

Application serveur : NodeJS

Environnement pour exécuter javascript en dehors de navigateur

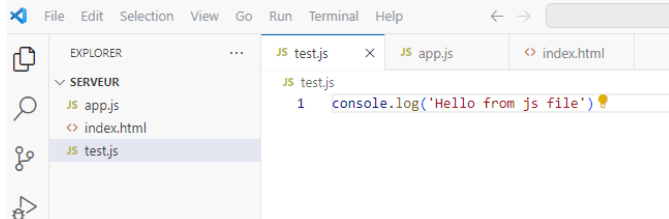
```
Invite de commandes - node
Microsoft Windows [version 10.0.22621.1194]
(c) Microsoft Corporation. Tous droits réservés.

C:\Users\kamel>node
Welcome to Node.js v18.17.1.
Type ".help" for more information.
>
```

```
Invite de commandes - node
Microsoft Windows [version 10.0.22621.1194]
(c) Microsoft Corporation. Tous droits réservés.

C:\Users\kamel>node
Welcome to Node.js v18.17.1.
Type ".help" for more information.
> console.log('hello from NodeJS')
hello from NodeJS
undefined
>
```

```
E:\Google Drive\Autres ordinateurs\HP po
MERN\Application\Serveur>node test.js
Hello from js file
```



Installer npm (node package manager) ! gestion de package dans nodejs

Créer le projet Node

Package.json

npm init

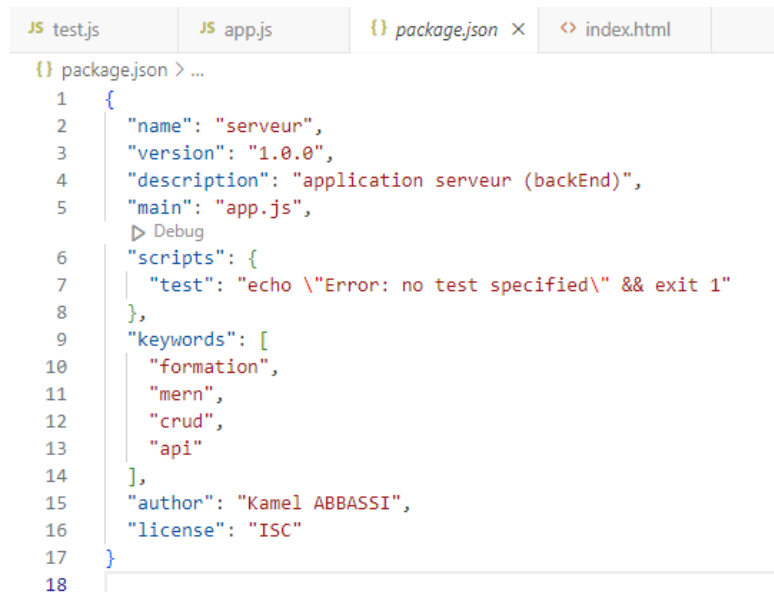
Point d'entrée: index.js

Lancer le projet

node index.js

Créer un serveur avec nodejs

1. Lancer la commande **npm init** dans le dossier serveur pour initialiser le projet
2. Reprendre aux questions comme suite :



```
JS test.js JS app.js {} package.json X <> index.html
{} package.json > ...
1 {
2   "name": "serveur",
3   "version": "1.0.0",
4   "description": "application serveur (backEnd)",
5   "main": "app.js",
6   "scripts": {
7     "test": "echo \"Error: no test specified\" && exit 1"
8   },
9   "keywords": [
10    "formation",
11    "mern",
12    "crud",
13    "api"
14  ],
15  "author": "Kamel ABBASSI",
16  "license": "ISC"
17 }
18
```

3. Ajouter un fichier index avec le code suivant
`console.log('Hello Word');`
4. Exécuter l'application
`node index.js`

```
PS C:\Users\kamel\Desktop\Application V2> node index.js
Hello Word
```
5. Installer le package **Expressjs**
<https://expressjs.com/fr>
`npm install express -save`
6. Tour d'horizon sur les applications d'expressjs (site officiel)

Pour comprendre les paramètres avec les méthodes GET et POST, vous pouvez créer des exercices simples. Voici quelques exemples d'exercices que vous pouvez utiliser pour vous familiariser avec ces concepts :

Exercice 1 : Paramètres GET

Créez une application Express qui accepte une requête GET à l'URL `"/hello"` avec un paramètre `"name"`. L'application devrait renvoyer un message de salutation en utilisant le nom fourni.

