

OPTIMISEZ VOS BACKENDS

pour des performances 10-30x supérieures

INTRODUCTION

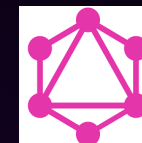
- **Node.js** : Runtime JavaScript basé sur le moteur V8 de Google
- **Créé en 2009**, il est l'environnement d'exécution le plus populaire pour du dev backend
- **Open Source** et porté par la fondation Open JS



POLYVALENT



- **Backends d'API REST** et GraphQL
- **Applications temps réel** WebSockets/Socket.io/WebRTC
- **Applications desktop** (Electron.js → Slack, VS Code, Discord)
- **Outils de CLI** (exemples: npm, vite, ng angular cli, webpack, eslint, ...)
- **Proxy / reverse proxy**
- **DevOps**: Cron jobs, automation, CI/CD
- **Serveurs d'email** (SMTP avec Nodemailer, mail servers custom)

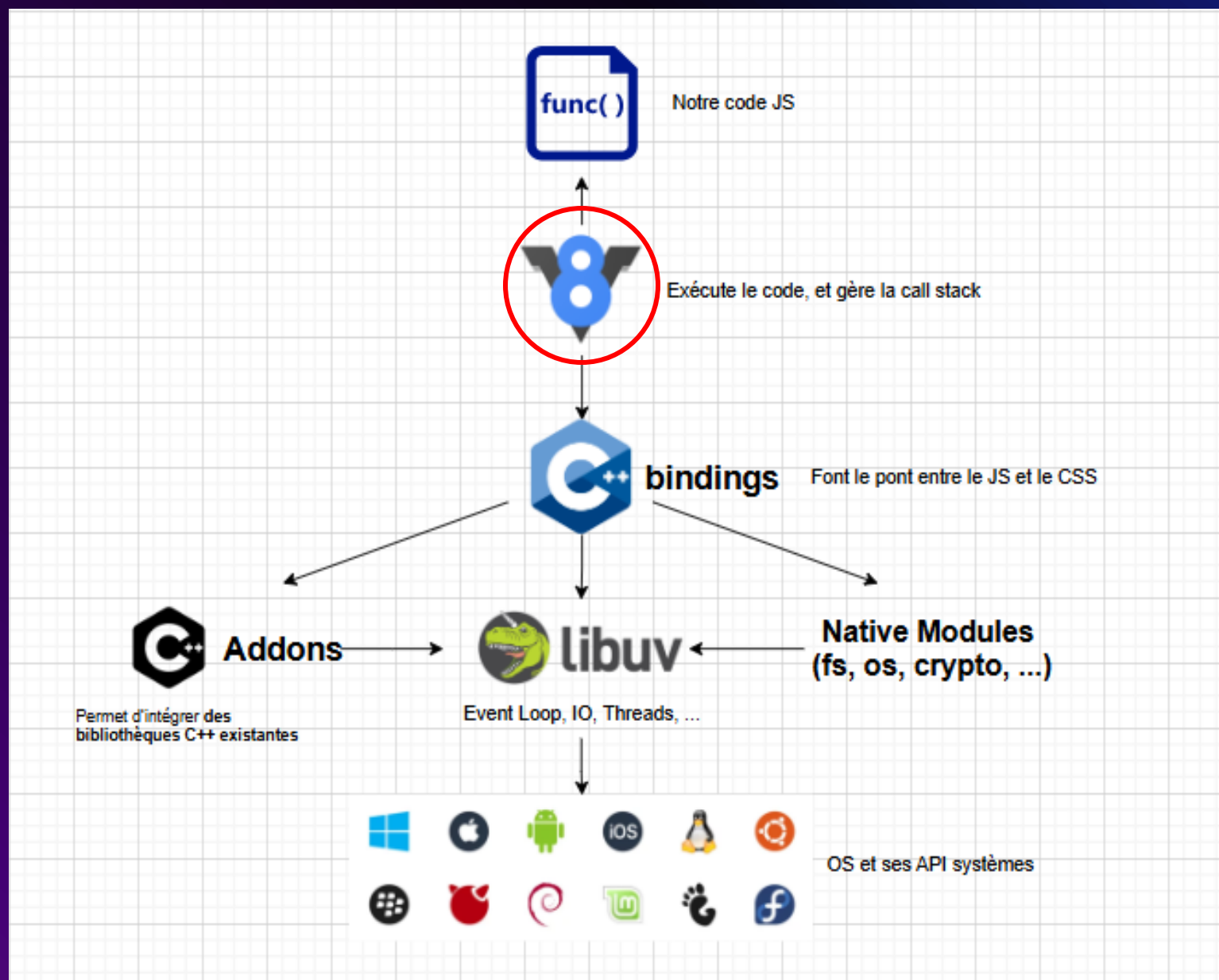


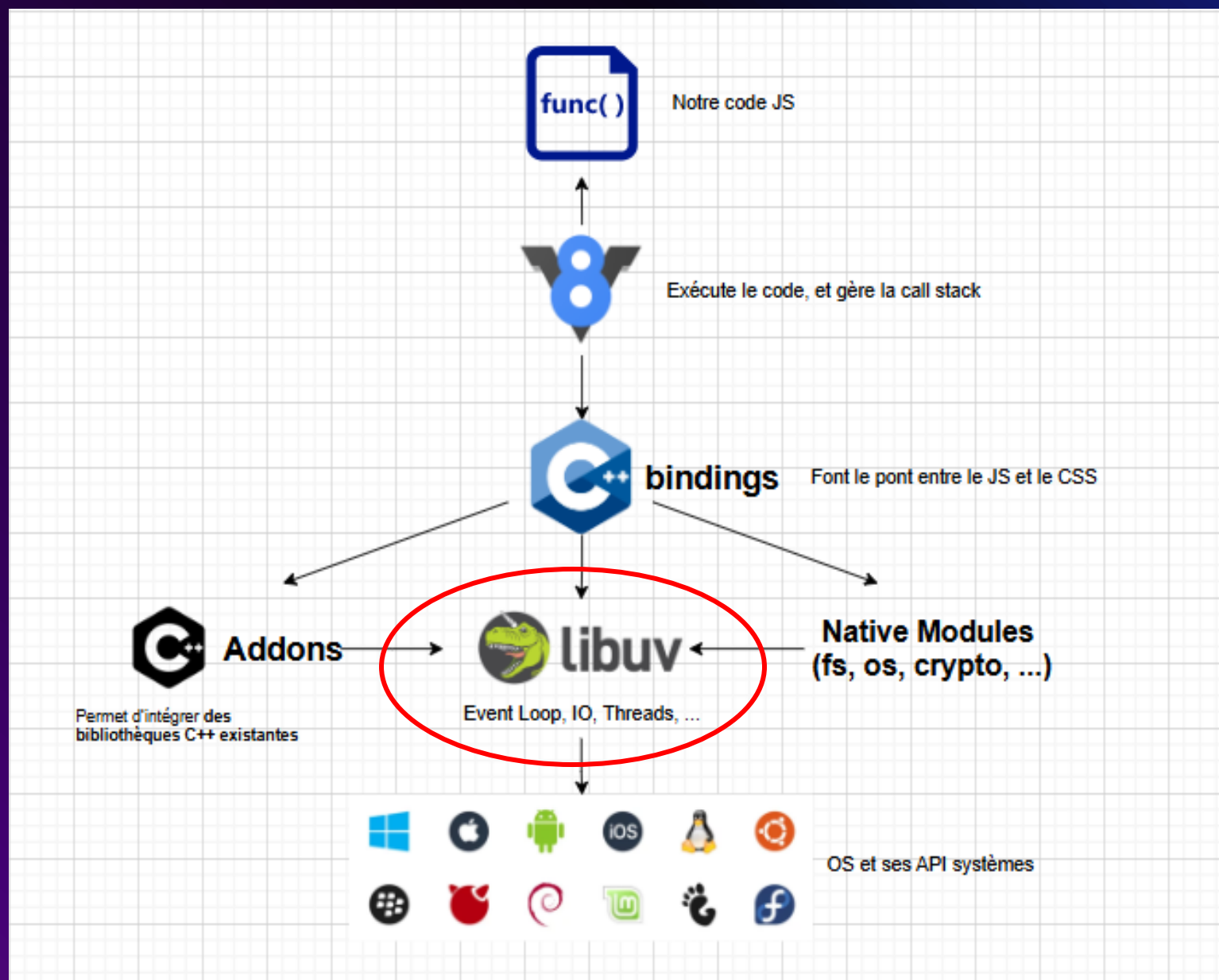
ORGANISATION

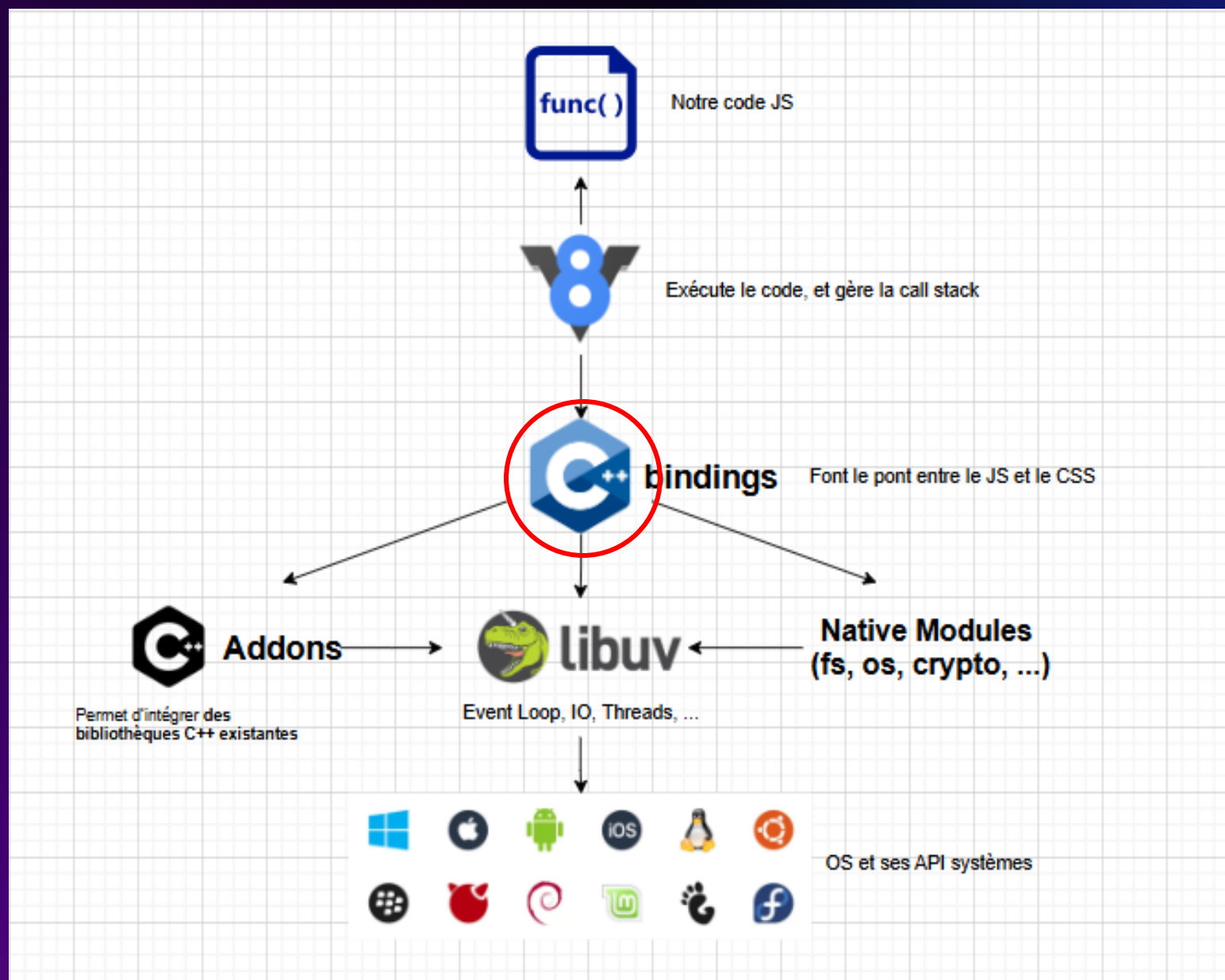
1. Architecture de Node.js
2. Optimisation de la base de données
3. Alternatives à JSON.parse/JSON.stringify
4. Comparaison des frameworks
5. Mise cache
6. Parallélisation des tâches lourdes
7. Conclusion

