Challenge Stack MicroServices

La stack technique que l'on a choisit d'implémenté est sous la forme d'une architecture micro service dans un conteneur Docker. Le but de cette stack est de réalisé une sorte de Benchmark ainsi qu'un test de technologie relativement récente pour voir leurs potentiel.

Stack Prévisionnelle

Voici la stack que l'on essaiera d'implémenter :

Front-End : Simple page client avec le framework Svelte pour réaliser les appels de "Benchmark" sur les différentes back-end

Les échanges seraient en **Grpc** et pourquoi pas comparer avec REST

Back-End :

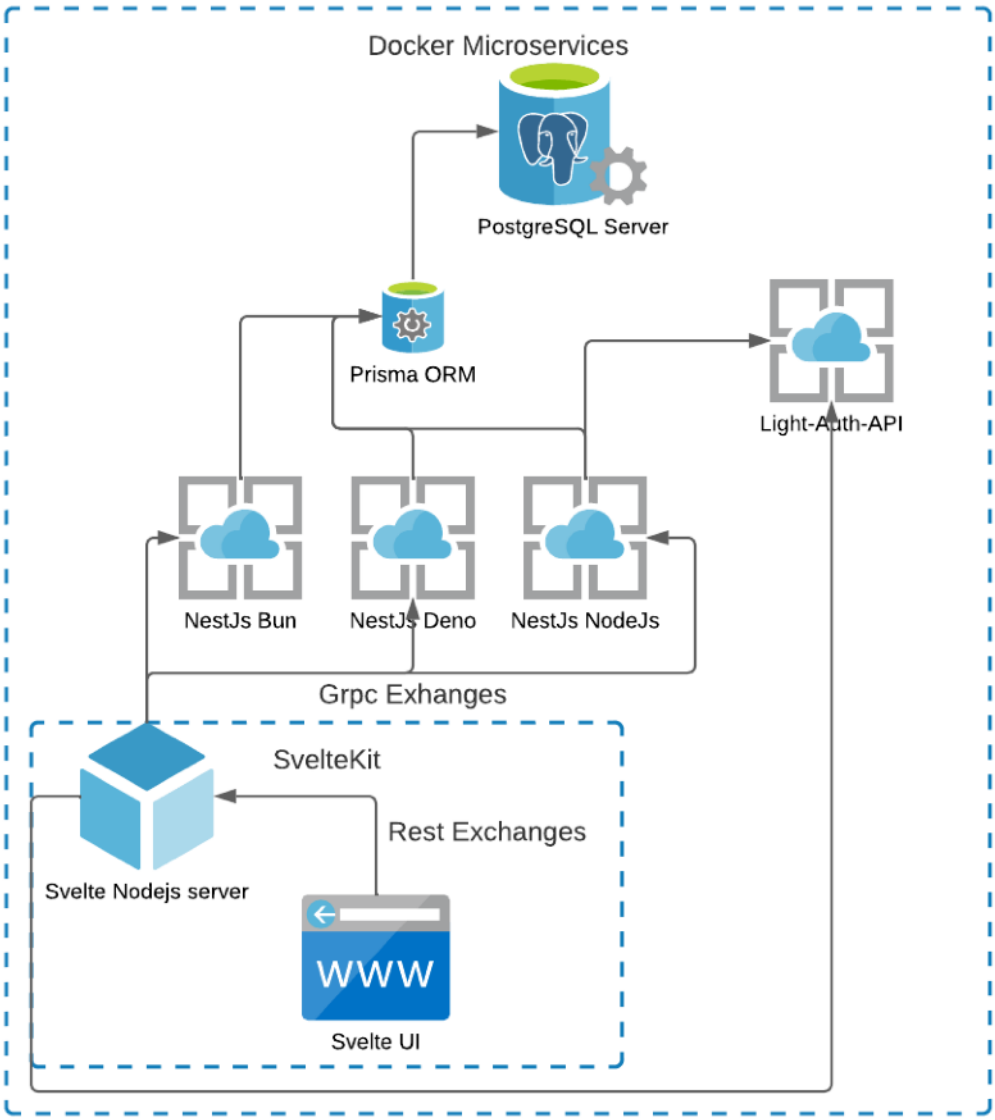
- NestJs sur Bun
- NestJs sur Deno
- NestJs sur NodeJs

