Exercices API RESTful : fondamentaux Javascript, Node.js et Express.js

Paul Schuhmacher

Octobre 2023

Module: API

> *Conseil : créer un dossier pour chaque exercice*

Pré-requis

Maîtrise des bases de Javascript (variables, blocs, fonctions, programmation asynchrone)

Exercice 0 : Revoir les bases

- Suivre le guide Premiers pas en JavaScript de la MDN pour apprendre la déclaration et l'utilisation de variables
- Suivre le guide Principaux blocs en JS de la MDN, pour apprendre l'usage des conditions, des boucles et l'écriture de fonctions. Faire l'auto-évaluation
- Suivre le guide JavaScript asynchrone de la MDN, pour apprendre les bases de la programmation asynchrone. Faire l'évaluation

Exercice 1 : Javascript dans l'environnement Node.js

> *Notions abordées: écriture de fonctions, programmation asynchrone, manipulation de fichiers, entrée et sortie standards, génération de nombre aléatoire*

Utiliser l'environnement Node.js pour écrire un jeu en ligne de commande que l'on placera dans un fichier **`game.js`**.

1. Initialiser un projet Node.js avec la commande **`npm init`**
2. Créer un fichier **`game.js`** qui contiendra le code source du jeu
3. Utiliser le module intégré **`process`** pour accéder à la lecture (**`stdin`**) et l'écriture (**`stdout`**) sur la sortie standard (canal de communication entre le programme et le joueur)

```
//Code de départ
console.log('Le jeu commence');
```

```
console.log('(Control-D pour quitter le jeu)');
//Initialisation du stream depuis l'entrée standard
process.stdin.resume();
//Fonction exécutée lorsque le joueur écrit sur l'entrée standard et
        termine par le caractère
//de retour à la ligne 'Retour chariot' (touche Entrée)
process.stdin.on('data', function (data) {
    console.log(`Vous avez tapé ${data}`);});
```

1. Implémenter le jeu suivant : le programme génère un nombre aléatoire compris entre 1 et 100, et le joueur doit deviner ce nombre. Le joueur dispose d'un nombre limité de tentatives pour deviner correctement. Pour ce faire, le joueur entre un nombre dans le terminal via l'entrée standard. Après chaque tentative infructueuse, le programme affiche un indice sur la sortie standard pour indiquer si le nombre était trop grand ou trop petit. Le jeu maintient également un décompte du nombre de tentatives utilisées par le joueur. Si le joueur atteint le nombre maximal de tentatives sans deviner correctement, le jeu affiche **`"Game Over : le nombre à trouver était X"`** et se termine. Si le joueur parvient à deviner correctement le nombre, avant d'utiliser toutes ses tentatives, le jeu affiche un message de félicitations avec le nombre de tentatives effectuées, par exemple, "Félicitations, le nombre à deviner était bien X ! Vous avez trouvé en N tentatives !"
2. Le jeu doit proposer de rejouer en affichant le message **`"Voulez-vous rejouer une partie ? ([Y]/n)"`**. Lorsque que l'utilisateur·ice appuie sur sur la touche **`Entrée`** ou **`y`** une partie est relancée, le programme se termine sinon.
3. A quoi sert l'option **`node --watch`** ? A quoi sert **`node --check`** ? **Tester-les**.
4. Ajouter un script dans le **`package.json`** pour lancer le jeu avec la commande **`npm run start-game`**
5. Ajouter un script dans le **`package.json`** pour checker la syntaxe de votre code source avec la commande **`npm run check`**

Bonus :

- **Implémenter** une possibilité d'enregistrer l'historique des parties dans un simple fichier texte. A la fin de chaque partie *gagnée*, enregistrer le nombre d'essais, le nombre d'essais max et les bornes **`min`** et **`max`**. Toutes ces données permettent de définir une difficulté et de calculer des scores *a posteriori*. Pour cela, utiliser le module **`core:fs`** (*file system*)
- **Implémenter** *un chronomètre* pour définir une durée de partie maximale, par exemple 1 minute. Le jeu doit afficher le temps restant à chaque tentative sur la sortie standard. Ajouter la durée de la partie en secondes et la durée max d'un jeu à la ligne de score dans le fichier.