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

app.get('/hello', (req, res) => {
  const name = req.query.name;
  if (name) {
    res.send(`Hello, ${name}!`);
  } else {
    res.send('Hello, World!');
  }
})
```

```
});  
  
app.listen(3000, () => {  
  console.log('Serveur Express en cours d\'exécution sur le port 3000');  
});
```

Pour tester cet exemple :

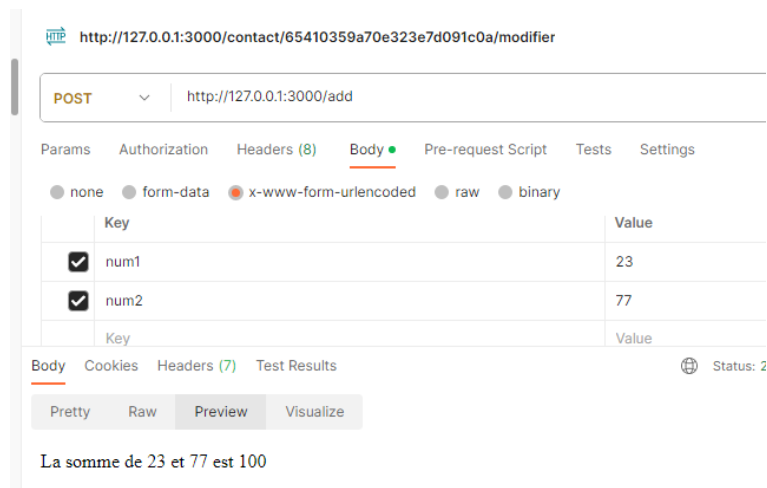
<http://localhost:3000/hello?name=abbassi>

Exercice 2 : Paramètres POST

Créez une application Express qui accepte une requête POST à l'URL `"/add"` avec deux paramètres `"num1"` et `"num2"`. L'application devrait renvoyer la somme des deux nombres.

```
const express = require('express'); // Importez le module Express  
const bodyParser = require('body-parser'); // Importez le module body-parser pour  
gérer les données POST  
const app = express(); // Créez une instance de l'application Express  
  
app.use(bodyParser.urlencoded({ extended: true })); // Utilisez body-parser pour  
analyser les données POST  
  
// Définissez une route pour gérer les requêtes POST à l'URL "/add"  
app.post('/add', (req, res) => {  
  const num1 = parseInt(req.body.num1); // Récupérez le premier paramètre "num1" de  
la requête POST  
  const num2 = parseInt(req.body.num2); // Récupérez le deuxième paramètre "num2"  
de la requête POST  
  
  if (!isNaN(num1) && !isNaN(num2)) { // Vérifiez si les paramètres sont des  
nombres valides  
    const sum = num1 + num2; // Calculez la somme des deux nombres  
    res.send(`La somme de ${num1} et ${num2} est ${sum}`); // Renvoyez la réponse  
avec la somme  
  } else {  
    res.status(400).send('Les paramètres num1 et num2 doivent être des nombres  
valides.');
```

```
  }  
});  
  
app.listen(3000, () => {  
  console.log('Serveur Express en cours d\'exécution sur le port 3000'); // Écoutez  
les requêtes sur le port 3000  
});
```



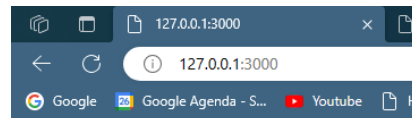
Pour ces exercices, vous pouvez utiliser Postman, Curl, ou un navigateur pour tester les requêtes GET et POST avec différents paramètres. Cela vous permettra de comprendre comment Express gère les paramètres de requête et les paramètres POST dans vos applications web.

7. Effacer le contenu du fichier index.js
8. Créer notre premier serveur

```
const express = require('express')
const bodyParser = require('body-parser');
const app = express()
const port = 3000
app.get('/', (req, res) => {
  res.send('Hello World!')
})
app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

9. Lancer le serveur

node app.js



Server worked !