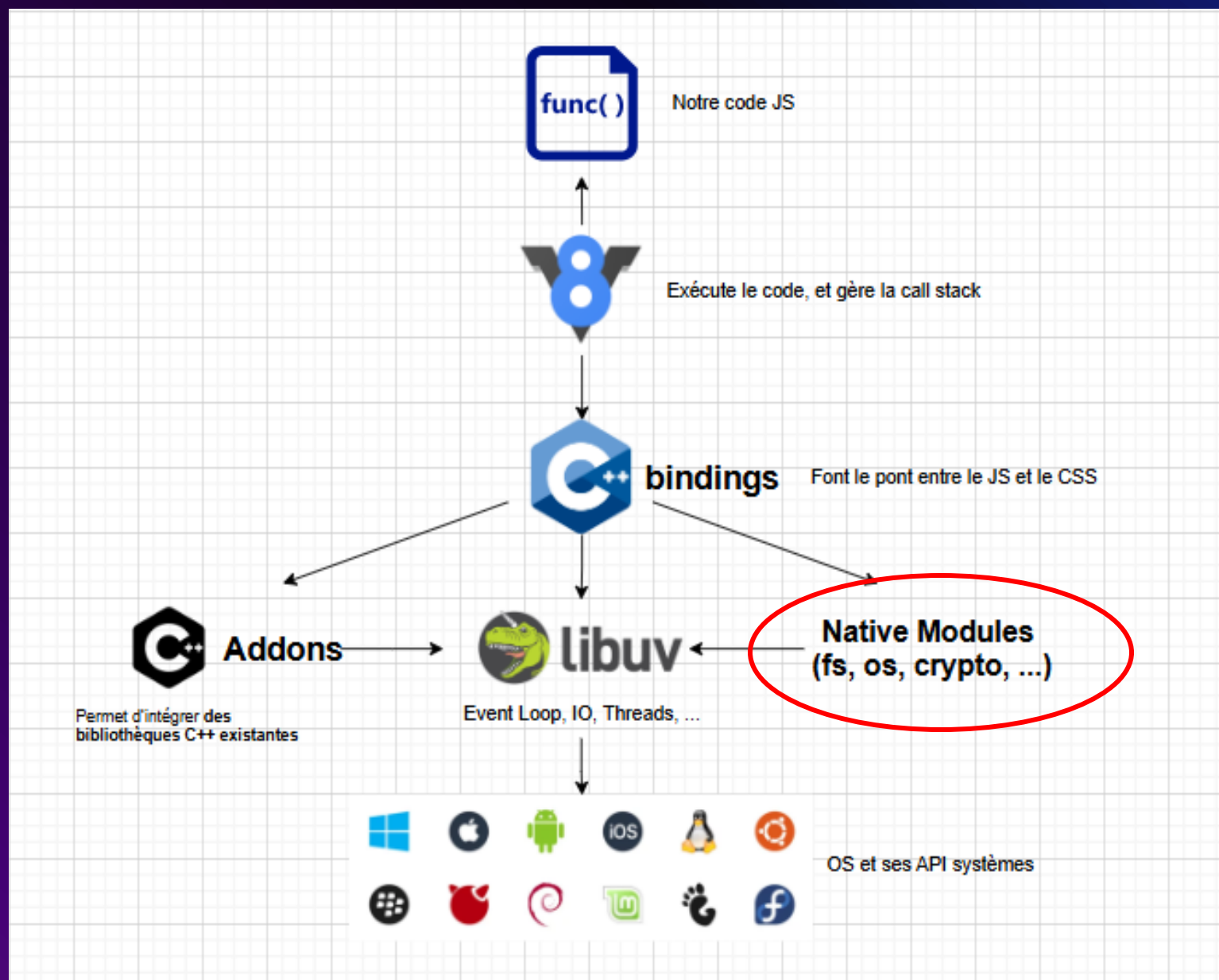
PARTIE 1

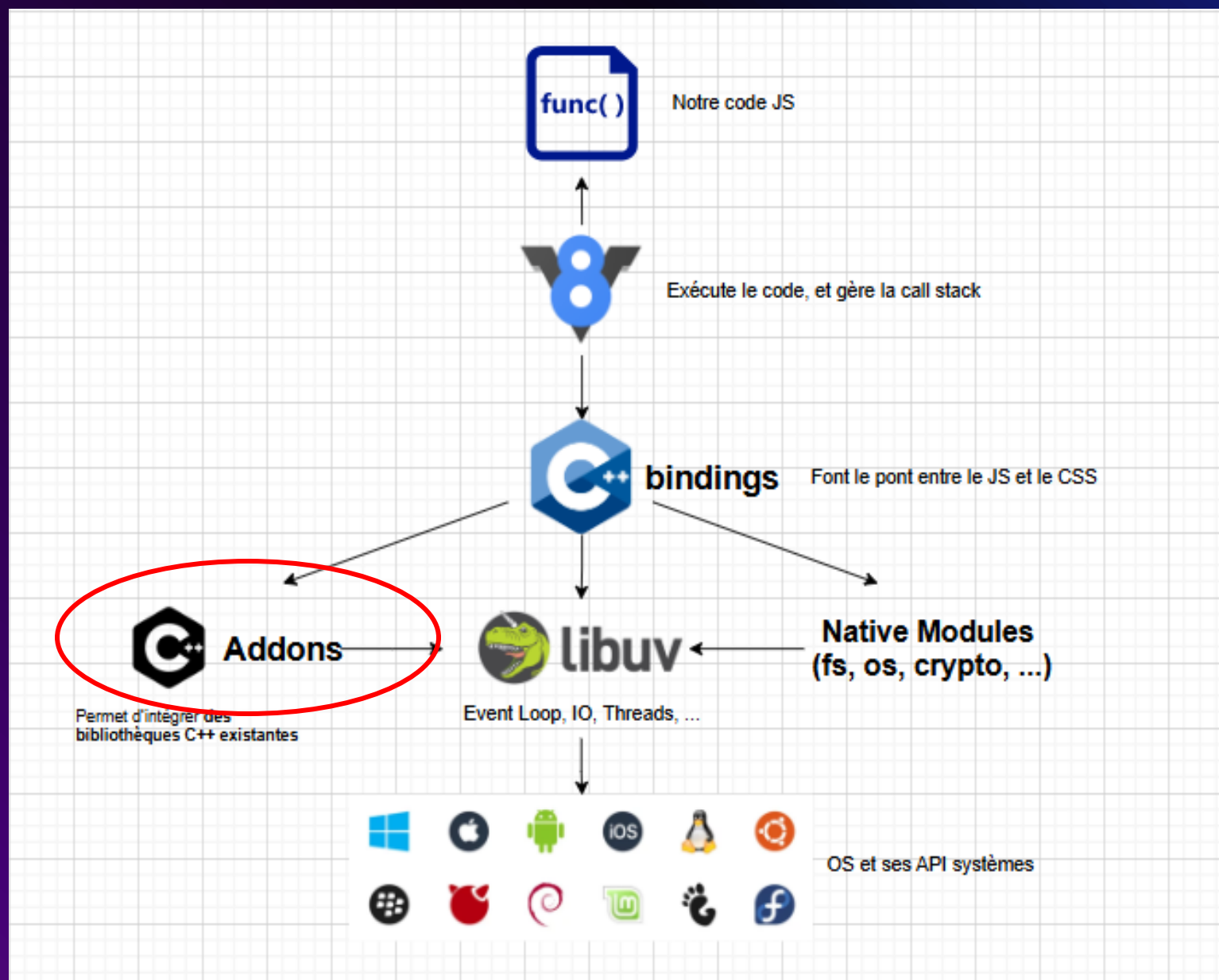
ARCHITECTURE DE NODE.JS



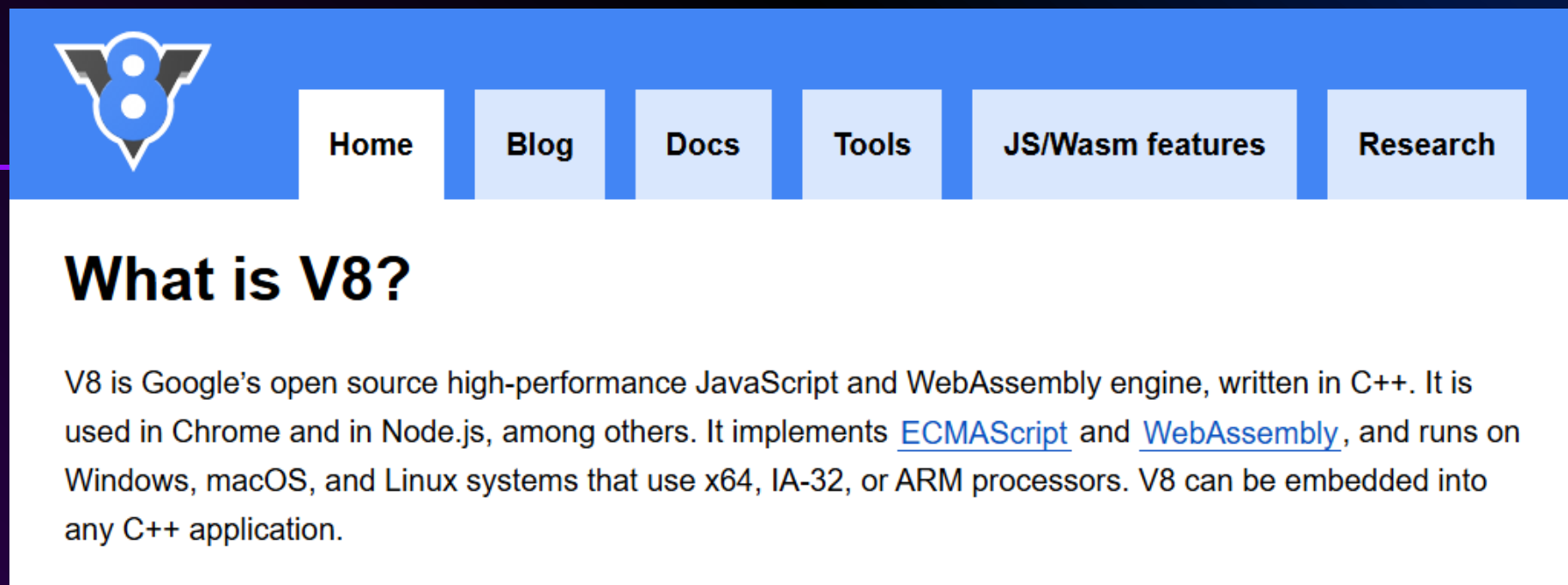




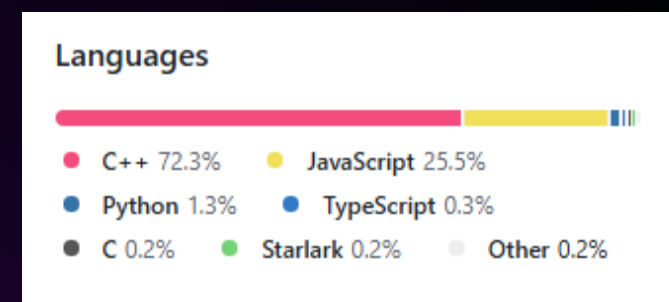




V8



- V8 c'est le moteur de Node.js
