin**

Travaux Pratiques

- Conception d'applications avec fonctions et objets
- Importance de la modularisation et création de modules
- Programmation fonctionnelle : fonctions anonymes, closures

Jour 2 : **Après-midi**

Structuration du Code

- Séparation en fichiers, définition de modules (AMD, Require.js)
- Qualité de code avec JSHint et JSLint
- Introduction à jQuery : bases et sélecteurs

Table des matières

Programme

Jour 3 : **Matin**

Exploitation des API HTML5

- Validation des formulaires, stockage (LocalStorage, IndexedDB)
- JSON, WebSockets, WebWorkers

Jour 3 : **Après-midi**

JavaScript et Node.js

- Différences entre les langages, asynchronisme
- REST avec Node.js et JavaScript
- Évolutions récentes : ECMAScript 6, 7 (classes, modules, promises)

Objectifs pédagogiques

Pourquoi ?

- ✓ Utiliser tous les outils de débogage à disposition
- ✓ Décrire les contextes d'exécution
- ✓ Structurer le code JavaScript en modules
- ✓ Implémenter les concepts objets en JavaScript et les concepts fonctionnels
- ✓ Identifier les aspects avancés des "closures" et les promises
- ✓ Mémoriser jQuery
- ✓ Identifier les différences avec Node.js et expliquer le rôle de chacun.

Public concerné

Pour qui ?

•**Développeurs** : Ce programme est conçu pour ceux qui veulent approfondir leurs compétences en JavaScript, notamment dans le débogage et la programmation orientée objet.

•**Architectes techniques** : Les architectes bénéficieront des meilleures pratiques en structuration de code et conception de solutions modulaires.

•**Chefs de projets techniques** : La formation les aidera à mieux coordonner les équipes de développement et aligner les choix techniques avec les objectifs du projet.

Connaissance requise

Pré-requis?

Il est nécessaire d'avoir suivi le cours "JavaScript - Fondamentaux" pour aborder les concepts avancés.

Connaissances pratiques : Une expérience préalable en JavaScript est indispensable,
• comprenant l'utilisation des fonctions, et structures de données de base.

Bases solides : Les participants doivent être à l'aise avec les syntaxes et concepts
•fondamentaux de JavaScript pour tirer pleinement parti de cette formation avancée.

Références bibliographiques

Livres

"You Don't Know JS (Yet): Scope & Closures" - Kyle Simpson

"Eloquent JavaScript: A Modern Introduction to Programming" - Marijn Haverbeke

"JavaScript: The Good Parts" - Douglas Crockford

Articles

MDN Web Docs (Mozilla Developer Network)
•URL : https://developer.mozilla.org

JavaScript.info
•URL : https://javascript.info

ECMAScript 6 Features
•URL : https://es6-features.org

Maîtrise des fonctions de "debug" dans les IDE et navigateurs

Le **débogage** est le processus d'identification et de résolution des erreurs dans le code.

Types d'Erreurs à Déboguer

1- **Erreurs de syntaxe** : Elles surviennent lorsque le code ne respecte pas la grammaire du langage. Ces erreurs sont souvent détectées au moment de la compilation ou lors de l'exécution immédiate du code.

```
console.log("Hello World" // Manque une parenthèse fermante
```

2- **Erreurs de référence (ReferenceError) :** Ces erreurs apparaissent lorsqu'une variable ou une fonction est utilisée sans être définie.

```
console.log(nonDefini); // La variable nonDefini n'existe pas
```

3- **Erreurs de type (TypeError)** : Elles surviennent lorsqu'une opération est effectuée sur une valeur d'un type inapproprié.

```
let nombre = 42;
nombre.toUpperCase(); // Impossible d'appeler toUpperCase sur un nombre
```

4- **Erreurs de logique**: Ces erreurs ne génèrent pas forcément de messages d'erreur mais conduisent à des résultats inattendus à cause d'une erreur dans la logique du programme.

```
let total = 100;
let reduction = 0.1;
let prixFinal = total - reduction; // Mauvaise logique, le calcul devrait être tot
```

5-**Erreurs d'exécution (RuntimeError)** : Elles se produisent pendant l'exécution du programme, souvent à cause de l'environnement ou des données inattendues.

```
let tableau = [1, 2, 3];
console.log(tableau[10]); // Essai d'accès à un index qui n'existe pas dans le tab
```

6- **Erreurs d'asynchronisme :** Ces erreurs apparaissent souvent lors de la gestion des opérations asynchrones (comme les promesses) lorsqu'elles ne sont pas correctement gérées.

```
fetch('http://api.exemple.com')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log(erreur)); // Typo dans "erreur", mauvaise gestion de
```

Comment les Déboguer :

•**Console** : Utiliser **console.log()** pour afficher la valeur des variables à différents points du code.