Base de donnée : Postgres dans un conteneur docker avec l'Orm Prisma depuis les backend NestJs

Rapport Microservices :

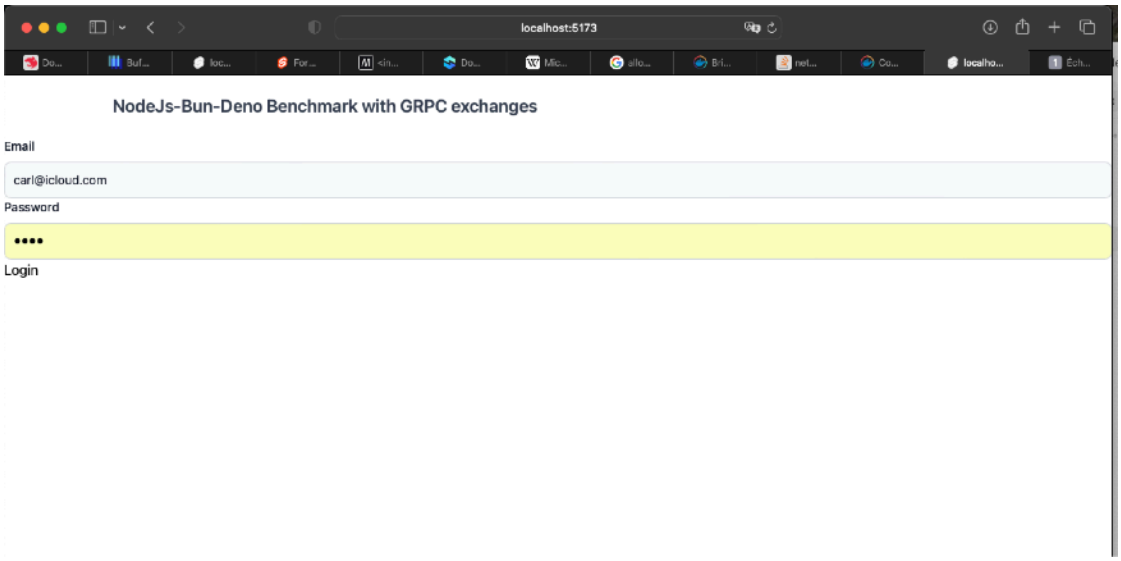
[Update] Ajout d'un micro service d'authentification «sécurisant » l'appel de Benchmarking