- C'est lui qui est responsable de la grammaire, des types de données. C'est v8 qui interprète le JS
- Il est open source: <https://v8.dev/>



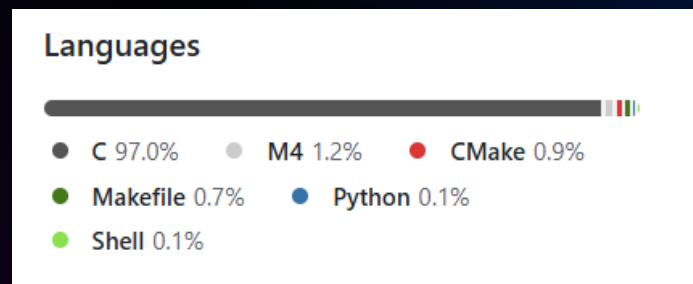
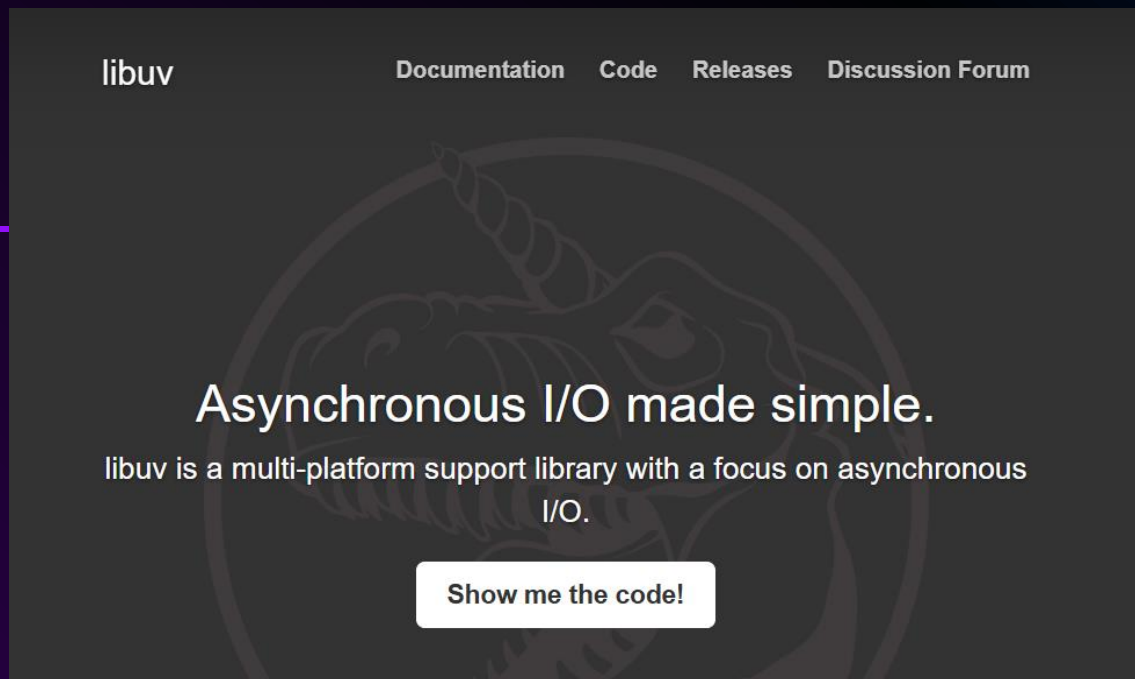
V8