•**Utilisation des breakpoints** : Placer des points d'arrêt (breakpoints) dans le code pour examiner l'état des variables et des fonctions.

•**Outils de débogage** : Utiliser les DevTools des navigateurs ou le débogueur d'un IDE comme Visual Studio Code pour suivre l'exécution du code pas à pas et extensions.

Différentes façons de déboguer du JavaScript :

Console Log : Utiliser console.log() pour afficher des valeurs et des messages à différentes étapes du code.

Débogueur du Navigateur : Utiliser les outils de développement intégrés dans les navigateurs (comme Chrome DevTools) pour placer des points d'arrêt, inspecter des variables et suivre l'exécution du code.

Alertes : Utiliser alert() pour afficher des messages simples, bien que cela soit moins pratique que les autres méthodes.

Try/Catch : Encapsuler du code dans un bloc try/catch pour capturer et gérer les erreurs sans interrompre l'exécution du programme.

Différentes façons de déboguer du JavaScript :

Linting : Utiliser des outils de linting comme ESLint pour détecter des erreurs de syntaxe et des problèmes de style avant l'exécution du code.

Tests Unitaires : Écrire des tests unitaires pour vérifier le comportement de petites parties de code et faciliter le débogage.

Éditeurs de Code : Utiliser des éditeurs de code avec des intégrations de débogage (comme Visual Studio Code) pour une expérience de débogage plus fluide.

Debugging Remote : Déboguer des applications web exécutées sur des appareils mobiles ou des environnements distants via des outils comme Chrome Remote Debugging.
Logger : Utiliser des bibliothèques de logging pour capturer des informations plus détaillées sur l'exécution de votre application.

Maîtrise des fonctions de "debug" dans les IDE et navigateurs

Pratique :

- **Débogage avec :**
- **IDE :** VS code
- **Navigateurs Devtools :** chrome , Edge , FireFox
- **Extension :** JavaScript Debugger (Nightly)

Éléments de premier ordre :

Définition :En JavaScript, les éléments de premier ordre sont les entités qui peuvent être manipulées comme des variables. Les fonctions et objets sont des éléments de premier ordre, ce qui signifie qu'ils peuvent être assignés à des variables, passés comme arguments à des fonctions, et retournés par des fonctions.

Exemple :

```
let fonction = function() {
  return "Hello!";
};
console.log(fonction()); // Output: Hello!
```

Ici, la fonction est assignée à une variable et appelée comme une fonction ordinaire.

Portée des données:

Définition : La portée (ou scope) détermine l'accessibilité des variables dans différentes parties du programme. En JavaScript, il existe deux types de portée :Portée globale : Les variables déclarées en dehors de toute fonction sont accessibles partout dans le script.Portée locale : Les variables déclarées à l'intérieur d'une fonction ne sont accessibles que dans cette fonction.

Exemple :

```
let globalVar = "Global";

function test() {
  let localVar = "Local";
  console.log(globalVar); // Accessible
  console.log(localVar);  // Accessible
}

test();
console.log(localVar); // Erreur : localVar n'est pas définie dans le contexte global
```

Fonctions et Objets : Choisir selon les besoins:

•**Définition** :
•Les **fonctions** et **objets** sont des éléments centraux de JavaScript. Les **fonctions** sont utilisées pour encapsuler du code réutilisable, tandis que les **objets** servent à regrouper des données sous forme de paires clé-valeur.
•**Quand utiliser quoi ?** :
•Utilisez une **fonction** lorsque vous avez besoin d'exécuter un ensemble d'instructions.
•Utilisez un **objet** pour regrouper des données associées et les manipuler ensemble.

```
let voiture = {
  marque: "Toyota",
  modele: "Corolla",
  demarrer: function() {
    return "La voiture démarre";
  }
};

console.log(voiture.demarrer()); // Output: La voiture démarre
```

Pratique : **branche : rappel**

- **Exercice 1 : Function**
- **Exercice 2 : Object**
- **Exercice 3 : Variables**

Typage faible

JavaScript est un langage à typage dynamique et faible
Les types sont convertis automatiquement, ce qui peut causer des comportements inattendus.

Exemple :

```
console.log(5 + "5"); // "55"
console.log(5 - "5"); // 0
```

•Les opérations peuvent être ambiguës (concaténation vs. addition).

•Soyez vigilant aux conversions implicites de type.

Le "Hoisting" en JavaScript

•Le hoisting remonte les déclarations de variables et de fonctions en haut de leur scope.
•Les variables déclarées avec var sont déclarées mais non initialisées.

Exemple :

```
console.log(x); // undefined
var x = 5;

console.log(y); // Erreur de référence
let y = 10;
```

•Les déclarations let et const ne subissent pas de hoisting de la même manière.

