Rapport Miniprojet
Programmer le Cloud

Johan VERNE | CSN 26

4 novembre 2025

URL dépôt Github : https://github.com/JohanVerne/CloudComputing_TypeScriptApp
URL dépôt DockerHub : https://hub.docker.com/repository/docker/johanve/sysinfo-api/

1 Tâches séance 1

1.1 TD1 : une application Node.js

J'ai commencé par créé le dépôt GitHub du projet à partir du template pour typescript : https://github.com/khannurien/i-want-typescript. J'ai ensuite effectué les différentes étapes décrites dans son **README** afin d'exécuter le code example pour vérifier le bon clonage du projet. J'ai ainsi pu obtenir un dépôt de base sur lequel je vais pouvoir construire mon application : https://github.com/JohanVerne/CloudComputing_TypeScriptApp

Je me suis ensuite intéressé aux fichiers **package.json** et **package-lock.json** créés par Node.js. **package.json** définit la configuration principale du projet et ses dépendances ; il décrit des informations générales sur le projet, comme le nom de l'auteur ou la licence utilisée, les scripts utiles pour les raccourcis de commande de projet, comme "start" pour exécuter le code compilé le projet ou "watch" pour lancer nodemon ou encore les différentes dépendances à installer pour exécuter le projet . De son côté, **package-lock.json** est généré automatiquement par `npm` et enregistre un arbre complet des dépendances utilisées dans le projet avec les différentes versions utilisées, afin d'assurer la reproductibilité du projet par d'autres développeurs.

Ensuite, j'ai installé le package `systeminformation` avec la commande **npm install systeminformation** afin d'obtenir des informations sur le système exécutant le code du serveur et les afficher sur la page web. Dans le fichier **package.json**, le champ `dependencies` indique les dépendances nécessaires à l'exécution du programme, alors que le champ `devDependencies` montre les outils de développement, utiles seulement pour coder ou compiler le projet.

Je me suis ensuite penché sur l'écriture de code simple pour le fonctionnement du serveur. Dans mon serveur, j'utilise le framework **Express** pour écouter sur le port 8000 et répondre aux requêtes HTTP. Lorsque quelqu'un accède à lURL `http://localhost:8000/api/v1/sysinfo`, j'appelle la fonction asynchrone getSystemInformation(), qui récupère à travers le module systeminformation diverses données sur la machine, comme le processeur, le système d'exploitation, la mémoire, le taux de charge du CPU, les processus en cours, les disques et les interfaces réseau. Ces informations sont ensuite renvoyées au client sous forme d'un objet JSON conforme à l'interface ISystemInformation. Si une erreur se produit lors de la récupération des données, j'envoie une réponse 500 avec un message d'erreur. Pour toutes les autres routes qui ne sont pas définies, je retourne une réponse 404 "Not Found", indiquant que la page demandée n'existe pas.

J'ai enfin pu tester le fonctionnement du serveur en lançant `npm run build && npm run start` ou `npm run watch`, et en me connectant aux URLs `http://localhost:8000` et `http://localhost:8000/api/v1/sysinfo`.

On utilise ce formalisme dURL pour lAPI, car ce découpage clair rend lAPI prévisible, stable et facile à étendre : chaque partie de lURL a un rôle précis (serveur, version, ressource), conforme à la logique REST où chaque ressource est identifiée par une URI unique.

2 Tâches séance 2

2.1 Fin TD1

Afin de tester notre application, on installe la dépendance Jest avec :

```
npm install --save-dev jest @types/jest ts-jest typescript
```

J'ai utilisé les Jest mock afin de simuler les objets Request et Response d'Express. Les réponses des mocks renvoient des méthodes nous permettant de tester les routes sans faire tourner le serveur HTTP.

On écrit ce jeu de tests afin de détecter les erreurs dès l'étape de développement du serveur avant même qu'il ne soit déployé, et ainsi pouvoir garantir son comportement.