Schéma d’architecture

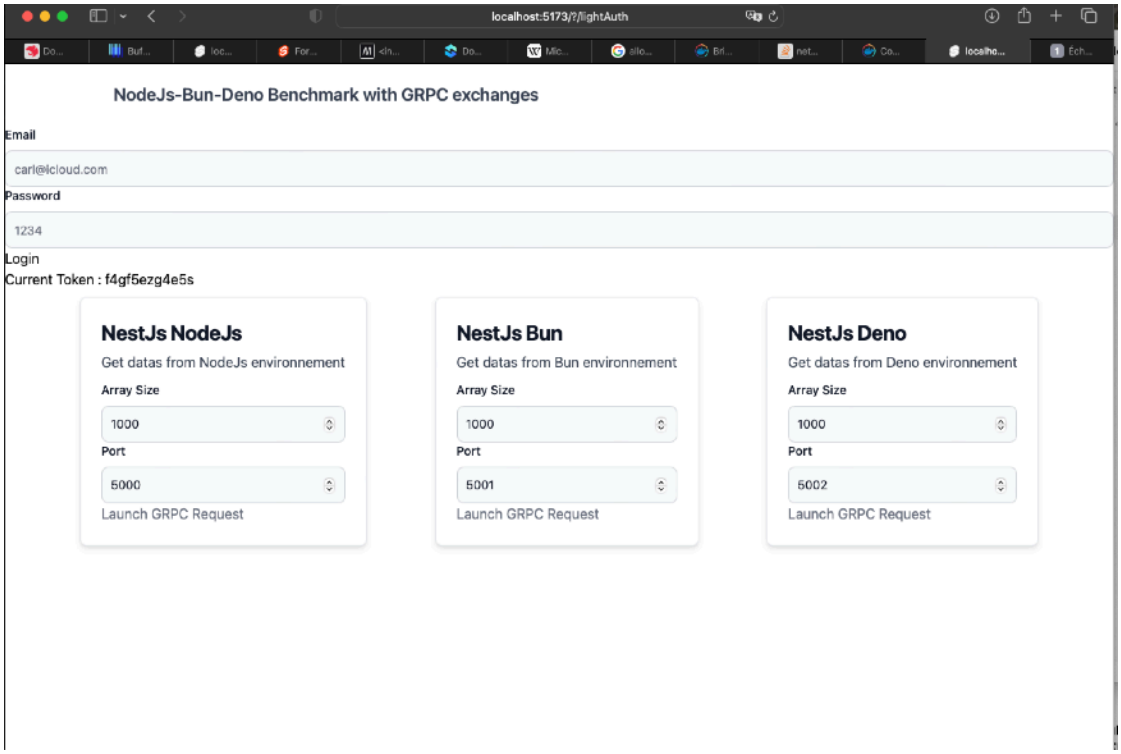


Descriptions des microservices crées :

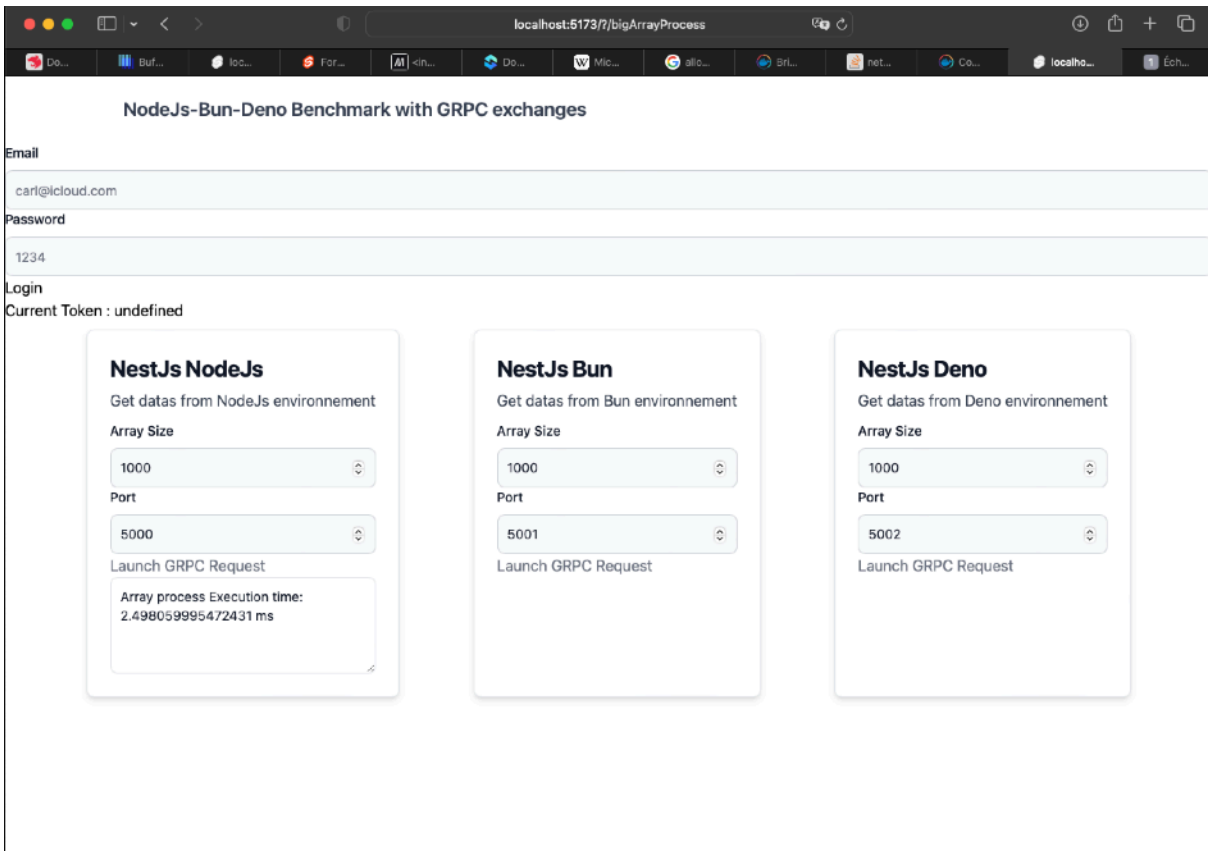
Client Front :