•Le hoisting peut entraîner des comportements inattendus si mal compris.

Contexte et Variables Globales

•Les variables globales peuvent être accidentellement modifiées et provoquer des conflits.
•JavaScript recherche les variables dans leur scope, puis dans les scopes parents jusqu'au global.

Exemple :

```
var globalVar = "Global";

function example() {
  globalVar = "Local"; // Modifie la variable globale
}
```

•Utilisez **let** et **const** pour limiter le scope des variables.

•Évitez de polluer l'espace global.

Changement de Contexte en JavaScript

•La valeur de this dépend de comment une fonction est appelée.

•En dehors d'une fonction, **this** se réfère à l'objet global (en mode non strict).

Exemple :

```
function showThis() {
  console.log(this);
}

showThis(); // Objet global (ou undefined en mode strict)

const obj = { showThis };
obj.showThis(); // Réfère à obj
```

•**bind**(), **call**(), et **apply**() permettent de contrôler explicitement le contexte de **this**.

•Attention aux changements de contexte dans les **callbacks**.

Pratique : sur La branche « piege »

Exercice 1

Exercice 2

Exercice 3

Exercice 4

Summary 1

- Comprendre l'utilisation de **console.log** pour déboguer rapidement.
- Se familiariser avec le **débogueur intégré** de Visual Studio Code pour une analyse plus approfondie.
- Utiliser une **extension** de débogage pour lancer et examiner votre code directement dans le navigateur ou le serveur.

- **Éléments de premier ordre** : Fonctions et objets manipulables comme des variables.
- **Portée des données** : Variables accessibles selon leur contexte (global ou local).
- **Fonctions et Objets** : Choisir selon les besoins de manipulation de données.
- Les pièges du langage JavaScript :Typage faible, Hoisting, Contexte et variables globales, Changement de contexte

Programmation objet en JS

Programmation orientée objet (POO) en JavaScript:

La programmation orientée objet (POO) est un paradigme qui utilise des objets pour structurer des programmes. Les objets contiennent des propriétés (données) et des méthodes (fonctions), permettant de modéliser des entités du monde réel.

```
let voiture = {
  marque: "Toyota",
  modele: "Corolla",
  demarrer: function() {
    return "La voiture démarre";
  }
};
```

Différentes façons de créer des objets

En JavaScript, il existe plusieurs façons de créer des objets :

Littéral d'objet : Créer directement un objet avec des accolades {}.

Object.create() : Créer un nouvel objet en héritant d'un autre.

Constructeur personnalisé : Utiliser une fonction pour construire des objets.

```
// Littéral d'objet
let personne1 = { nom: "Alice" };

// Object.create()
let personne2 = Object.create(personne1);

// Fonction constructeur
function Personne(nom) {
  this.nom = nom;
}
let personne3 = new Personne("Bob");
```

Les constructeurs en JavaScript

Un constructeur est une fonction utilisée pour créer des objets avec des propriétés et des méthodes prédéfinies. Le mot-clé new est utilisé pour instancier un objet à partir d'un constructeur.

```
function Voiture(marque, modele) {
  this.marque = marque;
  this.modele = modele;
}

let maVoiture = new Voiture("Honda", "Civic");
console.log(maVoiture.marque); // Output: Honda
```

Valeur de "this" dans un objet

Le mot-clé **this** fait référence à l'objet courant dans lequel une méthode est appelée. Sa valeur dépend de la façon dont la fonction est invoquée.

```
let voiture = {
  marque: "Toyota",
  demarrer: function() {
    console.log(this.marque); // "this" fait référence à l'objet "voiture"
  }
};

voiture.demarrer(); // Output: Toyota
```

Prototype et __proto__ en JavaScript

Chaque objet JavaScript a une propriété cachée appelée [[**Prototype**]] (accessible via **__proto__**), qui permet l'héritage de méthodes et propriétés d'autres objets. Le prototype est un modèle à partir duquel les objets peuvent hériter.

```
function Personne(nom) {
  this.nom = nom;
}

Personne.prototype.saluer = function() {
  return `Bonjour, je m'appelle ${this.nom}`;
};

let personne1 = new Personne("Alice");
console.log(personne1.saluer()); // Output: Bonjour, je m'appelle Alice
```

Ici, **Personne.prototype** définit une méthode saluer que tous les objets créés via Personne peuvent utiliser.

