Node JS - Activités N01

Activité N01

Application pratique

1. Creation du fichier `mathA.js`

```
const PI=3.14

function calculerAire(rayon){
  return PI*rayon*rayon;
}

console.log(calculerAire(2));
```

2. Creation du fichier `mathB.js`

```
const PI=3.14

function calculerAire(rayon){
  return PI*rayon*rayon;
}

console.log(calculerAire(2));
```

3. l'execution des deux fichiers a travers le terminal

```
node mathA.js
node mathB.js
```

4. creation du projet `app.js`

```
const PI=3.14

function calculerAire(rayon){
  return PI*rayon*rayon;
}

console.log(calculerAire(2));

const PI=3.14

function calculerAire(rayon){
  return PI*rayon*rayon;
}

console.log(calculerAire(2));
```

5. l'execution du fichier `app.js`

```
node app.js
```

Responses aux questions

1. *Que se passe-t-il si plusieurs fichiers définissent les mêmes noms ?*
   - Il cause des conflits de variables globales a cause de la redeclaration des constantes.
2. *Pourquoi est-ce un problème dans un projet réel ?*
   - Risque d'écrasement de variables, fonctions (overriding) ou bien rencontrer des erreurs.
3. *Comment le système de modules de Node résout-il ce problème ?*
   - Chaque module a son propre scope des variables, fonctions ... alors qu'ils ne sont pas globales sans les exporter/importer a travers les keyword `export` / `import`

Activité N02

Application pratique

1. L'export d'une seule fonction dans `greet.js`

```
function saluer(nom) {
  return `Bonjour, ${nom} !`;
}

module.exports = saluer;
```

2. l'import de la fonction `saluer()` dans `app.js`

```
const saluer = require("./utilities/greet");
console.log(saluer("Youness"));
```

3. L'export d'un objet avec plusieurs fonctions `greet.js`

```
function saluer(nom) {
  return `Bonjour, ${nom} !`;
}

function direAuRevoir(nom) {
  return `Au revoir, ${nom} !`;
}

module.exports = { saluer, direAuRevoir };
```

4. Import de l'objet contenant les deux fonctions dans `app.js`

```
const greetings = require("./utilities/greet");
console.log(greetings.saluer("Youness"));
console.log(greetings.direAuRevoir("Youness"));
```

5. L'execution du programme

```
node app.js
```

Responses aux questions

1. *Quelle différence entre `module.exports = fonction` et `module.exports = { ... }` ?*
   - `module.exports = fonction` exporte une seule fonction comme module
   - `module.exports = { ... }` exporte un objet contenant plusieurs fonctions/variables
2. *Dans quels cas préférez-vous l'un ou l'autre ?*
   - **Export simple** : quand le module a une seule fonction principale à exporter
   - **Export d'objet** : quand le module contient plusieurs fonctions liées qu'on veut grouper ensemble

Activité N03

Application pratique

1. Création du fichier `test.js`

```
console.log(__filename);
console.log(__dirname);
console.log(module);
console.log(exports === module.exports);

exports.direSalut = () => console.log("Salut !");
console.log(module.exports);
```

2. Création d'un fichier `importer.js` pour tester l'import

```
const mod = require('./test');
mod.direSalut();
```

3. l'execution a travers le terminal

```
node test.js
```

Responses aux questions

1. *Que signifient `__filename`, `__dirname`, `module`, et `exports` ?*

- `__filename` : contient le chemin absolu du fichier en cours d'exécution
- `__dirname` : contient le chemin absolu du dossier contenant le fichier en cours
- `module` : objet représentant le module actuel, avec ses métadonnées et exports
- `exports` : raccourci pour `module.exports`, utilisé pour exporter des fonctionnalités

2. *Pourquoi `exports = fonction()` ne fonctionne pas comme prévu ?*
   - Parce que `exports` est seulement une référence à `module.exports`. Réassigner `exports` casse cette référence sans affecter `module.exports` qui est le vrai point d'exportation.
3. *Quelle relation existe entre `exports` et `module.exports` ?*
   - `exports` est un alias de `module.exports` au début du module. Ils pointent vers le même objet, mais seule `module.exports` est retournée par `require()`.

Activité N04

Application pratique

1. Création du fichier `contactService.js`

```
const contacts = [];
function ajouterContact(nom, telephone) {
    contacts.push({ nom, telephone });
}
function listerContacts() {
    return contacts;
}

module.exports = { ajouterContact, listerContacts };
```

2. Création du fichier `utils/format.js`

```
function formaterContact(contact) {
    return `${contact.nom} - ${contact.telephone}`;
}

module.exports = formaterContact;
```

3. Création du fichier `app.js`

```
const { ajouterContact, listerContacts } = require('./contactService');
const formaterContact = require('./utils/format');

ajouterContact("Alice", "0600000000");
ajouterContact("Bob", "0611111111");

listerContacts().forEach(c => console.log(formaterContact(c)));
```

4. l'execution du programme a travers le terminal

```
node app.js
```

Reponses aux questions

1. *Quelle est la responsabilité de chaque module ?*
   - `contactService.js` : Gère la logique métier (ajout et liste des contacts)
   - `utils/format.js` : S'occupe du formatage des données d'affichage
   - `app.js` : Point d'entrée de l'application, coordonne les différents modules
2. *Pourquoi séparer le formatage, la logique et le point d'entrée ?*
   - Pour respecter le principe de séparation des concerns (chaque module a une responsabilité unique)
   - Permet de modifier l'affichage sans toucher à la logique métier
   - Facilite les tests unitaires sur chaque partie indépendamment
3. *Comment cela faciliterait la maintenance à long terme ?*
   - Évolutivité : on peut ajouter de nouvelles fonctionnalités sans tout réécrire
   - Réutilisabilité : les modules peuvent être utilisés dans d'autres parties du projet
   - Débogage : plus facile de localiser les erreurs dans des modules spécifiques
   - Collaboration : plusieurs développeurs peuvent travailler sur différents modules simultanément