Récupération d’un Token (faux) depuis ma light-auth-api en GRPC (Login)

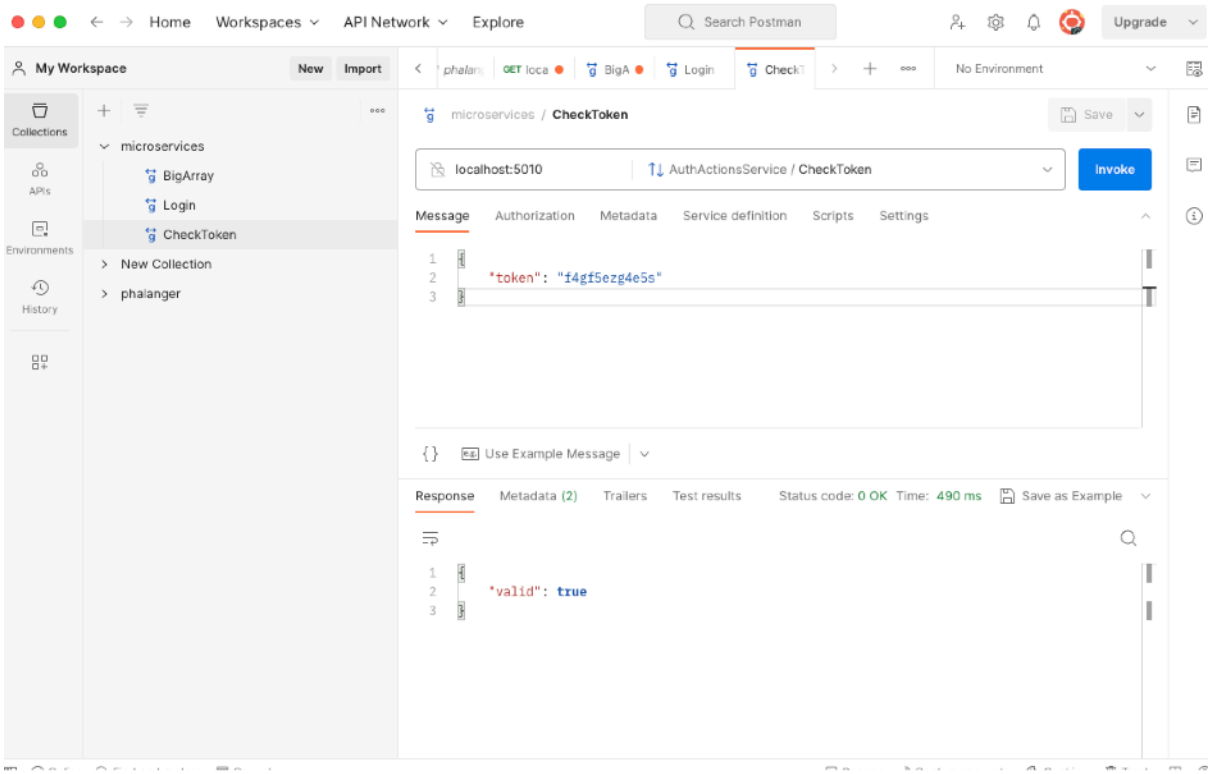


Envoi de la requête de Benchmarking avec le Token reçu précédemment



L’api nest-js-base effectuant le benchmarking vérifie le token avec la route CheckToken appelant elle aussi la light-auth-api coté serveur cette fois

Test et développement avec Postman



Dockerisation

J’ai passé beaucoup de temps a essayer de dockeriser toute la stack, malheureusement je bloque au niveau de la communication des containers entre-eux, je peux appeler par exemple le service de Login tournant sur Docker depuis postman mais par contre le front et le back-end n’y arrive pas je n’arrive pas à « lier leur réseau »

Explication du « benchmarking » metier

Le principale but de cette stack est de mesurer et de constater les différences de performances entre des runtimes coté serveur de Javascript.