Dans cette définition de Google, 2 éléments principaux nous intéressent

- **JS and Wasm engine:** V8 est un programme qui comprend et exécute du JavaScript et du WebAssembly.
- **V8 implémente les standards ECMAScript et WebAssembly:** Les standards définissent le coeur du langage (syntaxe, structures de contrôle, promises, etc.). Javascript et WASM sont des implémentations de ce standard
- **Ne font pas partie du standard ECMAScript:** `require()`, `fs`, `http`, `os`, `setTimeout`, `setInterval`, `setImmediate`, etc



LIBUV



- Quand on exécute du code en JavaScript, il peut y avoir des **opérations lentes**
- Plutôt que d'attendre que ces opérations se terminent, **libuv s'occupe de les exécuter en arrière-plan et prévient Node.js quand elles sont prêtes.**
- Libuv est donc un framework qui va **"masquer la complexité du flux d'exécution asynchrone"**

LIBUV

- Grâce à libuv, on va pouvoir lire le contenu d'un fichier, envoyer une requête via le réseau, connaître la quantité de RAM disponible sur l'OS, exécuter une action après qu'un certain nombre de temps se soit écoulé, etc
- Il est open source: <https://libuv.org/>
- **Lien avec l'Event Loop**
 - Tout comme le JS est au coeur de V8, **L'Event Loop est au cœur de libuv.**
 - L'Event Loop qui décide **quand et dans quel ordre** exécuter les tâches asynchrones.
 - Chef d'orchestre de la partie asynchrone



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

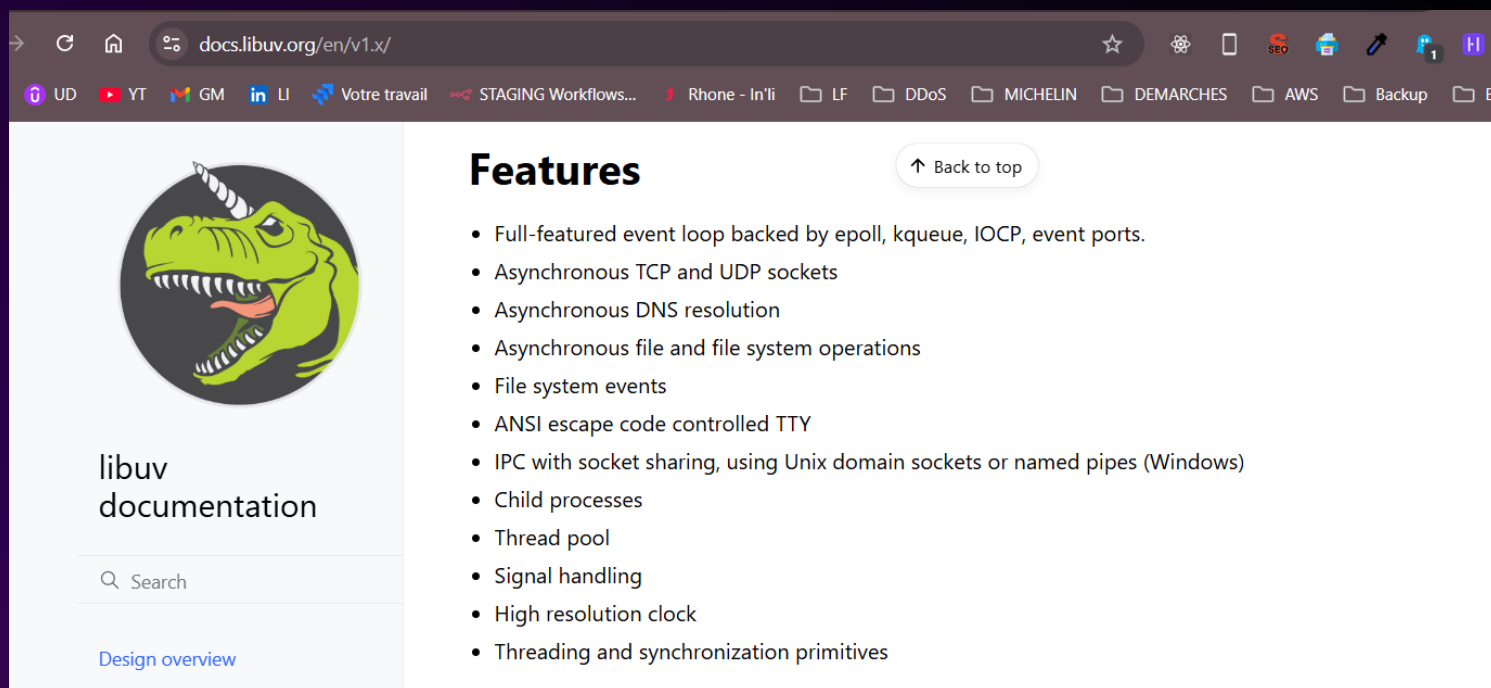
```
fs.readFile("fichier.txt", "utf8", (err, data) => {  
  console.log("Fichier lu :", data);  
});
```

```
console.log("Autre tâche...");
```

```
// fs.readFile() dit à libuv : "Lis ce fichier".  
// libuv exécute la lecture en arrière-plan (hors du thread principal).  
// Pendant ce temps, Node.js continue à exécuter console.log("Autre tâche...").  
// Quand le fichier est prêt, libuv informe Node.js, qui exécute le callback  
// console.log("Fichier lu : ...").
```


LIBUV

- Libuv permet d'implémenter un bon nombre d'APIs asynchrones spécifiques, sans lesquels Node ne serait pas Node



LES BINDINGS C++

- D'une part, on sait lire et exécuter du code JS standard grâce à v8.
- D'autre part, on dispose de libuv implémentant les features asynchrones
- Comment faire appel à libuv à partir du code JS? Pour ce faire, on a besoin d'une couche supplémentaire: les bindings C++
- **Les bindings C++** servent d'interface entre le JavaScript exécuté par V8 et les fonctionnalités bas niveau de libuv.
- Par exemple, l'API `fs.readFile()` en JavaScript **appelle en réalité une fonction C++** qui utilise libuv pour lire un fichier.



```
#include <node.h> // Inclut l'API Node.js pour créer des modules natifs
#include <v8.h>    // Inclut l'API V8 pour manipuler les objets JavaScript

// Fonction `Print` qui sera exposée à Node.js
void Print(const v8::FunctionCallbackInfo<v8::Value>& args) {
    printf("Hello world\n"); // Affiche "Hello world" dans la console
}

// Fonction d'initialisation du module
void Initialize(v8::Local<v8::Object> exports) {
    // Associe la fonction `Print` au nom "print" dans le module Node.js
    NODE_SET_METHOD(exports, "print", Print);
}

// Définit le point d'entrée du module, qui s'exécutera à son chargement
NODE_MODULE(binding, Initialize)
```



```
const printBinding = require('./build/Release/print_binding');

// Utilisation de la fonction `print` exposée en C++
printBinding.print("Hello from C++!");
```

LES MODULES "NATIFS"

- C'est l'ensemble des fonctionnalités fournies par **Node.js**
- Cf: <https://nodejs.org/docs/latest-v22.x/api/index.html>
- Ces modules **ne viennent pas de V8** mais sont développés en **JS et C++**.
- Beaucoup de ces modules utilisent **libuv et les bindings C++** sous le capot.