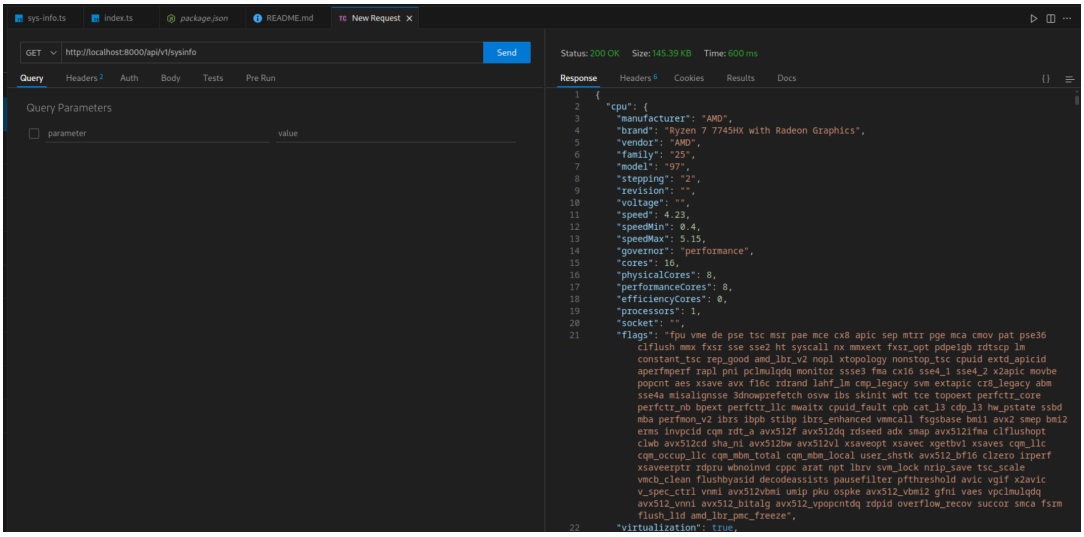

FIGURE 1 – Connection à l'API réussie

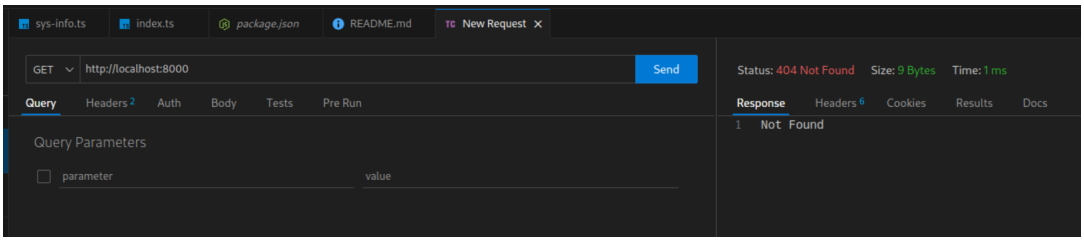

FIGURE 2 – Connection à l'API échouée

2.2 TD2 : conteneurisation avec Docker

Avec un Docker installé, nous allons pouvoir conteneuriser notre serveur hors du reste de la machine. pour cela, il nous faut créer un Dockerfile contenant les instruction d'exécution de notre serveur. Dans ce Dockerfile, on installe les paquets systèmes utiles à la compilation de notre serveur (nodejs, npm), on installe les dépendances du projet avec `npm ci` et on build le serveru avec `npm run build`. On peut enfin run le serveur après avoir exposé le port utilisé. On peut donc créer l'image relative au Dockerfile avec la commande

```
sudo docker build . -t sysinfo-api:0.0.1
```

On crée enfin le conteneur à partir de l'image, avec la commande

```
sudo docker run -p 8123:8000 -m1024m --cpus=1 sysinfo-api:0.0.1
```

L'option -p permet d'effectuer un mapping entre le port hôte (8123) et le port du conteneur (8000). Ainsi, toutes les requêtes effectuées vers le port 8123 de la machine hôte seront redirigées vers le port 8000 du conteneur.

L'option -m limite la mémoire RAM disponible pour le conteneur à 1014 Mo.

Le flag –cpus défini le nombre de coeurs CPU utilisables par le conteneur (ici 1).

Faire varier ces option aura donc un impact important sur la vitesse d'exécution de notre serveur. On pourrait lui allouer moins de mémoire RAM ou plus de couers CPU pour altérer sa vitesse d'exécution des instructions.

On peut ensuite inspecter notre image avec

```
sudo docker image history sysinfo-api:0.0.1
```

Cette commande permet d'inspecter les instruction exécutées par le Dockerfile et leur impact sur la taille totale de l'image.

