JS ASYNCHRONE

Steve Lebleu - 2CE-X75-A - 2024-2025

Konfer.

INTRODUCTION

"L'asynchronisme désigne le caractère de ce qui ne se passe pas à la même vitesse, que ce soit dans le temps ou dans la vitesse proprement dite, par opposition à un phénomène synchrone."

Les applications web sont confrontées à des problématiques de délais:

- Requêtes HTTP
- Requêtes DB
- Opérations *file system*
- ...

Sans optimisation dans la gestion de ce type de problématique, une application web ne peut pas être performante.

La programmation **asynchrone** apporte des solutions aux problématiques liées à l'exécution de tâches coûteuses en délais d'exécution.

Elle permet à un programme de démarrer une tâche potentiellement longue tout en restant réactif à d'autres événements.

Javascript est, par nature, un langage **événementiel** qui gère merveilleusement bien la **programmation asynchrone** !

- Modèle de programmation asynchrone
- Boucle d'événement (aka *event loop*)
- Gestionnaires d'événements

Les implications de ce caractère événementiel sont:

- Une réactivité accrue
- Des exécutions non-bloquantes
- Une complexité parfois plus élevée

EVENT LOOP

L'**Event Loop** est un mécanisme fondamental en Javascript et en Node.js qui, permet de gérer les **opérations asynchrones**.

L'implémentation varie sensiblement côté serveur (Libuv) et côté navigateur (V8), mais le principe reste le même.

NODE.JS.

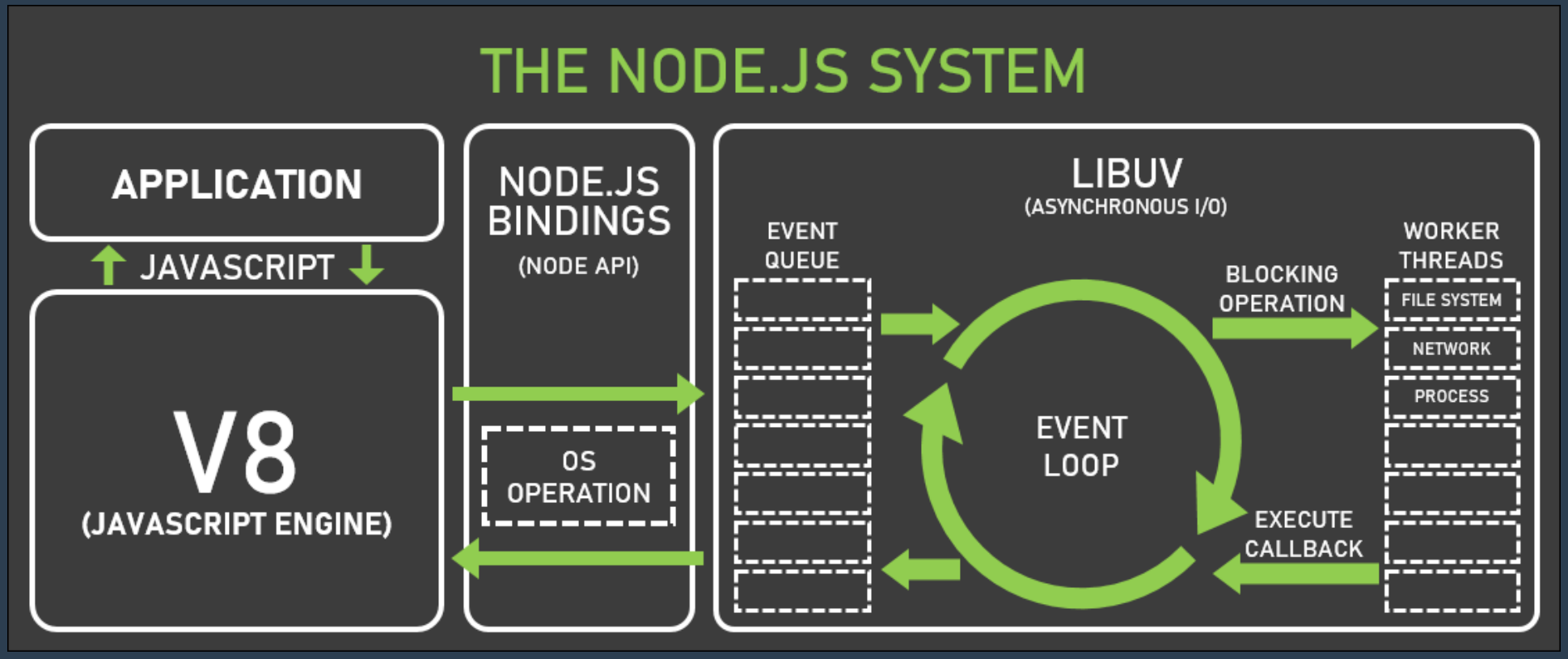
Node.js est basé sur un modèle d'exécution non-bloquant, ce qui signifie qu'il peut **gérer plusieurs opérations en parallèle sans bloquer le thread principal**.