Héritage avec ES6 (Classes)

En ES6, l'héritage en JavaScript est simplifié grâce à l'introduction des classes. Une classe peut hériter d'une autre classe à l'aide du mot-clé extends, ce qui permet de réutiliser les propriétés et méthodes d'une classe parent dans une classe enfant.

```
// Classe parent
class Animal {
  constructor(nom) {
    this.nom = nom;
  }

  parler() {
    return `${this.nom} fait un bruit.`;
  }
}
```

```
// Classe enfant
class Chien extends Animal {
  constructor(nom, race) {
    super(nom); // Appel du constructeur de la classe parent
    this.race = race;
  }

  parler() {
    return `${this.nom} (un ${this.race}) aboie.`;
  }
}

let monChien = new Chien("Rex", "Labrador");
console.log(monChien.parler()); // Output: Rex (un Labrador) aboie.
```

Visibilité

- **Accessible depuis l'extérieur de la classe :** Les propriétés et méthodes publiques peuvent être appelées ou modifiées depuis n'importe quel endroit, même en dehors de la classe où elles sont définies.

- **Par défaut**, toutes les propriétés et méthodes en JavaScript sont publiques si elles sont définies sans un mécanisme spécial pour les rendre privées.

```
class Voiture {
    constructor(marque, modele) {
        this.marque = marque; // Propriété publique
        this.modele = modele; // Propriété publique
    }

    demarrer() { // Méthode publique
        console.log(`${this.marque} ${this.modele} démarre.`);
    }
}
```

Visibilité

•**Accessible uniquement depuis l'intérieur de la classe :** Les propriétés et méthodes privées ne peuvent être utilisées que dans le contexte de la classe où elles sont définies. Elles sont cachées de l'extérieur et protègent ainsi les données sensibles.

•En JavaScript moderne (depuis **ES6+**), vous pouvez définir des propriétés privées avec un **préfixe** **#**.

```
class CompteBancaire {
    constructor(solde) {
        this.#solde = solde; // Propriété privée
    }

    #afficherSolde() { // Méthode privée
        console.log(`Le solde est : ${this.#solde}€`);
    }

    deposer(montant) {
        if (montant > 0) {
            this.#solde += montant;
            this.#afficherSolde();
        }
    }
}
```

```
const monCompte = new CompteBancaire(100);
monCompte.deposer(50); // Affiche : Le solde est : 150€
console.log(monCompte.solde); // Erreur : `solde` est privé
monCompte.#afficherSolde(); // Erreur : `afficherSolde` est privé
```

Pratique : branche poo

Exercice 1

Exercice 2

Exercice 3

Exercice 4 : visibilité

Summary 2

Programmation Objet en JavaScript

- **Création d'objets** : Littéraux, constructeurs, classes.
- **this dans un objet** : Réfère à l'instance actuelle selon le contexte.
- **Prototype et héritage** : Utilisation de prototype et __proto__ pour partager des propriétés.
- **Héritage** : Classes ES6, fonctions constructrices, et prototypes.
- **Visibilité** : Gérer l'accès aux propriétés (privées/public).

Ces concepts permettent une manipulation avancée et une meilleure structuration des objets en JavaScript.

Structuration et qualité du code

Définition :La structuration du code consiste à organiser les fichiers et les modules de manière logique et maintenable. Cela améliore la lisibilité, facilite les mises à jour et réduit les erreurs.

Exemple :Séparer les fonctionnalités en **modules** ou **fichiers** distincts rend le projet plus facile à comprendre.
Par exemple, un fichier pour la **logique**, un autre pour les **interactions** avec l'**API**, etc.

Séparation en multiple fichiers

Définition :La séparation du code en plusieurs fichiers permet de mieux organiser le projet, de réduire la taille des fichiers individuels, et de rendre chaque fichier plus spécialisé dans une fonctionnalité précise.

Exemple :

Projet JavaScript simple :

- **app.js** pour la logique principale
- **data.js** pour la gestion des données
- **ui.js** pour la manipulation de l'interface utilisateur

Définition de modules

Définition : Les modules sont des blocs de code autonomes qui peuvent être importés ou exportés d'un fichier à un autre, favorisant la réutilisabilité et la maintenabilité du code.

En JavaScript (ES6), un module peut être exporté et importé ainsi :

```
// data.js
export const getData = () => { return [1, 2, 3]; };

// app.js
import { getData } from './data.js';
console.log(getData());
```

Asynchronous Module Definition (AMD)

Définition : AMD est un standard pour définir et charger des modules JavaScript de manière asynchrone. Il est souvent utilisé dans les navigateurs pour améliorer les performances en permettant le chargement parallèle des fichiers.

```
define(['dependency'], function(dependency) {
  return {
    action: function() {
      dependency.doSomething();
    }
  };
});
```

AMD avec Require.js

Définition :**Require.js** est une bibliothèque JavaScript qui implémente AMD pour charger des modules de manière asynchrone et gérer les dépendances entre eux.

```
require(['moduleA', 'moduleB'], function (moduleA, moduleB) {
  moduleA.doSomething();
  moduleB.doSomethingElse();
});
```