Ajouter les routes suivantes

```
app.get('/home', (req, res) => {
  res.send('Bienvenue dans la page home')
})
```

Exercice 1: Initialisation de projet

Énoncé :

Utilisez la commande `npm init` pour initialiser un nouveau projet Node.js. Répondez aux questions interactives et examinez le contenu du fichier `package.json` généré.

Correction :

```
npm init
type package.json
```

Exercice 2: Installation et gestion des packages

Énoncé :

Installez-le package **lodash** en tant que dépendance de production et le package **nodemon** en tant que dépendance de développement. Vérifiez et mettez à jour le fichier package.json en conséquence.

Correction :

```
npm install lodash
npm install nodemon --save-dev
# Vérifiez et mettez à jour le fichier package.json
```

Exercice 3: Suppression de packages

Énoncé :

Supprimez-le package **lodash** de votre projet à l'aide de la commande appropriée. Vérifiez que la dépendance a été correctement supprimée du fichier package.json.

```
npm uninstall lodash
# Vérifiez et mettez à jour le fichier package.json
```

Exercice 4: Utilisation de scripts npm

Énoncé :

Ajoutez un script dans le fichier package.json qui exécute le fichier app.js avec Node.js. Exécutez ensuite ce script avec la commande npm run <nom-du-script>.

Correction :

Dans le fichier package.json, ajoutez :

```
"scripts": {
  "start": "node app.js"
}
```

Puis exécutez :

```
npm run start
```

Exercice 5: Création et publication d'un package

Énoncé :

Créez un nouveau répertoire pour un package fictif, initialisez-le avec npm init, créez un fichier JavaScript avec une fonction simple, puis publiez le package sur le registre npm.

Correction :

```
mkdir mon-package
cd mon-package
npm init
# Suivez les instructions et créez un fichier avec une fonction
# Ensuite, publiez le package
npm login
npm publish --access public
```

Exercice 6: Audit de sécurité

Énoncé :

Utilisez la commande `npm audit` pour vérifier la sécurité de vos dépendances. Identifiez une vulnérabilité et suivez les instructions pour la résoudre.

Correction :

```
npm audit
# Suivez les recommandations pour résoudre les vulnérabilités
```

Bien sûr, voici quelques exercices pour pratiquer l'approche modulaire de Node.js, en encourageant la création et l'utilisation de modules distincts. Ces exercices aideront à renforcer la compréhension de la modularité et de l'organisation du code.

Exercice 1: Création d'un module

Énoncé :

1. Créez un fichier `mathOperations.js` qui exporte deux fonctions : `add` et `multiply`.
2. Dans un autre fichier `main.js`, importez le module `mathOperations.js` et utilisez ces fonctions pour effectuer une addition et une multiplication.

Correction :

```
// mathOperations.js
module.exports = {
  add: (a, b) => a + b,
  multiply: (a, b) => a * b
};
```

```
// main.js
const mathOperations = require('./mathOperations');

const resultAdd = mathOperations.add(3, 5);
const resultMultiply = mathOperations.multiply(2, 4);

console.log('Addition:', resultAdd);
console.log('Multiplication:', resultMultiply);
```

Exercice 2: Organisation modulaire d'une application Express

Énoncé :

1. Créez un dossier `controllers` et ajoutez un fichier `UserController.js`.
2. Dans `UserController.js`, exportez une fonction `getUser` qui renvoie un objet utilisateur.
3. Dans `main.js`, utilisez ce module pour gérer une route `/user` dans une application Express.