Node.js v22.14.0 documentation

► [Other versions](#) | ► [Options](#)

- [About this documentation](#)
- [Usage and example](#)
- [Assertion testing](#)
- [Asynchronous context tracking](#)
- [Async hooks](#)
- [Buffer](#)
- [C++ addons](#)
- [C/C++ addons with Node-API](#)
- [C++ embedder API](#)
- [Child processes](#)
- [Cluster](#)
- [Command-line options](#)
- [Console](#)
- [Corepack](#)
- [Crypto](#)
- [Debugger](#)
- [Deprecated APIs](#)
- [Diagnostics Channel](#)
- [DNS](#)
- [Domain](#)
- [Errors](#)
- [Events](#)
- [File system](#)
- [Globals](#)
- [HTTP](#)
- [HTTP/2](#)
- [HTTPS](#)

PARTIE 2

CRITERES DE PERFORMANCE BACKEND

LATENCE ET DEBIT



Latence

- Temps total entre la requête et la réponse complète
- On mesure le 99e percentile, pas la moyenne ou la médiane.
- Pourquoi le 99e percentile ? Il reflète les pires cas : 99% des requêtes sont plus rapides que cette valeur.
- Si le **99e percentile est 50 ms**, cela signifie que **99 % des requêtes prennent moins de 50 ms**



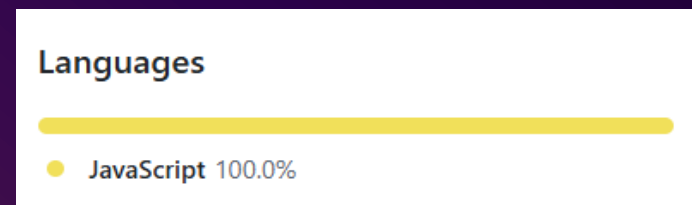
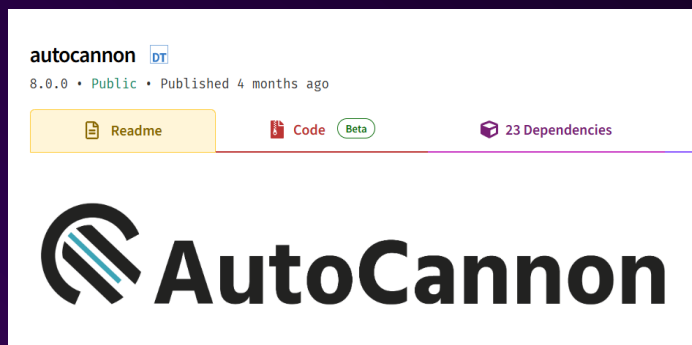
Débit (Throughput)

- Mesurer le nombre moyen de requêtes par seconde (RPS) et le nombre total de requêtes.

OBJECTIF DE PERFORMANCE



Réduire la latence et augmenter le débit.



- **Outil de benchmarking: Autocannon**
HTTP/1.1 benchmarking tool written in node
<https://www.npmjs.com/package/autocannon>
- **Installation**
`npm i autocannon -g`
- **Usage**
`autocannon [opts] URL`

PARTIE 3

APPROCHES D'OPTIMISATION

MONGOOSE VS DRIVER NATIF

- **Mongoose** est un ORM qui simplifie la gestion des modèles et des relations.
- Autres: Prisma, Knex.js, TypeORM
- Ils ajoutent une **surcharge** due à la sérialisation et à la gestion des références.
- Le driver natif interagit **directement** avec la base, sans intermédiaire.



```
// Définition du schéma Address
const AddressSchema = new mongoose.Schema({
  street: String,
  city: String,
  country: String
});

// Définition du schéma Person avec une référence à Address
const PersonSchema = new mongoose.Schema({
  name: String,
  age: Number,
  email: String,
  address: { type: mongoose.Schema.Types.ObjectId, ref: 'Address' }
});
```



```
// Définition du schéma et du constructeur
const PersonSchema = new mongoose.Schema({
  civility: String,
  name: String,
  age: Number,
})
const Person = mongoose.model('Person', PersonSchema)

// Route pour ajouter une personne
app.post('/persons', async (req, res) => {
  try {
    // Création et sauvegarde d'une nouvelle personne
    const { civility, name, age } = req.body
    const person = new Person({ civility, name, age })
    const savedPerson = await person.save()

    // Retourner l'objet inséré
    return res.status(200).json({
      message: 'Person created successfully',
      data: { _id: savedPerson._id },
    })
  } catch (error) {
    return res.status(500).json({ message: `Error saving person
${error.message}` })
  }
})
```



```
// Route pour ajouter une personne
app.post('/persons', async (req, res) => {
  try {
    // Création et sauvegarde d'une nouvelle personne
    const { civility, name, age } = req.body
    await db.collection('persons').insertOne({ civility, name, age })

    // Retourner l'objet inséré
    return res.status(200).json({message: 'Person created successfully'})
  } catch (error) {
    return res.status(500).json({ message: `Error saving person ${error.message}` })
  }
})
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	7 ms	14 ms	34 ms	41 ms	15.9 ms	8.22 ms	160 ms

Mongoose

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	297	297	650	804	609.3	165.13	297
Bytes/Sec	94.5 kB	94.5 kB	207 kB	256 kB	194 kB	52.5 kB	94.4 kB

Req/Bytes counts sampled once per second.
of samples: 10

6k requests in 10.04s, 1.94 MB read

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	5 ms	10 ms	21 ms	25 ms	11.24 ms	18.23 ms	572 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	363	363	926	1,031	851	192.49	363
Bytes/Sec	115 kB	115 kB	295 kB	328 kB	271 kB	61.2 kB	115 kB

Req/Bytes counts sampled once per second.
of samples: 10

9k requests in 10.04s, 2.71 MB read

Driver natif

CONFIGURER LA CONNECTION POOL

- Les applications communiquent avec MongoDB via une connexion.
- **Ouvrir et fermer une connexion à chaque requête est coûteux en termes de performance.**
- Le pool de connexions: réutiliser des connexions existantes au lieu d'en créer une nouvelle à chaque requête.