On peut aussi inspecter avec la commande

```
dive sysinfo-api:0.0.1
```

Celle-ci permet d'accèder à plus d'information qu'avec `history`, comme les fichiers modifiés à chaque couche.

On remarque que l'image est quand même assez volumineuse (250 Mo). On pourrait réduire la taille de l'image en séparant la phase de build de la phase d'exécution, afin que seuls les fichiers JS compilés et les dépendances de compilation soient présentent dans l'image finale.

En rajoutant une construction multistage dans notre Dockerfile (en précisant la partie builder et la partie runner), on obtient une taille d'image de 68Mo. Cela permet ici de diviser la taille d'image par 3 par rapport à une construction à simple étage. Dans un contexte réel, cela pourrait avoir un impact important sur le temps de déploiement de déploiement (images + légères –> téléchargement et démarrage + rapide) et permet de réduire les coûts d'infrastructures (moins d'espace de stockage nécessaire).

Une fois notre image crée, on peut la publier sur Docker avec les commandes

```
sudo docker tag sysinfo-api:0.0.2 johanve/sysinfo-api:0.0.2
sudo docker push johanve/sysinfo-api:0.0.2
```

L'image est ensuite présente sur Docker Hub. On peut créer un nouveau conteneur à partir de cette image avec la commande

```
sudo docker run -p 8123:8000 -m1024m --cpus=1 johanve/sysinfo-api:0.0.2
```

2.3 TD3 : CI/CD avec GitHub

Afin de créer un workflow pour notre projet, on peur créer un fichier .yml dans un dossier particulier du projet afin que ceux-ci puissent être exécutes par Github après certaines actions de version control de Git.

J'ai commencé par utilisé les templates de Github, décrites sur le site https://docs.github.com/en/actions/get-started/quickstart, afin de créer un workflow de base qui exécute quelques affichages et un `ls` du projet après un `push`.

J'ai ensuite créé un workflow de CI pour installer tester rapidement l'application, en installant Node et les dépendances de compilation, puis en buildant l'app et en effectuant les tests créés dans le TD3. Dès qu'on effectue un push ou un pull du repo, le workflow s'exécute automatiquement dans Github et s'exécute correctement (représenté par un checkmark sur la page du repo, ou dans la section "Actions"), on peut également observer le détail de l'exécution.

Cela permet d'effectuer automatiquement les tests de bon fonctionnement de notre application au push et assurer l'utilisabilité de l'application durant diverses phases de conception / développement.

J'ai également créé un workflow de CD, afin de build automatiquement l'image Docker de l'app au push et de l'uploader dans le Docker Hub. Pour cela, j'ai ajouté des secrets au repo Github, correspondant à mes identifiants Docker, afin d'automatiser l'upload de l'image. Une fois le push effectué et le workflow d'intégration réussi, le workflow de livraison se connecte automatiquement à mon compte Docker et upload le'app buildée précédemment sur le Hub. Au succès du workflow, on peut observer que l'image a bien été mise à jour sur Docker Hub.

3 Tâches séance 3

3.1 TD4 : Déploiement continu sur PaaS

On peut maintenant créer un PaaS Azure afin de hoster notre webserver. Pour cela, on créer une instance de conteneurs sur le site d'Azure, on défini l'image Docker créée précédemment et le port utilisé pour communiquer avec l'app, aussi défini précédemment. On peut maintenant accéder à notre app web, en inspectant l'overview de notre instance et en récupérant l'adresse IP du serveur.