Ressource utiles : process.stdin, process.stdout, module fs

Exercice 2 : Javascript, écriture de fonctions

Notions abordées: écriture de fonctions, reduce, filter, fonctions anonymes, JSDocs, random, first-citizen functions, récursivité, manipulation des dates et des timezone,

test léger avec assert

Écrire les fonctions qui réalisent les tâches suivantes :

1. Une fonction qui prend en argument une liste de nombres et retourne le nombre le plus grand. La fonction doit renvoyer null si l'argument n'est pas un tableau ou si le tableau est vide. Écrire un commentaire au format JSDocs pour documenter votre fonction. **Générer** la documentation de votre fonction avec le paquet jsdoc
2. Une fonction *récursive* qui effectue la somme d'une liste de nombres.
3. Une fonction **`sumOnRange`** qui prend en argument une liste d'entiers et deux entiers **`a`** et **`b`**, avec **`a < b`**. Cette fonction doit retourner la somme des nombres de la liste compris entre **`a`** et **`b`** (inclus). Par exemple, si on fournit la liste **`[1, 2, 3, 4]`**, **`a=1`**, **`b=3`**, la fonction doit renvoyer la somme de liste filtrée **`[1, 2, 3]`**, soit **`6`**. Si **`a > b`**, la fonction doit lancer une erreur et le programme doit échouer de manière gracieuse en affichant le message **`'a doit être plus petit que b'`**.
4. On aimerait *généraliser* la fonction **`sumInRange`**. **Réécrire** la fonction précédente que l'on appellera **`applyOnRange`** de sorte à ce que l'on puisse lui *passer en argument une fonction f* afin de changer la nature de l'opération réalisée sur la sélection de nombres (actuellement la *somme*). Tester votre fonction en lui passant en paramètre une fonction anonyme qui calcule le *produit de tous les nombres*, et une fonction anonyme qui calcule *la somme des carrés*.
5. Une fonction qui prend *un nombre indéterminé d'arguments* où chaque argument est un mot. La fonction doit retourner les mots sous forme de chaîne de caractères, où les mots sont triés par taille, du plus petit au plus grand, séparés le caractère **`(/)`**. Tester votre fonction avec la chaîne de caractères suivante : **`Cat, Sunshine, Bicycle, Fish, Harmony, Dart, Pillow, Atmosphere, Whisper, Chocolate, Adventure`**. **Bonus :** tout en conservant l'ordonnancement par taille, ajouter un tri alphabétique. Par exemple, **`Cat`** doit apparaître avant **`Dog`**. Sortie attendue :
   `Dog/Cat/Fish/Pillow/Bicycle/Harmony/`
   `Whisper/Sunshine/Chocolate/Adventure/Javascript/Atmosphere`
6. Une fonction qui renvoie vrai si une chaîne de caractères est un palindrome, faux sinon. Tester votre fonction avec le palindrome **`"Engage le jeu que je le gagne"`**.
7. Une fonction qui calcule votre âge (en secondes) à l'instant actuel, à la seconde près. Affichage attendu **`"1303236963 secondes vécues"`**.
8. Une fonction qui affiche l'heure courante à Paris, Londres et Sydney, formaté *au format français*. *Aide* : Utiliser la méthode Intl.DateTimeFormat du paquet core d'internationalisation **`intl`** de Node. Sortie attendue :

```
Heure à Paris : 11:41:51 UTC+2
Heure à Londres : 10:41:51 UTC+1
Heure à Sydney : 20:41:51 UTC+11
```

9. *Fizzbuzz*, avant d'être utilisé par les recruteur·euses pour faire passer des tests techniques, est un jeu pour apprendre la division aux enfants. Les règles sont simples : il faut compter jusqu'à un certain nombre *positif* qu'on se fixe à l'avance : si le nombre est

divisible par 3 on le remplace par `"Fizz"`, si il est divisible par 5 par `"Buzz"`, s'il est divisible par 3 et 5, comme 15, par `"Fizz Buzz"`. Sinon on se contente de dire le nombre. **Écrire** la fonction `String fizzbuzzIterative(int n)` qui retourne la réponse du jeu sous forme de chaîne de caractère, où `n` représente la taille du jeu. Par exemple, `fizzbuzzIterative(15)` retournera `"1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 Fizz Buzz"`. **Réécrire** la fonction de manière *récursive* (sans structure de contrôle de boucle). **Écrire un test** (avec `console.assert()`) qui permet de s'assurer que les deux fonctions renvoient bien la même réponse. Est-ce que ce test est suffisant pour avoir confiance en nos implémentations ?