Ce gain de performances est l'atout majeur de Node.js par rapport à d'autres technologies.

C'est plus particulièrement vrai concernant les opérations **I/O**, comme les requêtes réseau ou les opérations sur le système de fichiers.

Et c'est possible grâce à ... l'event loop !

FONCTIONNEMENT


THE NODE.JS SYSTEM
APPLICATION
JAVASCRIPT
V8
(JAVASCRIPT ENGINE)
NODE.JS
BINDINGS
(NODE API)
OS
OPERATION
LIBUV
(ASYNCHRONOUS I/O)
EVENT
QUEUE
EVENT
LOOP
BLOCKING
OPERATION
EXECUTE
CALLBACK
WORKER
THREADS
FILE SYSTEM
NETWORK
PROCESS

L'Event Loop fonctionne en boucle infinie, vérifiant constamment s'il y a des tâches à exécuter dans la file d'attente.

Chaque événement est mis dans une file d'attente, puis transmis en **Input** aux worker threads, qui l'exécute sans impacter le cycle de la boucle d'événements.

Lorsqu'une tâche est terminée, la fonction de **callback** qui lui est associée est exécutée, et le résultat est renvoyé en **Output** via la boucle événementielle.

PHASES

- **Timers**: exécution des callbacks de *setTimeout* et *setInterval*.
- **I/O Callbacks**: exécution des callbacks des opérations d'I/O.
- **Idle, prepare**: utilisé en interne par Node.js.
- **Poll**: récupération des nouvelles I/O.
- **Check**: exécution des callbacks de *setImmediate*.
- **Close Callbacks**: exécution des callbacks de fermeture, comme *socket.on('close')*.

EXEMPLE

```
console.log('Début');

setTimeout(() => {
  console.log('Timeout');
}, 0);

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

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

Dans cet exemple, l'ordre d'exécution sera:

1. Début
2. Fin
3. Immediate
4. Timeout

Le **concept fondamental** qui rend possible ce type d'implémentation, c'est celui de **callback**.

CALLBACKS

Méthode traditionnelle pour gérer les opérations asynchrones en JavaScript.

```
function fetchData(callback) {
  setTimeout(() => {
    callback('Données récupérées!');
  }, 1000);
}

fetchData(data => {
  console.log(data); // Affiche 'Données récupérées!'
});
```

Leur utilisation excessive peut conduire à un phénomène appelé "callback hell", où le code devient difficile à lire et à maintenir.

C'est toujours une technique très utilisée, mais qui s'est considérablement réduite à un usage raisonnable, principalement grâce aux *Promises* et à l'API *async/await*.

FONCTIONS NATIVES

SETTIMEOUT

La fonction *setTimeout* est utilisée pour exécuter une fonction après un certain délai, en millisecondes.