Utilisation de Require.js pour charger des modules

Impacts des closures sur la lisibilité

Définition :Une closure est une fonction qui retient les variables de son environnement lexical, même après que cette fonction soit exécutée.
Bien que très puissante, une utilisation excessive peut réduire la lisibilité du code.

```
function outerFunction(outerVariable) {
  return function innerFunction(innerVariable) {
    console.log(outerVariable, innerVariable);
  };
}
const newFunction = outerFunction('outside');
newFunction('inside'); // Affiche 'outside inside'
```

Qualité avec JSHint et JSLint

Définition : **JSHint** et **JSLint** sont des outils de linting utilisés pour analyser le code JavaScript à la recherche d'erreurs potentielles, de mauvaises pratiques ou de violations de conventions de style.

Exemple :

Lancer **JSHint** sur un fichier : Commande : **jshint app.js**

Résultat : JSHint affiche les erreurs ou les avertissements liés à la qualité du code, comme les variables non définies ou les erreurs de syntaxe.

Pratique :

Application 1

Application 2

Application 3

Summary 3

- **Séparation en plusieurs fichiers** : Améliorer la lisibilité et la maintenance du code.
- **Définition de modules** : Encapsuler les fonctionnalités dans des modules bien définis.
- **Asynchronous Module Definition (AMD)** : Gérer les dépendances de manière asynchrone avec Require.js.
- **Impacts des closures** : Utiliser avec précaution pour maintenir la lisibilité du code.
- **Qualité du code** : Utiliser des outils comme JSHint et JSLint pour détecter les erreurs et améliorer la qualité.

Bases de jQuery

Définition :
jQuery est une bibliothèque JavaScript légère et rapide qui simplifie la manipulation du DOM, la gestion des événements, l'animation et les requêtes AJAX.
Elle permet d'écrire moins de code pour obtenir des résultats similaires à ceux de JavaScript pur.

Syntaxe de base :
Le symbole $ est utilisé pour sélectionner des éléments dans le DOM, par exemple :

```
$(document).ready(function() {
  // Code jQuery ici
});
```

```
$('#element').hide();
```

Exploitation des sélecteurs en jQuery

Définition :
jQuery utilise des sélecteurs CSS pour sélectionner et manipuler des éléments HTML. Les sélecteurs permettent de cibler un ou plusieurs éléments du DOM pour les modifier ou interagir avec eux.

Principaux sélecteurs :
Par **ID** : **$('#id')** sélectionne un élément avec un ID spécifique.
Par **classe** : **$('.classe')** sélectionne tous les éléments avec une classe donnée.
Par **type d'élément** : **$('div')** sélectionne tous les éléments .

```
$('#element').hide();
```

Intérêts de jQuery par rapport à JavaScript

Avantages :

- Compatibilité entre navigateurs : jQuery gère automatiquement les différences entre les navigateurs.

- Syntaxe simplifiée : Des tâches comme l'animation et les requêtes AJAX sont beaucoup plus faciles à écrire.

- Performance : Bien que JavaScript pur puisse être plus rapide, jQuery offre des solutions optimisées pour des tâches courantes.

```
document.getElementById("bouton").addEventListener("click", function() {
  alert("Cliquez !");
});
```

```
$('#bouton').click(function() {
  alert("Cliquez !");
});
```

Les composants graphiques de jQuery

jQuery propose des composants et des effets visuels pour améliorer l'interaction avec l'utilisateur. Il permet de créer facilement des animations, des transitions et d'intégrer des widgets interactifs.

Exemples de composants graphiques :

Effets de glissement : Pour faire apparaître un élément de façon fluide.

```
$('#element').slideDown();
```

Animations personnalisées : Vous pouvez animer presque tous les attributs CSS.

```
$('#element').animate({ opacity: 0.5, height: '50px' });
```

Animation : Ajouter un effet de fondu à un élément

```
$('#element').fadeOut();
```

Pratique :

Application 1

Application 2

Application 3

Summary 4

- **Sélecteurs en jQuery** : Utiliser des sélecteurs puissants pour cibler des éléments du DOM facilement.
- **Avantages de jQuery** : Simplification de la manipulation du DOM, gestion des événements, et effets d'animation par rapport à JavaScript natif.
- **Composants graphiques** : Utilisation de plugins et d'éléments d'interface utilisateur (UI) fournis par jQuery pour améliorer l'expérience utilisateur.

Validation des formulaires en JavaScript

La validation des formulaires en JavaScript permet de vérifier que les données saisies par l'utilisateur respectent certains critères (par exemple : format d'email, champs obligatoires) avant leur soumission au serveur.