10. Une fonction `chooseOperation` qui prend en argument deux entiers `a` et `b`. Si `a` > `b`, la fonction doit retourner une fonction qui effectue le produit deux nombres, une fonction qui effectue la division entière de deux nombres sinon.
11. Une fonction `deriv` qui retourne la *dérivée* d'une fonction `f` mathématique à une seule variable. On rappelle que la dérivée d'une fonction en un point x est définie par `(f(x+dx)-f(x)/dx)` avec `dx << 1`. On prendra `dx=0.000001`. Évaluer la fonction qui à x associe x^3 en `x=2`. Évaluer sa dérivée au même point. Sortie attendue : `8 et 12`
12. Une fonction `addItems(list, ...elements)` qui ajoute des éléments `elements` à un tableau *sans modifier le tableau original* et retourne le résultat.

```
const list = [1,2,3];
const newList = addItems(list, 4, 5, 6);
console.log(list) // [1,2,3]
console.log(newList) // [1,2,3,4,5,6]
```

Exercice 3 : Javascript, programmation asynchrone

Notions abordées: programmation asynchrone (Promesse, async, await), client HTTP, json

1. Écrire un programme qui requête l'API jsonplaceholder et récupère tous les commentaires d'un post (via son identifiant unique). **Afficher** la réponse sous forme de chaîne de caractères. Écrire le programme de manière à effectuer la requête de manière *asynchrone*. Le programme doit afficher `"Fetching data... Please wait a moment."` *pendant* que le traitement a lieu.
2. Une fois la réponse obtenue, **afficher** uniquement les emails des auteurs de chaque commentaire.
3. Le programme doit également travailler sur les `posts` d'un utilisateur. **Écrire** une nouvelle fonction asynchrone pour afficher le nombre de posts d'un l'utilisateur.
4. Une nouvelle requête doit être faite sur la ressource `/comment`, qui n'existe pas. **Écrire** la requête et *échouer avec grâce* en gérant l'erreur et en affichant `"The ressource was not found"`.

Pour faire des requêtes HTTP, vous pouvez utiliser le module http (mais ce sera à vous d'implémenter l'interface asynchrone avec des Promesses, plus challenge) ou axios, qui a une API basée sur les promesses (plus facile à utiliser)

Exercice 4 : Créer et utiliser un module Node.js

Créer un module Node.js nommé **`geometry`** qui exporte des fonctions pour effectuer des opérations de calcul géométrique.

1. Créez un dossier pour votre projet d'exercice et initialisez un projet Node.js en utilisant **`npm init`**. Suivez les instructions pour configurer votre projet.
2. Créez un fichier JavaScript nommé **`geometry.js`**. Ce fichier sera votre module.
3. Dans **`geometry.js`**, déclarez une fonction **`perimeterOfCircle`** qui prend en paramètre le rayon d'un cercle et renvoie son périmètre. Pour calculer le périmètre d'un cercle, vous aurez besoin de la valeur de pi. Déclarer un objet **`constants`** qui contient les valeurs suivantes

```
const constants = {
    PI: 4 * Math.atan(1.0),
    _2PI: 2 * 4 * Math.atan(1.0),
};
```

*On pourrait également utiliser **`Math.PI`**.*

4. Exportez la fonction **`perimeterOfCircle`** en tant que méthode du module en utilisant **`module.exports`** :

```
module.exports.perimeterOfCircle = perimeterOfCircle;
```

5. Créez un fichier JavaScript principal (par exemple, **`app.js`**) dans le même dossier que **`geometry.js`**. Dans **`app.js`**, importez le module **`geometry`** que vous avez créé :

```
const geometry = require('./geometry');
```

6. Utilisez la fonction **`perimeterOfCircle`** du module pour effectuer une opération de calcul et affichez le résultat :

```
const result = geometry.perimeterOfCircle(5);
console.log(result);
```

7. Exécutez **`app.js`** en utilisant Node.js pour vérifier que le module **`geometry`** fonctionne correctement
8. Ajouter d'autres fonctions :
    1. **`surfaceOfCircle(radius)`** qui prend en argument le rayon d'un cercle et retourne son aire