Le plus connu étant bien sûr NodeJs, on va essayer de le comparer à deux solutions alternative se présentant comme plus performantes :

- Bun (3x)
- Deno

Le client Svelte sera capable d'appeler un même back-end NestJs déployé sous forme de 3 instances sur chaque runtime javascript.

Au niveau des métriques important à analyser

- Rapidité d'installation
- Taille des bundles
- Taille du microservices
- Simplicité de configuration
- Vitesse d' exécution

Pour mesurer la vitesse d'exécution j'ai choisis d'implémenter deux fonction appelés en GRPC, une récupérant beaucoup d'élément en base de données en utilisant l'ORM Prisma et une autre effectuant des opérations sur des tableaux.

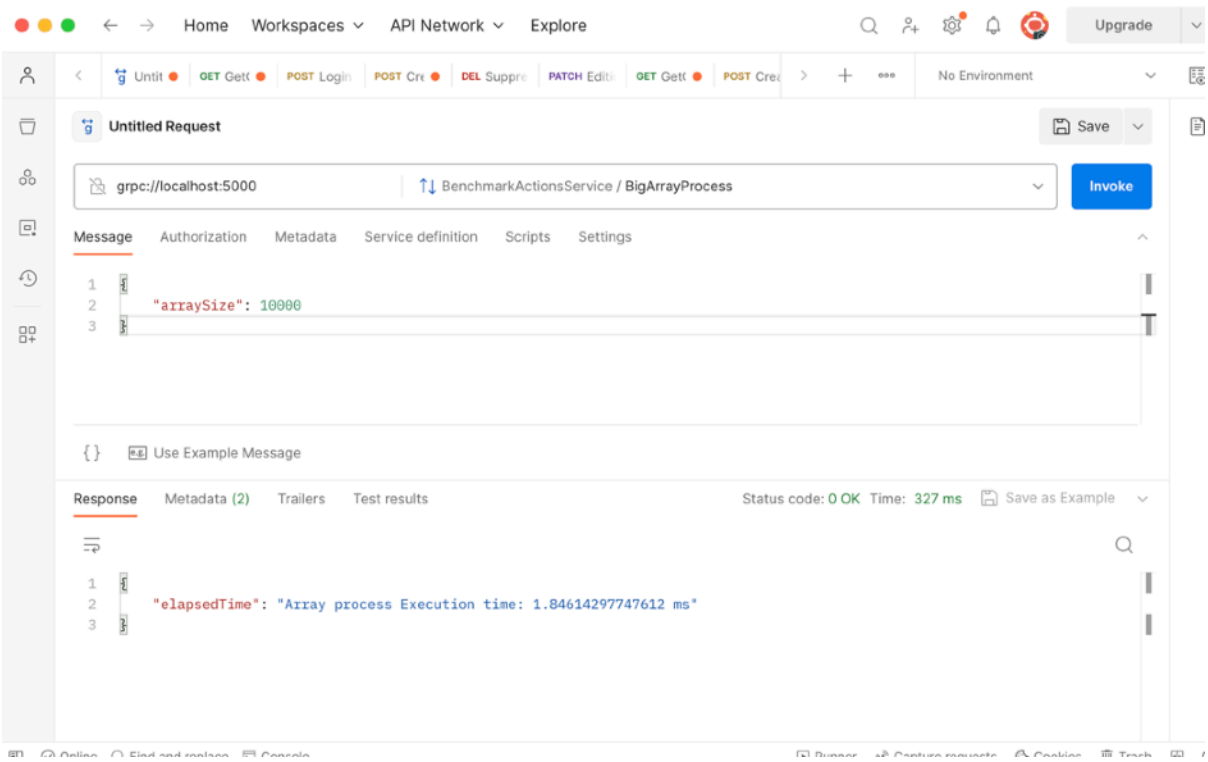
Rapport pour le challenge Stack (obsolète) :