Exemple :Validation d'un champ email

```
function validerFormulaire() {
  let email = document.getElementById("email").value;
  let regex = /\S+@\S+\.\S+/;
  if (!regex.test(email)) {
    alert("Veuillez entrer une adresse email valide.");
    return false;
  }
  return true;
}
```

La Solutions de stockage (LocalStorage et IndexedDB)

Définition :

LocalStorage : Permet de stocker des données sous forme de paires clé-valeur dans le navigateur de manière persistante.

IndexedDB : Une base de données NoSQL dans le navigateur qui permet de stocker des données structurées avec des index pour des recherches rapides.

Exemples:

```
localStorage.setItem("nom", "Jean");
console.log(localStorage.getItem("nom")); // "Jean"
```

```
let request = indexedDB.open("MaBaseDeDonnees", 1);
request.onsuccess = function(event) {
  console.log("Base de données ouverte avec succès");
};
```

JSON (JavaScript Object Notation)

Définition: **JSON** est un format léger de transfert de données, utilisé pour échanger des données entre un client (navigateur) et un serveur. Il s'agit d'un format basé sur du texte qui est facile à lire et à écrire pour les humains et simple à analyser par les machines.

Exemple : Conversion d'un objet JavaScript en JSON :

```
let utilisateur = { nom: "Jean", age: 30 };
let jsonUtilisateur = JSON.stringify(utilisateur);
console.log(jsonUtilisateur); // {"nom":"Jean","age":30}
```

WebSockets

Définition : Les WebSockets permettent d'établir une connexion bidirectionnelle en temps réel entre le navigateur et un serveur. Contrairement à HTTP, une connexion WebSocket reste ouverte, ce qui permet d'envoyer et recevoir des messages en continu.

Exemple :Connexion à un serveur **WebSocket** :

```
let socket = new WebSocket('ws://example.com/socket');
socket.onopen = function(event) {
  console.log("Connexion ouverte");
  socket.send("Bonjour serveur !");
};
socket.onmessage = function(event) {
  console.log("Message reçu : " + event.data);
};
```

WebWorkers

Définition : Les WebWorkers permettent d'exécuter des scripts JavaScript en arrière-plan, indépendamment de l'interface utilisateur principale. Cela permet de gérer des tâches lourdes sans bloquer l'exécution de l'interface.

Exemple :Création d'un WebWorker

```
let worker = new Worker('worker.js');
worker.postMessage('Démarrer le calcul');

worker.onmessage = function(event) {
  console.log("Résultat du calcul : " + event.data);
};
```

Contenu du fichier **worker.js** :

```
onmessage = function(event) {
  let resultat = calculLourd();
  postMessage(resultat);
};
function calculLourd() {
  // Longue tâche
  return 42;
}
```

Application : Créez une page qui effectue un calcul complexe en utilisant un WebWorker pour ne pas bloquer l'interface utilisateur.

Pratique :

Application 1

Application 2

Application 3

Summary 5

- **Validation des formulaires** : Utiliser JavaScript pour valider les données des formulaires avant l'envoi.
- **Solutions de stockage** :
  - **LocalStorage** : Stockage de données simples sous forme de paires clé/valeur dans le navigateur.
  - **IndexedDB** : Base de données orientée objet pour le stockage de grandes quantités de données.
- **JSON** : Format léger pour l'échange de données entre le client et le serveur.
- **WebSockets** : Permettre la communication bidirectionnelle en temps réel entre le client et le serveur.
- **Web Workers** : Exécuter des scripts en arrière-plan pour éviter de bloquer le thread principal.

Différences dans les langages

JavaScript : Langage de programmation principalement exécuté dans les navigateurs pour interagir avec les pages web (côté client).

Node.js : Environnement d'exécution qui permet d'exécuter du JavaScript côté serveur, en dehors du navigateur.

Différences :

Exécution : JavaScript s'exécute dans le navigateur, tandis que Node.js s'exécute sur un serveur.

Modules : JavaScript utilise des modules ES6 import/export, tandis que Node.js utilise require pour inclure des modules.

API : Node.js dispose d'API serveur (comme le module fs pour la gestion des fichiers) qui ne sont pas disponibles dans JavaScript du navigateur.

Différences dans les langages

Exemple

JavaScript côté client :

```
document.getElementById('myButton').addEventListener('click', () => {
  alert('Bouton cliqué!');
});
```

Node.js côté serveur

```
const fs = require('fs');
fs.readFile('fichier.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

Asynchronisme en JavaScript et Node.js

Asynchronisme : L'exécution de tâches en arrière-plan sans bloquer l'exécution du code principal. Utilisé pour gérer les opérations comme les requêtes réseau ou les interactions avec une base de données.

Méthodes principales :

Callbacks : Fonctions passées en argument qui sont exécutées lorsque l'opération asynchrone se termine.

Promises : Objet représentant une opération asynchrone qui peut réussir ou échouer.

Async/await : Syntaxe simplifiée pour gérer les promesses.

Asynchronisme en JavaScript et Node.js

Exemples:

Callbacks :

```
setTimeout(() => {
  console.log('Opération asynchrone terminée!');
}, 1000);
```

Promises :

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

Async/await :

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.log(error);
  }
}
fetchData();
```

REST Serveur en Node.js et REST Client en JavaScript

Définition :**REST (Representational State Transfer)** : Architecture qui permet la communication entre des systèmes via HTTP, souvent utilisée pour des API web.