Correction :

```
// controllers/userController.js
module.exports.getUser = () => {
  return { id: 1, name: 'John Doe' };
};
```

```
// main.js
const express = require('express');
const app = express();
const userController = require('./controllers/userController');

app.get('/user', (req, res) => {
  const user = userController.getUser();
  res.json(user);
});

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

Exercice 3: Utilisation de modules tiers

Énoncé :

1. Initialisez un projet Node.js avec `npm init`.
2. Installez le module `axios` à l'aide de la commande `npm install axios`.
3. Dans un fichier `weather.js`, utilisez le module `axios` pour faire une requête HTTP à une API météo (par exemple, OpenWeatherMap) et récupérer les données météorologiques.
4. Exportez une fonction permettant d'afficher les informations météorologiques dans `main.js`.

Correction :

```
// weather.js
const axios = require('axios');

module.exports.getWeather = async (city) => {
  try {
    const response = await
    axios.get(`https://api.openweathermap.org/data/2.5/weather?q=${city}&units=metric&appid=YOUR_API_KEY`);
    return response.data;
  } catch (error) {
    console.error('Erreur lors de la récupération des données météorologiques',
    error.message);
  }
};
```

```
// main.js
const { getWeather } = require('./weather');

const city = 'Tozeur';
getWeather(city)
  .then(weatherData => {
    console.log('Informations météorologiques pour', city, ':', weatherData);
  });
```

https://home.openweathermap.org/api_keys

Assurez-vous de remplacer `YOUR_API_KEY` par une clé API réelle d'OpenWeatherMap dans l'exemple ci-dessus.

```
C:\Users\Dell Precision\OneDrive\Bureau\Formation Web Tozeur\codes\project>node app.js
Informations météorologiques pour Tozeur : {
  coord: { lon: 8.1335, lat: 33.9197 },
  weather: [
    {
      id: 804,
      main: 'Clouds',
      description: 'overcast clouds',
      icon: '04n'
    }
  ],
  base: 'stations',
  main: {
    temp: 14.6,
    feels_like: 13.08,
    temp_min: 14.6,
    temp_max: 14.6,
    pressure: 1027,
    humidity: 37,
    sea_level: 1027,
    grnd_level: 1021
  },
  visibility: 10000,
  wind: { speed: 3.42, deg: 238, gust: 3.59 },
  clouds: { all: 100 },
  dt: 1707257093,
  sys: { country: 'TN', sunrise: 1707200468, sunset: 1707238912 },
  timezone: 3600,
  id: 2464648,
  name: 'Tozeur',
  cod: 200
}
```

L'utilisation de Node.js en REPL (Read-Eval-Print Loop) est un excellent moyen d'expérimenter rapidement avec du code JavaScript.

Voici un exercice pour pratiquer l'utilisation de Node.js en REPL :

Exercice : Manipulation de chaînes de caractères

Énoncé :

1. Lancez Node.js en mode REPL en ouvrant votre terminal et en tapant `node`.
2. Utilisez le REPL pour effectuer les opérations suivantes sur les chaînes de caractères :
 - Créez une variable `prenom` avec votre prénom.
 - Affichez la longueur de la chaîne `prenom`.
 - Convertissez la chaîne `prenom` en majuscules.
 - Concaténez la chaîne `prenom` avec une autre chaîne de votre choix.
 - Affichez les trois premiers caractères de la chaîne `prenom`.
 - Trouvez la position de la première occurrence d'une lettre spécifique dans la chaîne `prenom`.

Correction :

```
// 1. Créez une variable prenom avec votre prénom.
let prenom = 'VotrePrenom';
```



```
// 2. Affichez la longueur de la chaîne prenom.  
prenom.length;  
  
// 3. Convertissez la chaîne prenom en majuscules.  
prenom.toUpperCase();  
  
// 4. Concaténez la chaîne prenom avec une autre chaîne de votre choix.  
prenom += ' Smith';  
  
// 5. Affichez les trois premiers caractères de la chaîne prenom.  
prenom.substring(0, 3);  
  
