Université Paris Cité

Compte rendu

Développement Avancé : TP 2

Laurent Giustignano

Par Steven TEA 303

Étape 2

Avant de commencer le développement du serveur et la récupération de données depuis l'API de Marvel, nous commençons par tester comment les données de l'API sont récupérables avec les clés fournies. Avec Postman, nous effectuons un hachage MD5 avec la clé publique, privée et l'horodatage, ce hash est renvoyé à l'API pour nous identifier et s'il est correct, renvoyé la réponse de la requête que nous attendons. Pour ce faire, nous enregistrons dans la partie pre-request script pour écrire en JavaScript, des variables pour nos clés et le hash pour en faire des variables d'environnement dans Postman. Utiliser des variables d'environnement permet de simplifier le

Étape 3

Une fois testés, nous essayons de récupérer les données de l'API dans notre serveur NodeJS. J'ai eu des difficultés lors du développement de la méthode getData(), notamment sur la manière d'entrer les paramètres à envoyer pour que l'API valide notre identifiant. Je suis finalement arrivé en utilisant une instance de URLSearchParams, qui prend en entrée un objet JS avec l'ensemble de clé-valeur dont l'API a besoin. De plus, une fois que la requête a été effectuée, les données renvoyées ne sont pas automatiquement sous le format JSON, mais il faut le faire soit même avec la méthode .json() avant d'en extraire les données.

La fonction getData() peut être améliorée en termes de performance, il doit être possible de faire une unique boucle pour à la fois vérifier si le personnage a une image et s'il en a une, de lui affecter l'imageUrl avec l'extension.

Étape 4

Maintenant que nous pouvons nous connecter à l'API depuis notre application Node.js, nous créons un serveur avec des partials, des composants réutilisables dans d'autres fichiers Handlebars (un footer et un header dans notre cas). De plus, lorsque nous ouvrons la page 127.0.0.1:3000, la page index.hbs est affichée avec les données récupérées depuis l'API. Ces données sont envoyées à l'index.hbs qui, grâce à une boucle for, parcourt chaque élément pour les afficher.

D'un point de vue architectural, il est possible d'améliorer le serveur pour le rendre plus maintenable. Pour respecter l'architecture MVC, il serait nécessaire de mettre la

configuration de Fastify dans un fichier distinct du app.get(), correspondant à une méthode de contrôleur. Ainsi, il serait plus facile à maintenir grâce à un meilleur découplage des responsabilités.

Étape 5

Maintenant que notre programme est prêt, nous allons le mettre dans un conteneur avec Docker. Cela facilite le déploiement sans se soucier des dépendances. J'ai suivi les instructions et j'ai réussi à placer l'application dans un conteneur. Cependant, j'ai oublié de spécifier le port 3000 avec la commande EXPOSE dans le Dockerfile. Pour lancer l'application, nous utilisons la commande docker run -p 3000:3000 tp2. Cela lie le port de notre ordinateur au port du conteneur. Ensuite, nous pouvons accéder au serveur en ouvrant notre navigateur et en allant sur 127.0.0.1:3000.

Pour aller plus loin…

Séparation en deux affichages différents : vignettes et liste

J'ai également réalisé l'un des deux bonus du TP. J'ai ajouté deux nouveaux fichiers : card.hbs et li.hbs. L'extension Handlebars est nécessaire pour pouvoir transmettre des données entre la page index.hbs et les pages card et li, notamment le nom du personnage et l’URL de son image.

Pour faciliter la modification de notre page sans devoir intervenir dans le code, j'ai mis en place sur la page index.hbs un mécanisme permettant de modifier l'affichage grâce à deux boutons alternant entre la liste et les cartes. Lorsqu'un bouton est pressé, un paramètre, viewType, est ajouté à l'URL de la requête, et il est vérifié lors de l'affichage des données.