Test effectuées

Après implémentation de l'API Nest-JS, premier test d'exécution avec Postman en GRPC :

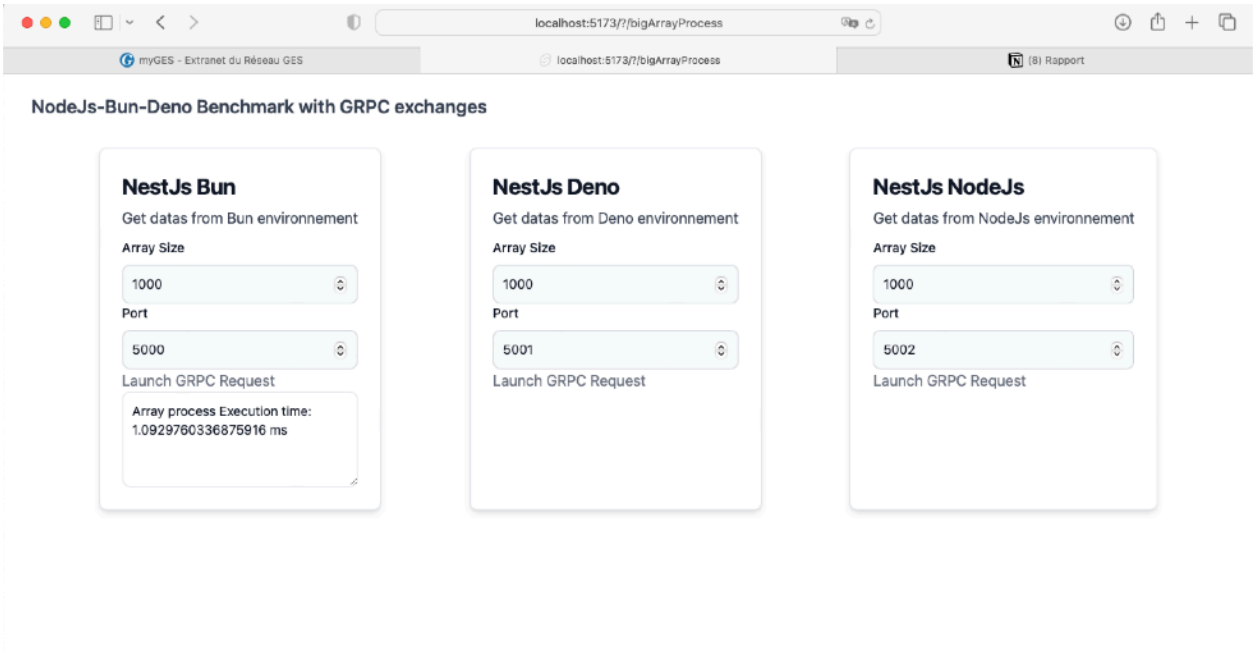
Pour arriver à cela il a fallu :

- *Créer des endpoints sur NestJs*
- *Créer le fichier Protobuf définissant le contrat GRPC*
- *Générer les stubs pour le client et le serveur à partir du protobuf*
- *Implémenter le fonctionnement GRPC de NestJS*