2. **`volumeOfSphere(radius)`** qui prend en argument le rayon d'une sphère et retourne son volume
3. **`distance(x1, y1, x2, y2)`** qui calcule la distance entre deux points situés aux coordonnées (x1,y1) et (x2,y2)
4. **`areOverlapping(x1, y1, radius1, x2, y2, radius2)`** qui retourne vrai si le cercle situé en x1, y1 et de rayon radius1 recouvre le cercle situé en x2, y2 et de rayon radius2, faux sinon

9. Exportez tout le contenu du fichier *sauf* **`constants`** et **`distance(x1, y1, x2, y2)`**. Dans **`app.js`**, tester chacune des fonctions exportées.

Ressource utile : JavaScript modules

Exercice 5 : Serveur web avec Node.js

L'objectif de cet exercice est de créer un serveur web simple en utilisant Node.js, **sans l'aide du framework Express**. Vous allez mettre en place un serveur qui répondra aux requêtes HTTP avec des réponses statiques.

1. Initialisez un projet Node.js en créant un dossier pour l'exercice et en exécutant **`npm init`**.
2. Créez un fichier JavaScript nommé **`server.js`**.
3. Dans **`server.js`**, utilisez le module **`http`** intégré de Node.js pour créer un serveur HTTP. Vous devrez utiliser la méthode **`http.createServer()`** pour créer le serveur.
4. Configurez le serveur pour écouter sur un port de votre choix (par exemple, **`3000`**) et pour répondre aux requêtes HTTP
5. Lorsqu'une requête **`GET`** est reçue sur le chemin **`'/'`** (la racine), le serveur devrait renvoyer une réponse HTTP avec un statut **`200`** et un corps de réponse contenant un message de bienvenue, par exemple : **`"Bienvenue sur mon serveur web !"`**.
6. Pour toute autre requête, le serveur devrait renvoyer une réponse avec un statut **`404`** (Non trouvé) et un message indiquant que la ressource n'existe pas.
7. Testez votre serveur en exécutant **`node server.js`** et en accédant à http://localhost:3000 (ou le port que vous avez choisi) dans un navigateur.

Bonus :

8. Créez une page HTML simple et servez-la en réponse à la requête racine ('/'). Pour cela, utiliser le module FileSystem **`node:fs`**. Si la lecture du fichier échoue, le serveur doit renvoyer une réponse HTTP avec un code status 500.
9. Ajouter un formulaire à la page **`index.html`** qui permet de poster un message (**`<input type="text" name="message"/>`**) sur l'URL **`/messages`**. On rappelle qu'une requête **`POST`** est envoyée par défaut à la soumission du formulaire. Côté serveur, récupérer la requête **`POST`** avec le contenu du formulaire et retourner le message au client avec le message *"Voici ce que nous avons reçu : message="*
10. Soumettez le formulaire avec cURL, sans passer par le navigateur web

Exercice 6 : Serveur web avec Express

Pré-requis : installer express

1. **Reproduire** les fonctionnalités du site web développé à l'exercice 5 mais cette fois-ci avec le framework Express. Pour créer le projet Express, utiliser le générateur de projet express et la commande **`express`**. Tester de requêter l'URL **`/messages`** avec le verbe HTTP **`OPTIONS`** pour vérifier qu'il est bien implémenté. **Constatez** les différences entre les deux projets en terme de code et d'abstractions nouvelles mises à disposition par express.
2. Sur l'URL **`/increment`**, **implémentez** la méthode **`POST`** qui permet d'incrémenter un compteur de 1 à chaque requête, et retourne la valeur du compteur. Le compteur démarre à 0. Serait-il judicieux d'incrémenter ce compter de cette façon avec la méthode **`PUT`** ? Est ce que cela respecterait les conventions de l'interface uniforme (rôle, sûreté et idempotence des méthodes HTTP) ?
3. **Ajoutez** une ressource *liste des nombres pairs positifs* sur l'URL **`/even-numbers`**. Par défaut, la ressource retourne les 100 premiers nombres pairs au format **`application/json`**.
4. **Ajoutez** des paramètres d'URL (*query parameters*) pour réaliser les variations suivantes :
    - **`start-at=X`**, pour commencer à partir d'une valeur différente de 0. **`X`** doit être positif et paire
    - **`quantity=X`**, pour changer le nombre de nombres pairs à retourner. **`X`** doit être positif
    - **`order=desc|asc`** pour les retourner du plus grand au plus petit ou inversement (**`asc`** par défaut)

Vous devez valider les valeurs des paramètres d'URL et retourner une réponse avec le code status adéquat (**`400`**) si une valeur est invalide.