REST Serveur : Un serveur qui expose des endpoints (routes HTTP) pour permettre au client d'interagir avec les données.

REST Client : Un client qui consomme ces API pour obtenir ou envoyer des données.

REST Serveur en Node.js et REST Client en JavaScript

Exemple REST Serveur en Node.js :

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

app.get('/api/data', (req, res) => {
  res.json({ message: 'Données du serveur' });
});

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

Exemple REST Client en JavaScript :

```
fetch('http://localhost:3000/api/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log('Erreur:', error));
```

Présentation d'une solution Web 100% JavaScript

Une solution web 100% JavaScript signifie qu'à la fois le **frontend** (client) et le **backend** (serveur) sont entièrement développés avec JavaScript. Cela inclut l'utilisation de Node.js côté serveur et de frameworks JavaScript comme React ou Vue.js côté client.

Avantages :

- Un seul langage pour le développement complet (frontend et backend).
- Réduction des coûts d'apprentissage et des transferts de compétences.
- Possibilité de réutiliser des modules et des bibliothèques entre le client et le serveur.
- **Exemple d'Architecture** :Frontend : React.js Backend : Node.js

Classes en ECMAScript 6

Définition : Les classes en JavaScript introduisent une syntaxe simplifiée pour la création d'objets et d'héritages, tout en s'appuyant sur le prototype JavaScript existant.

Avantage : Syntaxe plus claire et proche des autres langages orientés objet.

```
class Personne {
  constructor(nom, age) {
    this.nom = nom;
    this.age = age;
  }

  parler() {
    console.log(`Je m'appelle ${this.nom} et j'ai ${this.age} ans.`);
  }
}

const personne1 = new Personne('Alice', 30);
personne1.parler(); // "Je m'appelle Alice et j'ai 30 ans."
```

Modules en ECMAScript 6

Définition : Les modules permettent de diviser le code en fichiers séparés et de les importer/exporter selon les besoins, améliorant ainsi la réutilisabilité et la modularité.

Fichier math.js

```
export function addition(a, b) {
  return a + b;
}
```

Fichier main.js

```
import { addition } from './math.js';
console.log(addition(2, 3)); // 5
```

Avantage : Encapsulation de fonctionnalités, réduction des conflits de variables.

Les Fonctions Fléchées (Arrow Functions)

Définition : Les fonctions fléchées offrent une syntaxe plus concise pour écrire des fonctions anonymes. Elles ne lient pas leur propre this, ce qui est utile pour les callbacks.

```
// Fonction classique
const addition = function(a, b) {
  return a + b;
};

// Fonction fléchée
const additionFlèche = (a, b) => a + b;

console.log(additionFlèche(3, 4)); // 7
```

Avantage : Syntaxe courte et plus claire, surtout dans les fonctions de rappel (callbacks)

Promises en ECMAScript 6

Définition : Les Promises sont une façon de gérer l'asynchronisme en JavaScript, permettant d'exécuter du code une fois que des opérations asynchrones (comme des appels API) sont terminées.

Avantage : Gestion plus propre des opérations asynchrones, remplaçant les anciens "callback hells".

```
const promesse = new Promise((resolve, reject) => {
  let réussite = true;
  if (réussite) {
    resolve("Opération réussie !");
  } else {
    reject("Échec de l'opération.");
  }
});

promesse.then(message => console.log(message))
        .catch(erreur => console.log(erreur));
```

Nouvelles méthodes de Object

Définition : ECMAScript 6 introduit plusieurs nouvelles méthodes sur les objets pour simplifier la manipulation et l'inspection des objets.

Object.assign() : Copie les propriétés d'un ou plusieurs objets dans un objet cible.

```
const obj1 = { a: 1 };
const obj2 = { b: 2 };
const obj3 = Object.assign({}, obj1, obj2);
console.log(obj3); // { a: 1, b: 2 }
```

Object.entries() : Retourne un tableau de paires clé/valeur des propriétés énumérables d'un objet.

```
const obj = { a: 1, b: 2 };
console.log(Object.entries(obj)); // [['a', 1], ['b', 2]]
```

Avantage : Manipulation plus simple et efficace des objets.