// 6. Trouvez la position de la première occurrence d'une lettre spécifique dans la chaîne prenom.  
prenom.indexOf('S');
```

Quel intérêt de développer en asynchrone ?

Le développement asynchrone en Node.js présente plusieurs avantages significatifs, principalement liés à la nature événementielle et non bloquante de la plateforme. Voici quelques-uns des avantages clés :

1. Performances améliorées : Les opérations asynchrones permettent à Node.js de traiter plusieurs requêtes simultanément sans bloquer le thread principal. Cela améliore les performances globales, en particulier dans les applications nécessitant une gestion élevée des E/S (Entrées/Sorties).
2. Évolutivité (Scalabilité) : La conception asynchrone de Node.js est particulièrement bien adaptée aux applications nécessitant une grande extensibilité. Node.js peut gérer un grand nombre de connexions simultanées avec une utilisation efficace des ressources système.
3. Programmation basée sur les événements : Node.js suit un modèle de programmation basé sur les événements, ce qui signifie qu'il réagit aux événements et exécute le code associé. Cela facilite le traitement des événements en temps réel, comme les connexions réseau, les requêtes HTTP, les flux de données, etc.
4. Facilité de gestion des E/S : Les opérations E/S (Entrées/Sorties) sont gérées de manière asynchrone, ce qui évite le blocage du thread principal. Cela est particulièrement utile pour les applications nécessitant de nombreuses opérations E/S, comme l'accès à une base de données, la lecture/écriture de fichiers, etc.
5. Développement orienté web : Node.js est couramment utilisé pour le développement côté serveur d'applications web. L'asynchronisme facilite la manipulation de plusieurs connexions simultanées, ce qui est essentiel pour les serveurs web traitant de nombreuses demandes concurrentes.
6. Réduction des coûts : Grâce à son efficacité dans le traitement des connexions simultanées avec un seul thread, Node.js peut offrir des performances élevées avec moins de ressources matérielles par rapport à d'autres technologies.
7. Utilisation cohérente de JavaScript : Node.js permet aux développeurs de travailler avec le même langage (JavaScript) aussi bien côté client que côté serveur, ce qui favorise la cohérence et la réutilisation du code.
8. Écosystème NPM : L'écosystème NPM (Node Package Manager) de Node.js est riche en modules, bibliothèques et frameworks qui sont conçus pour fonctionner de manière asynchrone, facilitant ainsi le développement d'applications robustes et performantes.

En résumé, le développement asynchrone en Node.js permet d'améliorer les performances, l'évolutivité, la gestion des E/S, et offre une approche cohérente et efficace pour le développement d'applications web et de serveurs. Il s'agit d'une approche bien adaptée aux besoins modernes des applications en temps réel et à grande échelle.

La gestion des requêtes et réponses HTTP/HTTPS est une partie cruciale du développement en Node.js, en particulier pour la création de serveurs web.

Voici un exemple simple de la création d'un serveur HTTP et HTTPS en utilisant le module intégré `http` et le module `https` de Node.js.

```
Serveur HTTP
const http = require('http');
// Création du serveur HTTP
const server = http.createServer((req, res) => {
  // Gestion de la requête HTTP
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello, world!\n');
});

// Écoute du serveur sur le port 3000
const PORT = 3000;
server.listen(PORT, () => {
  console.log(`Serveur HTTP écoutant sur le port ${PORT}`);
});
```

Serveur HTTPS

Pour un serveur HTTPS, vous aurez besoin d'un certificat SSL. Vous pouvez générer un certificat auto-signé à des fins de test, mais dans un environnement de production, vous devriez obtenir un certificat signé par une autorité de certification.

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

// Chargement du certificat et de la clé privée pour le serveur HTTPS
const options = {
  key: fs.readFileSync('chemin/vers/clé-privée.pem'),
  cert: fs.readFileSync('chemin/vers/certificat.pem')
};

// Création du serveur HTTPS
const secureServer = https.createServer(options, (req, res) => {
  // Gestion de la requête HTTPS
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello, secure world!\n');
});

// Écoute du serveur HTTPS sur le port 3001
const SECURE_PORT = 3001;
secureServer.listen(SECURE_PORT, () => {
  console.log(`Serveur HTTPS écoutant sur le port ${SECURE_PORT}`);
});
```

Assurez-vous de remplacer les chemins vers votre clé privée (`clé-privée.pem`) et votre certificat (`certificat.pem`) par les vrais chemins vers vos fichiers.

Ces exemples créent des serveurs HTTP et HTTPS simples qui renvoient "Hello, world!" et "Hello,secure world!" respectivement. Vous pouvez ajouter plus de logique de gestion des requêtes en fonction des besoins de votre application.

