

INITIATION

NODE.JS

PROGRAMME

- ▶ Installation
- ▶ Dépendances
- ▶ Async / Await
- ▶ Création d'un premier serveur Express
- ▶ Création des routes
- ▶ Configuration et utilisation de mariaDB avec Node.js/Express
- ▶ Utilisation du module fs (File System)



INITIATION NODE.JS

INSTALLATION

QU'EST-CE QUE NODE.JS ?

- ▶ Runtime JavaScript côté serveur basé sur le moteur V8 de Chrome.
- ▶ Permet d'exécuter du JavaScript en dehors du navigateur.

Avantages de node.js :

- ▶ Large écosystème de packages (npm)
- ▶ Même langage front-end et back-end
- ▶ Grande communauté active

INSTALLATION

Installez node.js (la version LTS) à partir de <https://nodejs.org/en>

Pour vérifier que tout est bien installé tapez les commandes suivantes dans votre terminal.

```
node --version  
npm --version
```




INITIATION NODE.JS

DÉPENDANCES

COMMANDES NPM ESSENTIELLES

Initialiser un projet

```
npm init
```

Installer une dépendance

```
npm install express
```

Installer une dépendance de développement

```
npm install --save-dev nodemon
```

Désinstaller une dépendance

```
npm uninstall express
```




INITIATION NODE.JS

ASYNC / AWAIT

ASYNC / AWAIT

// Promise classique

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

// Async/Await

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




INITIATION NODE.JS

CRÉATION D'UN PREMIER SERVEUR EXPRESS

EXPRESS C'EST QUOI ?

Express est un framework qui simplifie la création de sites web et d'APIs avec Node.js. C'est comme une boîte à outils qui rend le développement plus facile.

Il rend la création d'applications web beaucoup plus simple que le HTTP standard de Node.js.

INITIALISATION D'UN PROJET DE TEST

- ▶ Créer un dossier appelé `node-test`
- ▶ Placez vous dans le dossier et tapez la commande `npm init -y` pour y générer un `package.json`
- ▶ Faites un `npm install express`
- ▶ Créez un fichier `app.js` à la racine du projet et y ajouter le code de la slide suivante

EXEMPLE SIMPLE

```
// Installation : npm install express
const express = require("express");
const app = express();

// Route simple
app.get("/", (req, res) => {
  res.send("Hello World!");
});

// Démarre le serveur
app.listen(3000, () => {
  console.log("Serveur démarré sur http://localhost:3000");
});
```


EXPLICATIONS

1. `npm install express`

- C'est la commande pour installer Express sur votre ordinateur
- À taper dans le terminal avant de commencer

2. `const express = require('express');`

- On importe Express pour l'utiliser
- C'est comme dire "je veux utiliser Express"

3. `const app = express();`

- On crée notre application web
- C'est le point de départ

4. `app.get('/', (req, res) => { ... })`

- On crée une page d'accueil ('/')
- Quand quelqu'un visite le site, on lui répond 'Bonjour!'

5. `app.listen(3000, ...)`

- On démarre le site sur localhost:3000

LES AVANTAGES D'EXPRESS

1. Routes faciles à créer
2. Gestion simple des requêtes (GET, POST, etc.)
3. Middleware pour ajouter des fonctionnalités



PRATIQUE

Heure

HEURE

Créer un script qui affiche la date et l'heure actuelle dans le terminal de VSCode



INITIATION NODE.JS

CRÉATION DES ROUTES

QU'EST-CE QU'UNE ROUTE ?

Une route, c'est comme une adresse sur votre site web qui définit :

1. Le chemin (URL)
2. L'action à faire quand quelqu'un visite ce chemin

Quand vous visitez :

- localhost:3000 → affiche "Accueil"
- localhost:3000/contact → affiche "Page Contact"
- localhost:3000/profil/123 → affiche "Profil numéro : 123"

STRUCTURE D'UNE ROUTE

- Créez un dossier `routes` à la racine de votre projet, puis un fichier `users.js` à l'intérieur
- Ajoutez le code suivant dans le fichier `users.js` :

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

router.get("/", (req, res) => {
  res.json({ message: "Users List" });
});

router.post("/", (req, res) => {
  res.json({ message: "Create new user" });
});

module.exports = router;
```


STRUCTURE D'UNE ROUTE

- Mettez à jour le fichier `app.js` avec le code suivant

```
// Installation : npm install express
const express = require("express");
const app = express();

const usersRouter = require("./routes/users");
app.use("/users", usersRouter);