```
setTimeout(() => {
  console.log('Fonction exécutée après 1 seconde');
}, 1000);
```

SETINTERVAL

La fonction *setInterval* est utilisée pour exécuter une fonction à intervalles réguliers, en millisecondes.

```
setInterval(() => {
  console.log('Fonction exécutée toutes les secondes');
}, 1000);
```

PROMISES

Une Promise est un objet qui représente la réussite ou l'échec d'une **opération asynchrone**.

Une Promise peut être dans l'un des états suivants:

- **Pending**: opération en cours d'exécution
- **Resolved**: opération réussie
- **Rejected**: opération échouée

```
// Exemple de création d'une Promise
const myPromise = new Promise((resolve, reject) => {
  const success = true;

  if (success) {
    resolve('Opération réussie!');
  } else {
    reject('Opération échouée.');
  }
});
```

Les Promises sont une manière plus structurée de gérer les opérations asynchrones.

Elles permettent de gérer plus efficacement les erreurs et de simplifier le code en utilisant les méthodes *then* et *catch*.

```
myPromise
  .then(result => {
    console.log(result); // 'Opération réussie!'
  })
  .catch(error => {
    console.error(error); // 'Opération échouée.'
  });
```

Les Promises permettent d'exécuter les opérations asynchrones de manière séquentielle, en enchaînant proprement les opérations asynchrones.

```
myPromise
  .then(result => {
    console.log(result);
    return new Promise((resolve, reject) => {
      resolve('Deuxième opération réussie!');
    });
  })
  .then(result => {
    console.log(result); // Affiche 'Deuxième opération réussie!'
  })
  .catch(error => {
    console.error(error);
  });
```

Plusieurs Promises peuvent être exécutées parallèlement à l'aide de la méthode *Promise.all*.

```
const promise1 = Promise.resolve('Promise 1 réussie');
const promise2 = Promise.resolve('Promise 2 réussie');
const promise3 = Promise.resolve('Promise 3 réussie');

Promise.all([promise1, promise2, promise3])
  .then(results => {
    console.log(results); // Affiche ['Promise 1 réussie', 'Promise
  })
  .catch(error => {
    console.error(error);
  });
```

ASYNC / AWAIT

L'API async/await est une extension de JavaScript introduite dans ECMAScript 2017.

Elle permet d'écrire du code asynchrone de manière synchrone en utilisant les mots-clés *async* et *await*.

Cela rend le code plus lisible et plus facile à écrire, le rendant similaire au code synchrone.

ASYNC

async est utilisé pour déclarer une fonction asynchrone, qui retourne une *Promise*.

```
async function doSomethingQuiteSlow() {
  return new Promise(resolve => setTimeout(resolve, 1000));
}
```

```
async function fetchData() {
  return fetch('https://api.example.com/data');
}
```

```
async function getNumber() {
  return 42;
}

const number = await getNumber(); // Promesse implicitement résolue
```

Une fonction *async* peut contenir une ou plusieurs expressions *await*, qui mettent en pause l'exécution de la fonction jusqu'à ce que la Promise soit résolue ou rejetée.

```
async function doSomethingQuiteSlow() {
  const test = await new Promise(resolve => setTimeout(resolve, 1000
  const otherTest = await new Promise(resolve => setTimeout(resolve,
}
```

AWAIT

C'est le *then* du monde d'avant.

await est utilisé pour attendre la résolution d'une Promise. Il ne peut être utilisé qu'à l'intérieur d'une fonction asynchrone.

Un *await* sur une méthode qui n'est pas résolue ni rejetée bloque le thread d'exécution.

Les expressions *await* peuvent être utilisées dans une boucle (for), dans une condition, ou dans une fonction récursive.