Le cœur de Node.js offre de nombreux modules intégrés qui couvrent divers aspects de la programmation, allant des opérations de fichiers aux communications réseau. Voici quelques-uns des modules principaux que Node.js offre :

D'accord, voici un exemple simple d'utilisation pour chaque module que j'ai mentionné précédemment :

1. ``fs`` (File System): Fournit des fonctionnalités pour travailler avec le système de fichiers, permettant la lecture, l'écriture, la suppression et la manipulation des fichiers

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

// Lecture d'un fichier
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data);
});
```

2. ``http`` et ``https``:

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

// Création d'un serveur HTTP
const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, World!\n');
});

server.listen(3000, () => {
  console.log('Server running on http://localhost:3000/');
});
```

3. ``net``: Facilite la création de serveurs et de clients pour les connexions TCP.

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

// Création d'un serveur TCP
const server = net.createServer((socket) => {
  socket.write('Hello TCP!\r\n');
  socket.pipe(socket);
});

server.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

4. ``dgram``: Permet de créer des serveurs et des clients pour les connexions UDP

```
const dgram = require('dgram');
const server = dgram.createSocket('udp4');

// Création d'un serveur UDP
server.on('message', (msg, rinfo) => {
  console.log(`Server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
});

server.bind(3000, () => {
  console.log('Server listening on port 3000');
});
```

5. ``events``: Fournit une infrastructure pour travailler avec les événements. Il est utilisé par de nombreux autres modules Node.js.

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

// Écouteur d'événement
myEmitter.on('customEvent', (arg) => {
  console.log(`Event triggered with argument: ${arg}`);
});

// Émission de l'événement
myEmitter.emit('customEvent', 'Hello, EventEmitter!');
```

6. ``path``: Facilite la manipulation des chemins de fichiers et de répertoires.

```
const path = require('path');
// Concaténation de chemins
const fullPath = path.join(__dirname, 'folder', 'file.txt');
console.log(fullPath);
```

7. ``os``: Fournit des méthodes pour interagir avec le système d'exploitation, telles que la récupération d'informations sur le CPU, la mémoire, etc

```
const os = require('os');
// Informations sur le système
console.log(`Total Memory: ${os.totalmem()} bytes`);
console.log(`Free Memory: ${os.freemem()} bytes`);
```

8. ``util``: Contient diverses utilitaires qui peuvent être utiles pour le développement, comme la gestion des promesses.

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

// Utilisation de util.promisify
const readFileAsync = util.promisify(fs.readFile);

readFileAsync('example.txt', 'utf8')
  .then((data) => console.log(data))
  .catch((err) => console.error(err));
```

9. ``crypto``: Offre des fonctionnalités de cryptographie, telles que la génération de hachages et la création de certificats.

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

```
// Génération d'un hachage
const hash = crypto.createHash('sha256');
hash.update('Hello, Crypto!');
console.log(hash.digest('hex'));
```

10. `querystring` : Facilite le traitement des chaînes de requête URL.

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

// Encodage et décodage de chaînes de requête
const params = { name: 'John', age: 30, city: 'New York' };
const encodedParams = querystring.stringify(params);
console.log(encodedParams);

const decodedParams = querystring.parse(encodedParams);
console.log(decodedParams);
```

11. `url` : Permet de travailler avec les objets URL.

```
const url = require('url');
// Analyse d'une URL
const urlString = 'https://www.example.com/path?query=string';
const parsedUrl = url.parse(urlString, true);
console.log(parsedUrl);
```

2. Construction d'un squelette d'application :

```
// app.js
const express = require('express');
const app = express();
const port = 3000;
// Middleware pour servir des fichiers statiques
app.use(express.static('public'));
// Middleware pour le traitement du corps de la requête (pour la gestion des formulaires)
app.use(express.urlencoded({ extended: true }));
app.use(express.json());
// Définition d'une route de base
app.get('/', (req, res) => {
  res.send('Hello, Express!');
});

// Écoute du serveur sur le port spécifié
app.listen(port, () => {
  console.log(`Server listening at http://localhost:${port}`);
});
```

3. Configuration d'Express et de l'application :

Vous pouvez ajouter des configurations spécifiques à votre application, par exemple :

```
// Configuration d'EJS comme moteur de modèle
app.set('view engine', 'ejs');
```

Il faut installer npm install ejs

4. Le rendu de vues avec EJS :

Créez un fichier de modèle EJS (`views/index.ejs`):