Pratique :

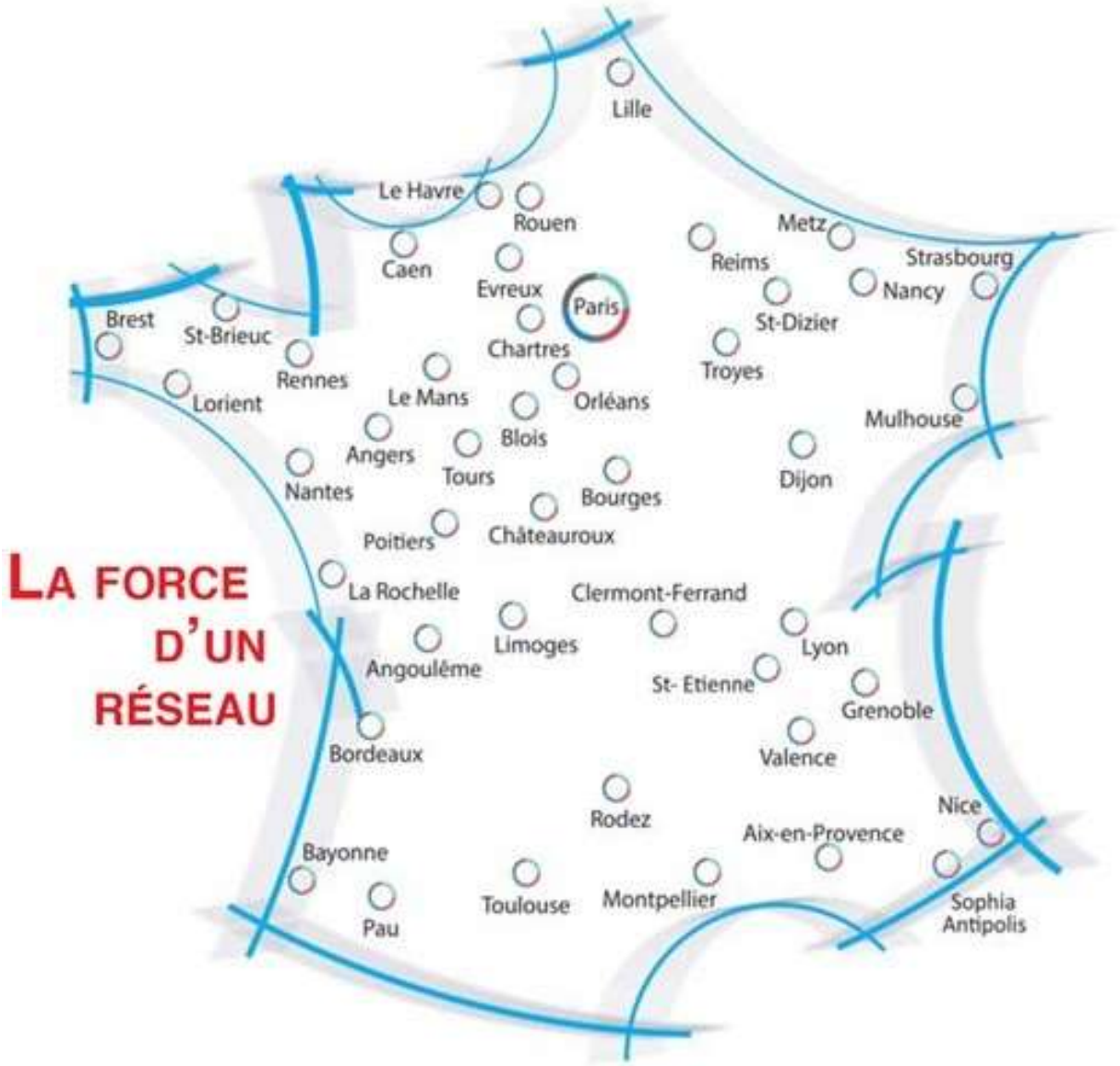
Application 1

Application 2

Application 3

Summary 6

- **Différences dans les langages** : JavaScript est principalement exécuté dans le navigateur, tandis que Node.js permet l'exécution côté serveur.
- **Asynchronisme** : Node.js utilise un modèle non-bloquant pour gérer les opérations d'E/S, permettant une meilleure performance pour les applications réseau.
- **REST serveur en Node.js** : Création d'API RESTful pour gérer les requêtes et les réponses.
- **REST client en JavaScript** : Utilisation de JavaScript pour consommer des API REST, souvent à travers des requêtes AJAX.



N°Azur 0 810 007 689

PRIX D'UN APPEL LOCAL DEPUIS UN POSTE FIXE

Découvrez également l'ensemble des stages à votre disposition sur notre site

http://www.m2iformation.fr