5. **Ajoutez** une ressource *Détail d'un·e utilisateur·ice* exposée sur l'URL **`/users/:pseudo`**, où **`:pseudo`** est la partie *dynamique* de l'URL. Cette route doit utiliser un middleware pour vérifier si l'utilisateur existe. Si l'utilisateur·ice existe, retourner les détails de son profil au format **`application/JSON`**, sinon une réponse HTTP avec le code status **`400`** avec le message "Cette ressource n'existe pas". Voici un jeu de données test simulant une base de données utilisateurs :

```
const users = [
   {pseudo : 'john', firstName : 'John', lastName : 'Doe', city :
        'Cincinnati'},
   {pseudo : 'jane', firstName : 'Jane', lastName : 'Doe', city :
        'Angers'},
 ]
```

D’après les règles de design des URL et de la hiérarchisation des ressources, que devrait exposer l’URL **`/users`** ?

6. Ajouter le Middleware suivant sur l’URL **`/users:pseudo`** :

```
router.use('/users/:pseudo', (req, res, next) => {
  console.log(`Réponse envoyée : ${res.statusCode}`);
  console.log(`Faire quelque-chose après l'envoi de la réponse`);
  next();
});
```

Comment faire pour exécuter cette fonction *après* le middleware de traitement de la requête ?

7. Installer **`swagger-autogen`** et **`swagger-ui`** pour générer automatiquement la documentation et la servir sur l’url **`/doc`** de votre application web.

Ressources utiles : Request (Object req), Response (Object res),

Exercice 7 : Choisir entre plusieurs représentations d’une ressource

Notre site web francophone expose des articles dans plusieurs langues : français, anglais et espagnol. Voici le résumé d’un article sur le recyclage dans les 3 langues mentionnées :

```
//fr
Le recyclage est une pratique essentielle pour préserver notre planète. Il
        permet de réduire les déchets, d'économiser des ressources naturelles
        et d'aider à lutter contre le changement climatique. En recyclant, nous
        contribuons à un avenir plus durable pour les générations futures.

//en
Recycling is an essential practice to preserve our planet. It helps reduce
        waste, conserve natural resources, and combat climate change. By
        recycling, we are contributing to a more sustainable future for
        generations to come.

//es
El reciclaje es una práctica esencial para preservar nuestro planeta. Ayuda a
        reducir los residuos, conservar los recursos naturales y combatir el
        cambio climático. Al reciclar, estamos contribuyendo a un futuro más
        sostenible para las generaciones venideras.
```

Nous aimerions exposer la ressource *Article sur le recyclage* dans les 3 langues, sur l’URL **`/articles/recyclage`**. Pour le faire, nous avons 3 options :

- Créer 3 URL différentes : **`/articles/recyclage/`** ou **`/articles/recyclage/fr`**, **`/articles/recyclage/en`**,**`/articles/recyclage/es`**

- Créer une URL canonique avec les headers Link et **`rel=[canonical] (https://developer.mozilla.org/fr/docs/Web/HTML/Attributes/rel#canonical)`** et ajouter un paramètre d'URL pour créer des variations du contenu : **`articles/recyclage?lang=es`**
- Créer une URL **`/articles/recyclage/`** et utiliser la la négociation de contenu avec les headers **`Content-Language`** et **`Accept-Language`**

1. Avec une application Node.js/Express, **implémenter** les 3 stratégies possibles.
2. Quels sont les avantages et inconvénients de chaque stratégie ?

> *Ces questions se posent également pour la négociation de contenu sur le format (HTML, PDF, XML, JSON, etc.)*

Exercices supplémentaires

- Exercices sur CodeWars, choisissez des katas adaptés à votre niveau (le but est d'apprendre et/ou de vous perfectionner en JavaScript). Vous devez vous créer un compte sur codewars
- Demandez-moi !