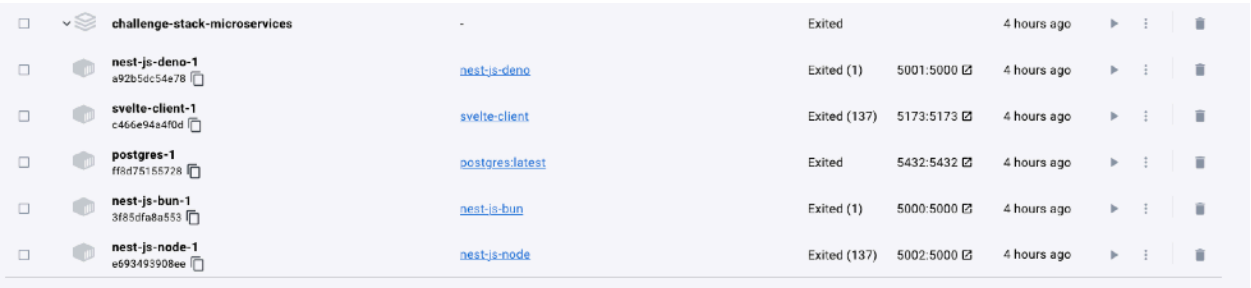
On a maintenant le projet de base pour effectuer nos benchmarks on passe maintenant à la création du client qui va ci connecter pour l'utiliser

Création d'un projet avec Svelte Kit, les appels GRPC sont effectué coté serveur svelte, le client svelte et le serveur svelte communique en rest, le principal avantage de cette méthode et que le ***front peut fonctionner sans Javascript***



Le serveur svelte communique grâce au stubs généré depuis le protobuf et le module GrpcTransport développé par la communauté les utilisants.

Dockerisation pour les benchmarks de comparaison



Toutes la stack a été déployé dans docker pour faciliter le processus de Benchmarking et la duplication sous 3 instance du projet NestJs.

Analyse des métriques et Conclusion

Le seul métrique pertinent que j'ai réussi à faire fonctionner dans le temps impartit est le build des packages npm en comparaissant entre nodeJs et Bun,

Dans ce cas d'utilisation on voit que le temps de build est 3 fois plus important sur nodejs ce qui appuie les promesses de Bun. La comparaison avec Deno n'a pas été faite.

```
=> [nest-js-node 4/5] RUN npm install --force                    42.8s
=> [nest-js-bun 4/4] RUN bun install                             15.4s
```

Au niveau de la taille des images entre nest-js sur bun et nodeJs on constate 1.4gb sur NodeJs et 500mb pour Bun.

Cependant ces informations sont à relativiser puisqu'on arrive au problème rencontré qui m'a empêché de réaliser tout mes tests : la compatibilité avec nodejs.

Bun ce veut comme un remplaçant plus efficace de NodeJs mais toutes les fonctionnalités ne sont pas implémentés , par exemple les fonctionnalités assez avancé http2 utilisé dans mon projet utilisant GRPC ne sont pas encore compatible. Je n'ai donc pas pu démarré le service dans docker.

```
13 |
14 |     super(feature + " is not yet implemented in Bun." + (issue ? " Track the status & thumbs up
the issue: https://github.com/oven-sh/bun/issues/" + issue : ""));
       ^
NotImplementedError: node:http2 createServer is not yet implemented in Bun. Track the status & thumbs
 up the issue: https://github.com/oven-sh/bun/issues/887
 code: "ERR_NOT_IMPLEMENTED"
```

L'alias nest dans le script a retardé ma découverte du problème puisque la commande bun run start fonctionnait en local sur mon pc mais pas dans le conteneur docker. Cela s'explique car l'alias nest utilise nodejs et non bun.

```
"scripts": {
  "build": "nest build",
  "format": "prettier --write \"src/**/*.ts\" \"test/**/*.ts\"",
  "start": "nest start",
  "start:dev": "nest start --watch",
  "start:debug": "nest start --debug --watch",
  "start:prod": "node dist/main",
```

En conclusion voici la limite de par exemple ce nouveau runtime bun qui est bloquant dans ce cas d'utilisation (à cause du GRPC).

Il aurait aussi été intéressant de comparer les vitesses d'échanges entre GRPC et Rest, GRPC est censé être plus rapide mais pose des problèmes de compatibilité dans notre stack puisque moins répandu.

Pour finir on voit qu'en choisissant des technologies moins répandus et plus nouvelles on peut être confronté à des problèmes de compatibilité.

Pour choisir sa stack il faut donc peser le gain apporté, dans notre cas on cherchait la performance, par rapport à la complexité de mise en place et les problème rencontrés (de compatibilité chez nous).

Lien du projet GitHub : https://github.com/Carlitos73490/challenge-stack-microservices