```
const uri = 'mongodb://localhost:27020/my-db'
const client = new MongoClient(uri, { maxPoolsize: 500 })

client
  .connect()
  .then(() => {
    console.log('Connected to MongoDB')
  })
  .catch((err) => {
    console.error(' MongoDB connection error:', err)
  })
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	6 ms	12 ms	25 ms	31 ms	13.83 ms	20.67 ms	483 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	337	337	706	871	697.2	150.21	337
Bytes/Sec	93.1 kB	93.1 kB	195 kB	241 kB	192 kB	41.5 kB	93 kB

Sans
connection
pool

Req/Bytes counts sampled once per second.
of samples: 10

7k requests in 10.03s, 1.92 MB read

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	6 ms	11 ms	21 ms	27 ms	12.35 ms	19.49 ms	518 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	369	369	818	954	777	176.91	369
Bytes/Sec	102 kB	102 kB	226 kB	263 kB	214 kB	48.8 kB	102 kB

Avec
connection
pool

Req/Bytes counts sampled once per second.
of samples: 10

8k requests in 10.03s, 2.14 MB read

CONNECTION POOL: RISQUES

- **1 connexion = mémoire + ressources CPU côté client et serveur.**
- **MongoDB a une limite max de connexions simultanées** (maxIncomingConnections)
 - Si le pool est trop grand, vous risquez d'atteindre cette limite rapidement, bloquant ainsi les nouvelles requêtes.
- **Bonne pratique : Trouver le bon équilibre**
 - Réglez le pool à une taille adaptée à votre charge réelle et surveillez les statistiques
 - **Peu de requêtes simultanées** => un pool de 10-20 connexions.
- **Choix stratégique**
 - productivité (Mongoose) vs. performance (driver natif).
 - Mongoose pour les petites/moyennes apps, driver natif pour les **grandes échelles**.

INDEXES DE BDD

- **Index:** Structure de données qui permet de rechercher rapidement des documents sur un ou plusieurs champs spécifiques
- C'est comme un index à la fin d'un livre, et permet de réduire le temps de recherche des documents qui correspondent à certains critères
- **Sans index, MongoDB doit analyser chaque document d'une collection** pour renvoyer les résultats de la requête.



```
// Définition du schéma et du constructeur
const PersonSchema = new mongoose.Schema({
  civility: String,
  name: String,
  age: Number,
  email: { type: String, required: true }
})
```

```
// Création de l'index sur le champ email
// en ordre croissant
PersonSchema.index({ email: 1 });
```

```
const Person = mongoose.model('Person', PersonSchema)
```



```
borelkoumo@THINKPAD-CBTW ~/MKUTANO-2025 (main) $ autocannon -c 10 -m 'GET' http://localhost:3000/persons
Running 10s test @ http://localhost:3000/persons
10 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	634 ms	896 ms	1392 ms	1459 ms	936.64 ms	207.71 ms	1595 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	5	5	10	14	10.2	2.49	5
Bytes/Sec	2.96 MB	2.96 MB	5.92 MB	8.29 MB	6.04 MB	1.47 MB	2.96 MB

```
Req/Bytes counts sampled once per second.
# of samples: 10
```

```
112 requests in 10.04s, 60.4 MB read
```

```
borelkoumo@THINKPAD-CBTW ~/MKUTANO-2025 (main) $ autocannon -c 10 -m 'GET' http://localhost:6000/persons
Running 10s test @ http://localhost:6000/persons
10 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	581 ms	771 ms	1118 ms	1160 ms	801.05 ms	132.61 ms	1224 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	8	8	12	13	12	1.42	8
Bytes/Sec	4.49 MB	4.49 MB	6.73 MB	7.29 MB	6.73 MB	794 kB	4.48 MB

```
Req/Bytes counts sampled once per second.
# of samples: 10
```

```
130 requests in 10.03s, 67.3 MB read
```

Sans index

Avec index

AVOID JSON.PARSE & JSON.STRINGIFY

- **Exemple de `JSON.stringify()`**
 - JavaScript regarde chaque propriété de l'objet, vérifie leur type et les convertit en texte.
 - **Processus est effectué à chaque appel à `JSON.stringify()`.**
- **Requête POST** => désérialisation du corps de la requête via `JSON.parse`
- **Requête GET** => sérialisation corps de la réponse (convert an object to a string)
- **Avec express, par défaut on utilise le middleware `body-parser`**, qui lui-même fait appel à `JSON.parse/stringify`
- **Problème:** Ces opérations sont **synchrones et bloquent l'Event Loop**, ce qui peut ralentir l'application pour un gros olume de données.

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	11 ms	20 ms	39 ms	48 ms	23.52 ms	41.13 ms	850 ms

JSON.stringify

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	101	101	459	512	415.8	115.64	101
Bytes/Sec	1.21 MB	1.21 MB	5.48 MB	6.12 MB	4.97 MB	1.38 MB	1.21 MB

Req/Bytes counts sampled once per second.
of samples: 10

4k requests in 10.03s, 49.7 MB read

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	11 ms	20 ms	37 ms	44 ms	20.93 ms	7.74 ms	166 ms

fast-json-stringify

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	310	310	474	543	465.5	67.01	310
Bytes/Sec	3.19 MB	3.19 MB	4.87 MB	5.59 MB	4.79 MB	689 kB	3.19 MB

Req/Bytes counts sampled once per second.
of samples: 10

5k requests in 10.04s, 47.9 MB read

FRAMEWORKS BENCHMARKING

- **Le choix du framework a un impact sur les performances**
- **J'ai été moi-même surpris des résultats**
- **Methodologie: logique identique à travers toutes les implémentations.**
 - Le même type de requête (par exemple, POST /persons)
 - La même opération sur la base de données (insertOne)
 - La même structure de données (par exemple, { name, age, email })
 - Le même format de réponse (retourner l'objet inséré)



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

```
const app = express()
```

```
const PORT = 3000
```

```
app.get('/', (req, res) => {  
    return res.status(200).send('Hello world')  
})
```

```
app.listen(PORT, () => {  
    console.log('Listening on port ' + PORT)  
})
```



```
const Fastify = require('fastify')
const fastify = Fastify()
```

```
const PORT = 6000
```

```
fastify.get('/', (request, reply) => {
  return 'Hello World'
})
```

```
fastify.listen({ port: Number(PORT) }, (err, address) => {
  if (err) {
    process.exit(1)
  }
  console.log(`Server started on port ${PORT}`)
})
```



```
import { Controller, Get } from '@nestjs/common';  
import { AppService } from '../app.service';
```

```
@Controller()
```

```
export class AppController {
```

```
  constructor(private readonly appService: AppService) {}
```

```
  @Get()
```

```
  getHello(): string {
```

```
    return 'Hello World!';
```

```
  }
```

```
}
```

```
borelkoumo@THINKPAD-CBTW ~/MKUTANO-2025 (main) $ autocannon -c 10 -m 'GET' http://localhost:3000
Running 10s test @ http://localhost:3000
10 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	2 ms	3 ms	8 ms	11 ms	3.11 ms	2.05 ms	49 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	1,515	1,515	2,963	3,253	2,778.6	523.58	1,515
Bytes/Sec	361 kB	361 kB	705 kB	774 kB	661 kB	125 kB	361 kB

Req/Bytes counts sampled once per second.
of samples: 10

28k requests in 10.04s, 6.61 MB read