// Démarre le serveur
app.listen(3000, () => {
  console.log("Serveur démarré sur http://localhost:3000");
});
```

EXPLICATIONS

1. `const express = require('express');`

- On importe Express

2. `const router = express.Router();`

- On crée un "router" : c'est comme un organisateur de routes
- Il permet de regrouper plusieurs routes ensemble

3. `router.get('/', (req, res) => {...})`

- Quand quelqu'un visite cette route avec GET
- On renvoie un message "Liste des utilisateurs"

4. `router.post('/', (req, res) => {...})`

- Quand quelqu'un envoie des données avec POST
- On renvoie un message "Création d'utilisateur"

5. `module.exports = router;`

- On rend le router utilisable dans d'autres fichiers



INITIATION NODE.JS

CONFIGURATION ET UTILISATION DE MARIADB AVEC NODE.JS/
EXPRESS


INSTALLATION

- ▶ Pour faciliter la création d'un backend avec express nous allons installer un package globalement sur la machine : **npm install express-generator -g**
- ▶ Une fois ce package installé, vous n'aurez qu'à taper la commande suivante pour générer une config express de base :

express --no-view --git ./

INSTALLATION

- ▶ Dans votre backend nouvellement crée, installez **mysql2** et **dotenv**.
- ▶ À la racine de votre dossier backend, ajoutez maintenant un fichier **.env**

 .env

```
1  HOST='localhost'  
2  PASSWORD="password"  
3  DATABASE_NAME="database"
```

- Dans un fichier **connection.js** à la racine du backend ajoutez le code suivant :

```
JS connection.js > ...
1  const mysql = require("mysql2");
2
3  const connection = mysql.createConnection({
4    host: process.env.HOST,
5    user: "root",
6    password: process.env.PASSWORD,
7    database: process.env.DATABASE_NAME,
8    waitForConnections: true,
9    connectionLimit: 10,
10   queueLimit: 0,
11 });
12
13 connection.connect((err) => {
14   if (err) {
15     console.error("Erreur de connexion à la base de données:", err);
16     return;
17   }
18   console.log("Connexion à la base de données réussie !");
19 });
20
21 module.exports = connection;
```


UTILISATION

- ▶ Dans le fichier `/routes/index.js` nous allons pouvoir écrire nos premières requêtes !

```
router.get("/aliments", (req, res) => {  
  connection.query("SELECT * FROM aliment", (err, results) => {  
    if (err) {  
      return res.status(500).json({ error: err.message });  
    }  
    res.json(results);  
  });  
});
```

UTILISATION

```
router.post("/aliments", (req, res) => {  
  connection.query("INSERT INTO aliment SET ?", req.body, (err, results) => {  
    if (err) {  
      return res.status(500).json({ error: err.message });  
    }  
    res.json(results);  
  });  
});
```


UTILISATION

```
router.delete("/aliments/:id", (req, res) => {  
  connection.query(  
    "DELETE FROM aliment WHERE id = ?",  
    [req.params.id],  
    (err, results) => {  
      if (err) {  
        return res.status(500).json({ error: err.message });  
      }  
      res.json(results);  
    }  
  );  
});
```

UTILISATION AVEC ASYNC / AWAIT

```
router.get("/aliments", async (req, res) => {  
  try {  
    const [results] = await connection.query("SELECT * FROM aliment");  
    res.json(results);  
  } catch (err) {  
    res.status(500).json({ error: err.message });  
  }  
});
```


UTILISATION AVEC ASYNC / AWAIT

```
router.post("/aliments", async (req, res) => {
  try {
    const [results] = await connection.query("INSERT INTO aliment SET ?", [
      req.body,
    ]);
    res.json(results);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});
```

UTILISATION AVEC ASYNC / AWAIT

```
router.delete("/aliments/:id", async (req, res) => {  
  try {  
    const [results] = await connection.query(  
      "DELETE FROM aliment WHERE id = ?",  
      [req.params.id]  
    );  
    res.json(results);  
  } catch (err) {  
    res.status(500).json({ error: err.message });  
  }  
});
```




INITIATION NODE.JS

UTILISATION DU MODULE FS (FILE SYSTEM)

INTRODUCTION

fs est un module intégré de Node.js pour manipuler les fichiers. Il existe en deux versions :

- Callbacks: `const fs = require('fs')`
- Promesses (recommandé): `const fs = require('fs').promises`

EXEMPLE CALLBACK

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

function readFileContent(fileName) {
  fs.readFile(fileName, "utf8", (err, data) => {
    if (err) {
      console.error("Reading error", err);
      return;
    }
    console.log("Content:", data);
  });
}

readFileContent("file.txt");
```

EXEMPLE PROMISE

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

// Lire un fichier
async function readFileContent(fileName) {
  try {
    const data = await fs.readFile(fileName, "utf8");
    console.log("Content:", data);
  } catch (err) {
    console.error("Error:", err);
  }
}

readFileContent("file.txt");
```


AVANTAGES FS.PROMISES

Les avantages de la version Promesses :

1. Code plus propre et lisible
2. Plus facile à gérer avec `async/await`
3. Meilleure gestion des erreurs
4. Évite le "callback hell"

EXEMPLE PLUS COMPLEXE