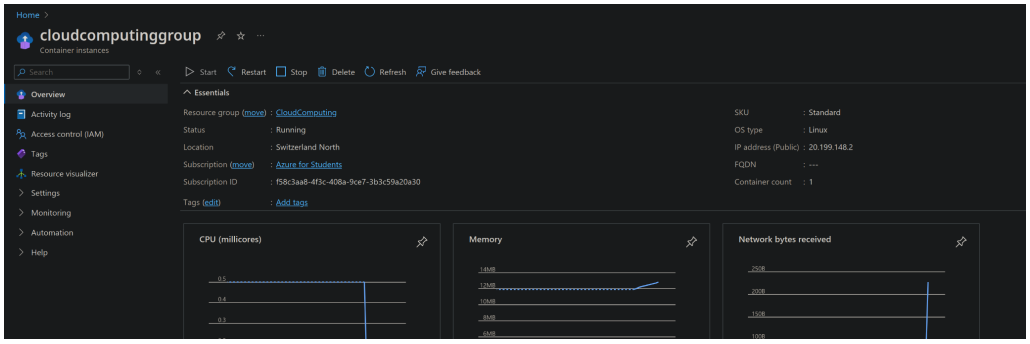

FIGURE 3 – Overview de notre webapp hostées sur Azure

On peut ainsi se connecter à notre webapp avec l'adresse :

```
http://20.199.148.2:8000/api/v1/sysinfo
```

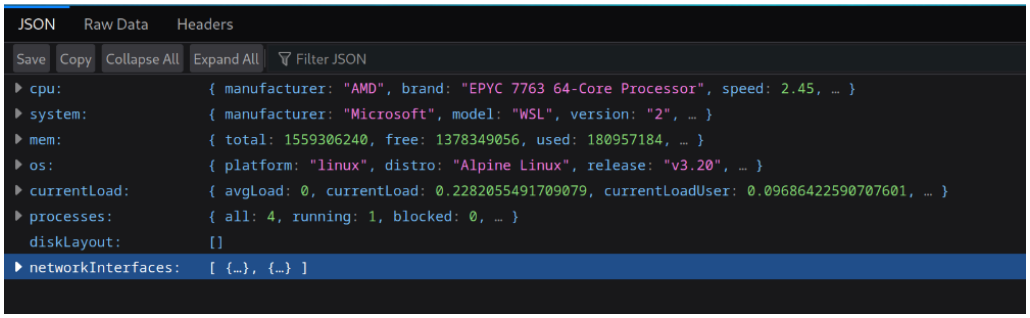

FIGURE 4 – Résultat de l'appel à l'API hosté par Azure

On peut ainsi observer les divers caractéristiques du conteneur qui host le serveur :

— Le CPU est un AMD EPYC 7763 64-Core Processor, mais indiquée avec un seul cur visible (cores : 1) dans le conteneur (défini par nous à la création du conteneur.
— Le système d'exploitation est Alpine Linux version 3.20 avec un noyau Linux 6.1.124.1 qui tourne sous WSL.
— La mémoire disponible est environ 1,5 Go (défini à la création du conteneur) avec une grande partie libre.
— La charge CPU est très faible (0.12%) et plusieurs processus système sont listés, notamment "pause", "tail", "node".

On aurait obtenu un résultat différent si le serveur était hosté sur une machine virtuelle plutôt que sur un conteneur :

Une machine virtuelle fait tourner un système d'exploitation complet avec son propre noyau, ce qui demande plus de ressources et une isolation plus forte. La sortie observée dans un conteneur serait différente d'une machine virtuelle, car, dans une VM, on aurait accès aux informations complètes du système invité (plus de ressources visibles, différents noyaux et versions, plusieurs curs CPU).

Afin de permettre le Continuous Deployment de notre app, pour automatiquement modifier le conteneur Azure à la modification de notre GIT, nous allons créer une WebApp sur Azure qui nous permettra d'avoir une site web pour notre app. Dans la WebApp crée, on peut activer la checkbox "Continuous deployment for the main container", dans le menu Deployment Center. Cette option ajoute un WebHook vers notre dépôt Docker. Dès que notre image Docker est mise à jour avec un `docker push`, un message est envoyé au WebHook qui redémarre la webapp avec l'image mise à jour. il faut également cocher l'option "Basic Auth Publishing Credentials" dans les réglages de notre webapp Azure pour assurer le bon fonctionnement du webhook.

Il nous suffit enfin de sélectionner notre image dans Docker Hub et, dans la section Webhooks, créer un nouveau webhook avec l'adresse créée par Azure. Désormais, à la modification et au psuh de la webapp sur Github, uen image Docker est automatiquement créée et publiée sur Docker Hub qui, à son tour, informe le Azure webapp de la modification de l'image pour qu'il modifie le site hosté.

L'URL final pour accèder à la webapp est la suivante :
https://cloudcomputingwebapp-fac7a2fvapbxhsfx.switzerlandnorth-01.azurewebsites.net/api/v1/sysinfo