```
async function processArray(array) {
  for (let item of array) {
    await new Promise(resolve => setTimeout(resolve, 1000));
    console.log(item);
  }
}

processArray([1, 2, 3, 4]);
```

Il est techniquement possible d'utiliser *await* dans une boucle *forEach* ou *map*, mais ça fonctionne beaucoup moins bien et les résultats peuvent être aléatoires.

Un chaînage *catch* sur une méthode exécutée via *await* autorise la poursuite de l'exécution du code et complexifie la gestion de l'erreur.

Il est préférable de wrapper un *await* dans une expression *try...catch*.

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

fetchData();
```

PATTERN OBSERVER

Le pattern Observer est basé sur le principe de **publication/abonnement**, où un objet (le sujet) maintient une liste d'observateurs qui sont notifiés des changements d'état.

IMPLÉMENTATION NODE.JS

Les émetteurs d'événements sont utilisés pour gérer les événements asynchrones dans Node.js.

Ils permettent de créer, déclencher et gérer des événements.

Utile pour les applications qui fonctionnent sur un modèle basé sur les événements ou... sur le pattern Observer.

```
// Importation du module events de Node.js
const EventEmitter = require('events');

// Création d'une classe qui étend EventEmitter
class Subject extends EventEmitter {
  constructor() {
    super();
    this.state = 0;
  }

  // Méthode pour changer l'état et notifier les observateurs
  setState(newState) {
    this.state = newState;
    this.emit('stateChange', this.state);
  }
```

La classe *Subject* étend *EventEmitter* et notifie les observateurs des changements d'état en émettant un événement *stateChange*.

```
// Création d'une classe Observateur
class Observer {
  constructor(name) {
    this.name = name;
  }

  // Méthode pour réagir aux changements d'état
  update(state) {
    console.log(`${this.name} a reçu la nouvelle valeur d'état: ${st
  }
}
```

```
// Instanciation du sujet et des observateurs
const subject = new Subject();
const observer1 = new Observer('Observateur 1');
const observer2 = new Observer('Observateur 2');

// Enregistrement des observateurs pour écouter les changements d'ét
subject.on('stateChange', (state) => observer1.update(state));
subject.on('stateChange', (state) => observer2.update(state));

// Changement de l'état du sujet
subject.setState(1);
subject.setState(2);
```

Les observateurs s'inscrivent pour écouter cet événement et réagissent en conséquence.

OBSERVABLES

Les observables sont une autre approche, plus moderne - et plus complexe, pour gérer les opérations asynchrones.

Les observables permettent de gérer les **flux de données** et les **événements** de manière plus efficace, en utilisant des **opérateurs** pour transformer, filtrer et combiner les données.

La librairie la plus populaire pour implémenter les observables est la bibliothèque **RxJS**.

Les observables sont créés à l'aide du constructeur *Observable*, qui prend une fonction en argument, qui reçoit un objet *observer* en argument.

```
import { Observable } from 'rxjs';

const observable = new Observable(observer => {
  observer.next('Première valeur');
  observer.next('Deuxième valeur');
  observer.complete();
});
```

Les observables sont souscrits à l'aide de la méthode *subscribe*, qui prend un objet en argument avec des fonctions de rappel pour gérer les valeurs émises par l'observable.

```
observable.subscribe({
  next: value => console.log(value),
  complete: () => console.log('Terminé!')
});
```

WORKERS

Fonctionnalité avancée qui permet d'exécuter des tâches intensives en arrière-plan, en utilisant des threads supplémentaires pour exécuter le code.