```
borelkoumo@THINKPAD-CBTW ~/MKUTANO-2025 (main) $ autocannon -c 10 -m 'GET' http://localhost:6000
Running 10s test @ http://localhost:6000
10 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	0 ms	0 ms	0 ms	1 ms	0.03 ms	0.25 ms	17 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	21,839	21,839	35,551	46,751	36,515.2	6,591.63	21,838
Bytes/Sec	3.84 MB	3.84 MB	6.26 MB	8.22 MB	6.43 MB	1.16 MB	3.84 MB

Req/Bytes counts sampled once per second.
of samples: 10

365k requests in 10.03s, 64.3 MB read



```
borelkoumo@THINKPAD-CBTW ~/MKUTANO-2025 (main) $ autocannon -c 10 -m 'GET' http://localhost:9000
Running 10s test @ http://localhost:9000
10 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	1 ms	1 ms	6 ms	10 ms	1.77 ms	1.78 ms	28 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	3,155	3,155	4,387	6,647	4,520.7	941.28	3,154
Bytes/Sec	754 kB	754 kB	1.05 MB	1.59 MB	1.08 MB	225 kB	754 kB

Req/Bytes counts sampled once per second.
of samples: 10

45k requests in 10.03s, 10.8 MB read



FRAMEWORKS BENCHMARKING

- **Express**
 - Simple mais pas le plus rapide en raison de son approche basée sur les middlewares.
- **Fastify**
 - Beaucoup plus rapide qu'Express grâce à son cœur léger et son optimisation de la gestion du JSON.
- **NestJS**
 - Idéal pour les applications bien structurées, mais ajoute une légère surcharge due aux décorateurs.

EXPLOITER LE CACHE

- **Permet d'améliorer les performances d'un backend**, - latence et - ressources (CPU/RAM)
- **Utilisations du cache pour:**
 - Eviter de recalculer des données qui ne changent pas fréquemment
 - Récupération de données fréquemment demandées (produits populaires, les paramètres de l'application, ...)
 - Résultats de requêtes complexes (nécessitant plusieurs appels de bdd)
 - servir des pages statiques
 - stocker les infos liées à un utilisateur connecté
 - mettre en cache le résultat d' Appels à des services externes pour récupérer des données (météo, taux de change, API publiques).



```
autocannon -c 10 -m 'GET' http://localhost:9000/decode -H 'Authorization=Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1bm90dXN0IjoiIiwiaWF0IjoiYXNjaWwzYm90iLCJpYXQiOiE3NDAwMDU5ODcsImV4cCI6MTc0MDYxMDc4N30.HapyotLoWYDriroP-CR1_KuEqWY-Z2SjniRckdJY_XU'
```



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

const PORT = 3000

const person = {
  name: 'Paul BIYA',
  age: 92,
  email: 'paulbiya@gmail.com'
}

app.get('/decode', (req, res) => {
  res.json(person);
});

app.listen(PORT, () => console.log(`Server running on port
${PORT}`));
```



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

const PORT = 3000

const person = {
  name: 'Paul BIYA',
  age: 92,
  email: 'paulbiya@gmail.com'
}

app.get('/decode', (req, res) => {
  res.json(person);
});

app.listen(PORT, () => console.log(`Server running on port
${PORT}`));
```

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

const app = express();
const SECRET_KEY = 'MY_PRIVATE_SECRET';
const PORT = process.argv[2].split('=')[1];

const tokenCache = new Map(); // Cache en mémoire

// Middleware de vérification du JWT avec cache
function authenticateJWT(req, res, next) {
  const token = req.headers.authorization?.split(' ')[1];

  if (!token) return res.status(401).json({ message: 'Unauthorized' });
  // Vérifier si le token est déjà en cache
  if (tokenCache.has(token)) {
    req.user = tokenCache.get(token);
    return next();
  }

  jwt.verify(token, SECRET_KEY, (err, decoded) => {
    if (err) return res.status(403).json({ message: 'Forbidden' });

    tokenCache.set(token, decoded); // Stocker dans le cache
    const { name, age, email } = decoded
    req.user = { name, age, email }
    next();
  });
}

app.get('/decode', authenticateJWT, (req, res) => {
  res.json({ ...req.user });
});

app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	2 ms	3 ms	11 ms	14 ms	4.08 ms	2.47 ms	33 ms

Serveur sans
authentification
JWT

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	1,376	1,376	2,311	2,621	2,189	357.4	1,376
Bytes/Sec	403 kB	403 kB	677 kB	768 kB	641 kB	105 kB	403 kB

Req/Bytes counts sampled once per second.
of samples: 10

22k requests in 10.05s, 6.41 MB read

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	1 ms	3 ms	10 ms	18 ms	8.93 ms	58.89 ms	854 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	469	469	891	1,842	1,060	428.95	469
Bytes/Sec	153 kB	153 kB	292 kB	603 kB	347 kB	140 kB	153 kB

Req/Bytes counts sampled once per second.
of samples: 10

11k requests in 10.03s, 3.47 MB read

Serveur avec
authentification
JWT + cache

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	7 ms	13 ms	27 ms	31 ms	13.56 ms	4.89 ms	58 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	592	592	709	791	710.7	60.66	592
Bytes/Sec	174 kB	174 kB	208 kB	232 kB	208 kB	17.8 kB	173 kB

Req/Bytes counts sampled once per second.
of samples: 10

7k requests in 10.03s, 2.08 MB read

<http://localhost:6000/>
Appuyez sur Ctrl en cliquant pour suivre le lien.

Serveur avec
authentification
JWT

EXPLOITER LE CACHE

- Les types de caches utilisables:
 - Cache in-memory (non recommandé)
 - Redis
 - Memcached
 - Varnish
 - CDN

PARTIE 4

RECOMMENDATIONS

AUTRES

- **Parralélisation des traitements**
 - node:worker_threads
 - Lorsque la quantité de travail est uniforme
 - Approche DPR: diviser le fichier par le nombre de processeurs
- **The One Billion Row Challenge with Node.js**
 - <https://github.com/1brc/nodejs>
 - <https://jackyef.com/posts/1brc-nodejs-learnings>

AUTRES

- **Retrouvez tous les codes ici**
 - **<https://github.com/borelkoumo/mkutano-2025-demo>**

THANK YOU