```
// CALLBACK VERSION
const fs = require('fs');

function readAndCombineFiles(file1, file2, outputFile) {
  fs.readFile(file1, 'utf8', (err, data1) => {
    if (err) {
      console.error('Error reading first file:', err);
      return;
    }

    fs.readFile(file2, 'utf8', (err, data2) => {
      if (err) {
        console.error('Error reading second file:', err);
        return;
      }

      fs.writeFile(outputFile, data1 + data2, (err) => {
        if (err) {
          console.error('Error writing combined file:', err);
          return;
        }
        console.log('Files successfully combined!');
      });
    });
  });
}

readAndCombineFiles('file1.txt', 'file2.txt', 'combined.txt');
```


EXEMPLE PLUS COMPLEXE

```
// PROMISE VERSION
const fsPromises = require('fs').promises;

async function combineFiles(file1, file2, outputFile) {
  try {
    const data1 = await fsPromises.readFile(file1, 'utf8');
    const data2 = await fsPromises.readFile(file2, 'utf8');
    await fsPromises.writeFile(outputFile, data1 + data2);
    console.log('Files successfully combined!');
  } catch (error) {
    console.error('Error:', error);
  }
}

combineFiles('file1.txt', 'file2.txt', 'combined.txt');
```

OPÉRATIONS DE BASES

```
// 1. READ a file
async function readFile() {
  try {
    const data = await fs.readFile("file.txt", "utf8");
    console.log("File content:", data);
  } catch (error) {
    console.error("Read error:", error);
  }
}

// 2. WRITE to a file
async function writeFile() {
  try {
    await fs.writeFile("newfile.txt", "Hello World!");
    console.log("File written successfully");
  } catch (error) {
    console.error("Write error:", error);
  }
}
```


OPÉRATIONS DE BASES

```
// 3. APPEND to a file  
async function appendFile() {  
  try {  
    await fs.appendFile("file.txt", "\nNew line of text");  
    console.log("Content appended successfully");  
  } catch (error) {  
    console.error("Append error:", error);  
  }  
}
```

```
// 4. DELETE a file  
async function deleteFile() {  
  try {  
    await fs.unlink("fileToDelete.txt");  
    console.log("File deleted successfully");  
  } catch (error) {  
    console.error("Delete error:", error);  
  }  
}
```

OPÉRATIONS DE BASES

```
// 5. CREATE a directory
async function createDirectory() {
  try {
    await fs.mkdir("newFolder");
    console.log("Directory created successfully");
  } catch (error) {
    console.error("Directory creation error:", error);
  }
}

// 6. READ directory content
async function readDirectory() {
  try {
    const files = await fs.readdir("folder");
    console.log("Directory contents:", files);
  } catch (error) {
    console.error("Read directory error:", error);
  }
}
```


OPÉRATIONS DE BASES

```
// 7. CHECK if file exists
async function checkFile() {
  try {
    await fs.access("file.txt");
    console.log("File exists");
  } catch (error) {
    console.log("File does not exist");
  }
}

// 8. RENAME a file
async function renameFile() {
  try {
    await fs.rename("oldname.txt", "newname.txt");
    console.log("File renamed successfully");
  } catch (error) {
    console.error("Rename error:", error);
  }
}
```

OPÉRATIONS DE BASES

```
// 9. GET file info
async function getFileInfo() {
  try {
    const stats = await fs.stat("file.txt");
    console.log({
      size: stats.size,
      created: stats.birthtime,
      lastModified: stats.mtime,
      isFile: stats.isFile(),
      isDirectory: stats.isDirectory(),
    });
  } catch (error) {
    console.error("Stats error:", error);
  }
}

// 10. COPY a file
async function copyFile() {
  try {
    await fs.copyFile("source.txt", "destination.txt");
    console.log("File copied successfully");
  } catch (error) {
    console.error("Copy error:", error);
  }
}
```


DOCUMENTATION OFFICIELLE

<https://nodejs.org/dist/latest-v10.x/docs/api/fs.html>



PRATIQUE

Lecture de fichier

LECTURE DE FICHIER

Créer un script qui lit un fichier texte et compte le nombre de lignes qu'il contient.



PRATIQUE

Appel API

APPEL API

Créer un script qui fait appel à une API et qui affiche la réponse dans la console.



PRATIQUE

Crud file system

CRUD FILE SYSTEM

En utilisant `express` et le module `fs` créer les routes suivantes :

- GET /users : Récupère une liste d'utilisateurs
 - POST /user : Ajoute un utilisateur dans un fichier (l'uuid est généré côté serveur)
 - PUT /user/:uuid : Modifie un utilisateur
 - DELETE /user/:uuid : Supprime un utilisateur
 - GET /user/:uuid : Récupère un utilisateur par son uuid
- 💡 Installez la dépendance `uuid` pour générer des id uniques.