```
<!-- views/index.ejs -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Express EJS Example</title>
</head>
<body>
  <h1><%= message %></h1>
</body>
</html>
```

Utilisez ce modèle dans votre application Express :

```
// Utilisation du moteur de modèle EJS
app.get('/ejs', (req, res) => {
  res.render('index', { message: 'Hello from EJS!' });
});
```

5. Gestion de formulaires et des uploads de fichiers :

```
<!-- views/form.ejs -->
<form action="/submit" method="post" enctype="multipart/form-data">
  <label for="username">Username:</label>
  <input type="text" id="username" name="username" required>

  <label for="avatar">Choose an avatar:</label>
  <input type="file" id="avatar" name="avatar" accept="image/*">

  <button type="submit">Submit</button>
</form>
```

```
const express = require("express");
const app = express();
const port = 3000;
const bodyParser = require('body-parser');
const multer = require('multer');
const path = require('path');

app.set('view engine', 'ejs');
app.use(bodyParser.urlencoded({ extended: true }));

// Définir le moteur de stockage
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'uploads/'); // Dossier de destination pour les fichiers téléchargés
  },
  filename: (req, file, cb) => {
    cb(null, file.fieldname + '-' + Date.now() + path.extname(file.originalname));
  },
});

// Initialiser multer
const upload = multer({
  storage: storage,
  limits: { fileSize: 1024 * 1024 * 5 }, // Limite de taille de fichier à 5 Mo
});

app.get('/ejs', (req, res) => {
  res.render('index', { message: 'Hello from EJS!' });
});
```

```
app.get('/form', (req, res) => {
  res.render('form');
});

// Traitement du formulaire
app.post('/submit', upload.single('avatar'), (req, res) => {
  const username = req.body.username;
  const avatar = req.file; // Utilisez req.file au lieu de req.files
  // Logique de traitement
  if (!avatar) {
    return res.status(400).send('Aucun fichier n\'a été téléchargé.');
```

```
  }

  res.send(`Username: ${username}, Avatar: ${avatar.filename}, Extension:
${path.extname(avatar.originalname)}`);
});

app.listen(port, () => {
  console.log(`Serveur démarré http://localhost:${port}`);
});
```

6. Routage d'URL par Express :

Créez un fichier de route (`routes/users.js`):

```
// routes/users.js
const express = require('express');
const router = express.Router();

// Définition des routes spécifiques
router.get('/', (req, res) => {
  res.send('Users Home');
});

router.get('/profile', (req, res) => {
  res.send('User Profile');
});
module.exports = router;
```

Utilisez ce routeur dans votre application principale :

```
// Utilisation du routeur
const usersRouter = require('./routes/users');
app.use('/users', usersRouter);
```

7. Mise en place d'une API REST :

```
// API REST simple
const books = [
  { id: 1, title: 'Node.js in Action' },
  { id: 2, title: 'Express.js Guide' },
];

// Obtenir tous les livres
app.get('/api/books', (req, res) => {
  res.json(books);
});

// Obtenir un livre par ID
app.get('/api/books/:id', (req, res) => {
  const book = books.find(b => b.id === parseInt(req.params.id));
  if (!book) return res.status(404).send('Book not found');
  res.json(book);
});
```

```
});  
  
// Ajouter un nouveau livre  
app.post('/api/books', (req, res) => {  
  const book = { id: books.length + 1, title: req.body.title };  
  books.push(book);  
  res.json(book);  
});  
  
// Mettre à jour un livre par ID  
app.put('/api/books/:id', (req, res) => {  
  const book = books.find(b => b.id === parseInt(req.params.id));  
  if (!book) return res.status(404).send('Book not found');  
  book.title = req.body.title;  
  res.json(book);  
});  
  
// Supprimer un livre par ID  
app.delete('/api/books/:id', (req, res) => {  
  const bookIndex = books.findIndex(b => b.id === parseInt(req.params.id));  
  if (bookIndex === -1) return res.status(404).send('Book not found');  
  const deletedBook = books.splice(bookIndex, 1);  
  res.json(deletedBook[0]);  
});
```