Examen Théorique sur Node.js

SOKA Dissima

2024-11-14

1 Introduction à Node.js

1.1 Quelle est la définition de Node.js et à quoi sert-il ?

Node.js est une plateforme d'exécution JavaScript construite sur le moteur JavaScript V8 de Google Chrome. Il permet d'exécuter du code JavaScript côté serveur, ce qui facilite le développement d'applications web évolutives et performantes, notamment pour les applications en temps réel.

1.2 Quel est l'intérêt d'utiliser Node.js pour les applications modernes ?

Node.js est performant pour les applications qui nécessitent un traitement asynchrone et en temps réel, comme le chat, les jeux en ligne et le streaming. Il offre une exécution rapide grâce à son modèle non bloquant, optimisant les performances et la scalabilité des applications.

1.3 Expliquer les concepts de "Single Thread" et "Event Loop" dans Node.js.

Node.js utilise un modèle de thread unique (Single Thread) combiné à une boucle d'événements (Event Loop) pour gérer les opérations asynchrones. Cela permet à Node.js de traiter de nombreuses requêtes simultanément sans bloquer l'exécution, en déléguant les opérations d'entrée/sortie au système sous-jacent.

1.4 Décrire les caractéristiques principales de Node.js.

Les principales caractéristiques de Node.js incluent :

- **Non-bloquant** : Grâce à son modèle asynchrone, Node.js est idéal pour les applications nécessitant des réponses rapides.
- **Événementiel** : Utilise un modèle d'événements pour gérer les opérations.
- **Open-source** : Node.js est gratuit et bénéficie d'une large communauté.
- **Cross-platform** : Fonctionne sur divers systèmes d'exploitation comme Windows, macOS et Linux.

2 Modules et Gestion des Dépendances

2.1 Qu'est-ce qu'un module dans Node.js ?

Un module dans Node.js est un fichier contenant du code JavaScript réutilisable. Les modules aident à organiser le code en unités logiques, facilitant ainsi la maintenance et la réutilisabilité.

2.2 Qu'est-ce que "npm" et comment l'utiliser pour gérer les dépendances ?

"npm" (Node Package Manager) est le gestionnaire de paquets par défaut de Node.js, permettant l'installation, la mise à jour, et la gestion de bibliothèques externes. Pour installer une dépendance, on utilise la commande `npm install <nom-du-paquet>`, qui ajoute la dépendance au fichier `package.json`.

2.3 Comment créer un module dans Node.js ?

Pour créer un module dans Node.js, on écrit du code dans un fichier JavaScript et on exporte les fonctions ou objets avec `module.exports`. Par exemple :

```
// myModule.js
function sayHello() {
    console.log("Hello, world!");
}
module.exports = sayHello;
```

Ensuite, le module peut être importé avec `require('./myModule')`.

2.4 Donner des exemples de modules principaux de Node.js.

Quelques modules principaux de Node.js incluent :

- **http** : pour créer des serveurs HTTP.
- **fs** : pour manipuler le système de fichiers.
- **path** : pour travailler avec les chemins de fichiers.
- **os** : pour obtenir des informations sur le système d'exploitation.

3 Gestion des APIs et Routage

3.1 Expliquer ce qu'est une API REST.

Une API REST (Representational State Transfer) est une interface qui utilise des méthodes HTTP pour permettre la communication entre un client et un serveur, en se basant sur des ressources identifiées par des URL. Les opérations principales incluent GET, POST, PUT et DELETE.

3.2 Quelles sont les principales méthodes HTTP utilisées dans les API REST ? Donner un exemple d'utilisation pour chaque méthode.
Les principales méthodes HTTP incluent :

- **GET** : pour récupérer des ressources. Ex : `GET /users` pour obtenir la liste des utilisateurs.
- **POST** : pour créer une nouvelle ressource. Ex : `POST /users` pour ajouter un utilisateur.
- **PUT** : pour mettre à jour une ressource existante. Ex : `PUT /users/1` pour modifier l'utilisateur ayant l'ID 1.
- **DELETE** : pour supprimer une ressource. Ex : `DELETE /users/1` pour supprimer l'utilisateur ayant l'ID 1.

3.3 Comment gérer les routes dans une application Node.js avec Express ?
Dans une application Node.js avec Express, les routes sont définies en utilisant les méthodes HTTP disponibles dans Express. Par exemple :

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

app.get('/users', (req, res) => {
    res.send("Liste des utilisateurs");
});

app.post('/users', (req, res) => {
    res.send("Nouvel utilisateur créé");
});
```

Chaque route est associée à une URL spécifique et à une fonction de rappel qui gère la requête.

4 Asynchronisme et Gestion des Promesses

4.1 Qu'est-ce qu'une fonction asynchrone ? Expliquer son utilité.
Une fonction asynchrone en JavaScript est une fonction qui permet d'exécuter des opérations non bloquantes, souvent en utilisant des promesses. L'utilité d'une fonction asynchrone est d'améliorer la performance et la réactivité d'une application, notamment en exécutant des tâches lourdes ou dépendantes de réponses externes sans bloquer le flux principal du programme.

4.2 Qu'est-ce qu'une Promesse en JavaScript ? Expliquer les états possibles d'une Promesse.
Une Promesse en JavaScript est un objet représentant la réalisation ou

l'échec éventuel d'une opération asynchrone. Les états possibles d'une promesse sont :

- **Pending** : la promesse est en cours d'exécution.
- **Fulfilled** : l'opération asynchrone est terminée avec succès.
- **Rejected** : l'opération asynchrone a échoué.

4.3 Expliquer avec un exemple comment utiliser `async` et `await` pour gérer les Promesses en JavaScript.
En JavaScript, `async` et `await` sont utilisés pour simplifier l'utilisation des promesses. Voici un exemple :

```
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();
```

Dans cet exemple, la fonction `fetchData` utilise `await` pour attendre la résolution de chaque promesse. Si une erreur se produit, elle est capturée dans le bloc `catch`.

4.4 Qu'est-ce qu'une fonction de rappel (callback) en JavaScript et comment peuvent-elles être utilisées avec les Promesses ? Donner un exemple.
Une fonction de rappel, ou *callback*, est une fonction passée en argument à une autre fonction et exécutée après la fin de cette dernière. Les *callbacks* peuvent être utilisés avec des promesses pour exécuter du code après l'achèvement d'une tâche asynchrone. Exemple :

```
function getData(callback) {
    setTimeout(() => {
        callback("Données récupérées");
    }, 1000);
}

getData((message) => {
    console.log(message);
});
```

5 Proxies JavaScript

5.1 Qu'est-ce qu'un Proxy en JavaScript et comment fonctionne-t-il ?

Un Proxy en JavaScript est un objet qui permet de personnaliser les opérations fondamentales (comme la lecture et l'écriture) sur un objet. En créant un Proxy, on peut intercepter et redéfinir les comportements de base de l'objet cible.

5.2 Donner un exemple de Proxy interceptant la lecture et l'écriture de propriétés sur un objet.

Exemple de Proxy interceptant la lecture et l'écriture :

```
let target = { name: "soka" };
let handler = {
    get: (obj, prop) => {
        return prop in obj ? obj[prop] : "Propriété inexistante";
    },
    set: (obj, prop, value) => {
        console.log(`Modification de ${prop} à ${value}`);
        obj[prop] = value;
    }
};
let proxy = new Proxy(target, handler);

console.log(proxy.name); // Alice
console.log(proxy.age); // Propriété inexistante
proxy.age = 25; // Modification de age à 25
```