WEB WORKERS

Les **Web Workers** sont des threads séparés qui permettent d'exécuter du code Javascript en arrière-plan, **dans le navigateur**, sans affecter le thread principal.

```
// main.js
// Création d'un Web Worker
const worker = new Worker('calculator.js');

// Envoi de données au worker
worker.postMessage({
  operation: 'fibonacci',
  number: 42
});

// Réception des résultats
worker.onmessage = function(e) {
  console.log('Résultat reçu du worker:', e.data);
};
```

```
// Fonction de calcul de Fibonacci
function fibonacci(n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

// Écoute des messages
self.onmessage = function(e) {
  const { operation, number } = e.data;

  if (operation === 'fibonacci') {
    const result = fibonacci(number);
    self.postMessage(result);
  }
};
```

- Utilisés dans le navigateur
- Communication par messages
- Pas d'accès au DOM
- Idéal pour les calculs intensifs

Cas d'usages:

- Traitement d'images
- Calculs complexes
- Parsing de données

SERVICE WORKERS

Les **Service Workers** agissent comme un **proxy** entre l'**application**, le **navigateur** et le **réseau**. Ils sont particulièrement utiles pour la mise en cache et le fonctionnement hors ligne.

```
// Enregistrement du Service Worker
if ('serviceWorker' in navigator) {
  window.addEventListener('load', () => {
    navigator.serviceWorker.register('/service-worker.js')
      .then(registration => {
        console.log('Service Worker enregistré!', registration);
      })
      .catch(error => {
        console.error('Erreur:', error);
      });
  });
}
```

```
// service-worker.js
const CACHE_NAME = 'v1';
const URLS_TO_CACHE = [
  '/',
  '/index.html',
  '/styles.css',
  '/app.js'
];

// Installation du Service Worker
self.addEventListener('install', event => {
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(cache => cache.addAll(URLS_TO_CACHE))
  );
```

- Proxy entre l'app et le réseau
- Fonctionne hors ligne
- Gestion du cache
- Push notifications

Cas d'usages:

- Mise en cache
- Notifications push
- Fonctionnement hors ligne

WORKER THREADS

Les **Worker threads** sont des threads séparés qui permettent d'exécuter du code Javascript en arrière-plan, **dans Node.js**, sans affecter le thread principal.

```
const { Worker } = require('worker_threads');

function runWorker(data) {
  return new Promise((resolve, reject) => {
    const worker = new Worker('./worker.js', {
      workerData: data
    });

    worker.on('message', resolve);
    worker.on('error', reject);
    worker.on('exit', code => {
      if (code !== 0) {
        reject(new Error(`Worker stopped with exit code ${code}`));
      }
    });
```

```
const { workerData, parentPort } = require('worker_threads');

// Calcul de la somme des nombres
const sum = workerData.numbers.reduce((acc, curr) => acc + curr, 0);

// Envoi du résultat au thread principal
parentPort.postMessage(sum);
```

- Spécifique à Node.js
- Partage de mémoire possible
- Communication par messages
- Idéal pour les opérations CPU intensives

Cas d'usages:

- Traitement de fichiers
- Calculs parallèles

CONCLUSION

L'asynchronisme est une caractéristique essentielle de Javascript et de Node.js, qui permet de gérer les opérations coûteuses en temps de manière efficace.

Le fondement de ce type de programmation repose sur les principes événementiels, caractérisés par une **event loop**.

Bien qu'ils soient moins utilisés aujourd'hui, les **callbacks** font partie intégrante de la programmation asynchrone.

Les **Promises** ont permis de structurer le code asynchrone et de gérer plus efficacement les erreurs.

Depuis 2017, l'API **async/await** permet du code asynchrone encore plus lisible et plus facile à écrire, en le rendant similaire au code synchrone.

Le **pattern Observer** est une approche classique pour gérer les événements asynchrones, en utilisant un modèle de publication/abonnement.

Les **observables** sont une approche plus avancée qui peut être utilisée pour des applications plus complexes.

Les **workers** sont une autre fonctionnalité avancée qui permet d'exécuter des tâches intensives en arrière-plan.

En général, la combinaison des **Promises** et de **async/await** est la méthode recommandée pour gérer les opérations asynchrones en Javascript et Node.js, car elle permet de garder le code lisible et maintenable tout en offrant un bon contrôle sur le flux d'exécution.