Request Scheduler / Orchestrateur de Requete HTTP

Pour Canal+

Remarques Generales

Je tiens tout d'abord à vous remercier pour le temps que vous avez pris pour tout mon processus, et merci pour le temps que vous avez pris pour review mon code. J'ai essayé de le rendre le plus lisible possible, même si je ne suis pas pris, je serais très interessé par tout feedback possible !

Le test est un des tests les plus compliqués que j'ai du à faire, ce qui me rassure sur le fait que j'apprendrais plein de chose à Canal+. Mais il m'a permis d'explorer plusieurs notions de `javascript` qui ne sont pas si évidentes. La chose la plus compliquée que j'ai confronté dans ce sujet était de designer l'API et surtout de trouver une manière de remonter la valeur de la résolution de la requête là ou elle est demandée.

Stratégies explorées et temps utilisé

Ma première stratégie (3 heures) était de faire une queue, et d'essayer de planifier les bons appels au bon moment avec `setTimeout` et chain des Promise avec une fonction récursive , sauf que je n'avais aucun moyen de remonter la valeur de la résolution, ce qui m'a poussé à faire un générateur, et `yield` les valeurs (j'ai d'ailleurs découvert qu'un des polyfill de `async/await` était bâti avec `yield` et des `Promise`) et je m'étais retrouvé avec un code qui marche, mais qui avait une API presque inutilisable (obligé d'utiliser `yield`) et le code ne me plaisait personnellement pas.

Ma deuxième stratégie (1 heures) était de refaire la queue en utilisant un `Binary MaxHeap` pour la gestion de priorité, vu que techniquement plus performant sur la papier, mais notre cas n'a que peu de d'intérêt vu qu'on a qu'un nombre fini et discret de priorités, donc on se retrouvait avec un arbre où les nodes étaient quasiment tous les mêmes, l'intérêt du MaxHeap était presque perdu. Et aussi je trouvais l'implémentation compliquée personnellement.

Ma dernière stratégie (2 heures) , je suis revenu sur mon idée de base , une `queue` simple, et c'est même pas totalement vraie, c'est juste une liste normale, et à l'addition d'une requête, je rajoute le `resolve` et le `reject` à l'objet, ce qui me permettait de `resolve` à tout moment et de renvoyer la valeur là où il le fallait. Puis je cherche les prochaines requêtes à exécuter selon la logique de la priorité fournie dans l'exercice. J'utilise `Promise.allSettled` pour l'exécution parallèle des requêtes au lieu de `Promise.all` pour ne pas reject dès qu'une promise fail, mais je ne l'utilise pas pour renvoyer dans l'api `addBatch` car je ne veux pas bloquer l'exécution à l'ajout, car on peut ajouter des requêtes avec une priorité très haute comme très basse, donc ça serait totalement contre l'esprit du scheduler, vu que si je fait `scheduler.addBatch(VERY_HIGH,VERY_LOW)` on attendrait l'execution du `VERY_LOW` dans tout les cas, donc je renvoie juste une array de `Promise`

Améliorations

- J'aurais fait le deuxième bonus clairement, le comportement est un peu compliqué surtout quand utilisé avec le `changePriority` , si on change la priorité d'une requête en `VERY_HIGH` qui était déjà lancée , puis une déjà lancée en `VERY_LOW` , ce qui est attendu c'est que la deuxième soit immédiatement cancel, ce qui est pas forcement facile , mais j'aurais pu finir ça
- J'aurais clairement utilisé une max heap, c'est plus optimisé au final, et la rapidité dans ce genre de bibliothèque est cruciale. J'aurais pu rajouté plein d'optimisation
- Ajouter une manière pour rajouter des params aux requetes GET (voir les changer si la requête est encore waitlisté
- Ajouter un `re-queue` automatique si on fail une requête mais avec une priorité plus basse
- Voir comment ça s'intègre avec un cache, si on a des réponses déjà cached, est-ce que c'est plus judicieux de fetch le cache directement sans attendre (donc augmente la priorité artificiellement) ou quand même attendre ?
- Si ce code run dans un backend, ajoute une manière de `sync` la queue avec un `Message Broker` de type `Redis ou RabbitMQ` pour garder de la persistence, envoyer des notifications a la resolution des promises.

- Le code n'est pas très scalable (enfin pas scalable de manière horizontal, mais il est relativement scalable de manière verticale) donc je chercherai une manière de distribuer le code et de le gérer sur plusieurs worker

- Actuellement, `cancel` et `getStatus` passe par une url, mais si jamais on avait une `API GraphQL` a fetch, l'url ne changerait pas, donc ca serait compliqué de faire ça, je trouve qu'une bonne manière serait de générer un `id` unique pour chaque requête

Documentation de l'API

La librairie propose 2 classe à utiliser , `HTTPRequest` et `TaskScheduler` ainsi que `TaskPriority` un enum de priority

General

- `TaskPriority: VERY_HIGH | HIGH | NORMAL | LOW | VERY_LOW` : les différentes priorités possibles pour une requête

- `TaskStatus: WAITLIST | RUNNING | DONE | CANCELED` : les différents états possibles pour une requête

HTTPRequest

- `constructor<T>: (url: string, priority: TaskPriority) => HTTPRequest<T>` : construit une Requete qui peut etre run à tout moment, dont la réponse est un objet de Type `T`
- `run: () => Promise<T>` : run la HTTPRequest avec fetch et renvoie une Promise de la reponse en JSON
- `cancel: () => void` : abort la requete

TaskScheduler

- `constructor: () => void` construit un scheduler et init sa queue
- `add: (request: HTTPRequest<T>) => Promise<T>` : ajoute une requête au scheduler et renvoie une Promise qui resolve avec le résultat de la requete (ou reject si la requête ne réussit pas)
- `addBatch: (...requests: HTTPRequests[]) => Promise[]` : ajoute plusieurs

requêtes d'un coup au scheduler et renvoie une liste de Promise , chacune resolvera quand elle sera fini selon les règles de priorités

- `getStatus: (url: string) => TaskStatus` : renvoie l'état `RUNNING/WAITLIST` d'une requête, la recherche se fait pour le moment par l'url
- `cancel: (url: string) => void` : abort une requete par son url
- `peek: () => {url: string, status: TaskStatus}[]` : envoie l'état de toutes les requêtes, identifiées par leurs url
- `changePriority: (url: string, priority: TaskPriority) => void` : change la priorité d'une requete, identifiée par son url

PS

J'ai malheureusement dû utilisé , malgré l'interdiction de l'exercice, une librairie externe qui est `node-fetch` car l' `API Fetch` n'est pas implementée sur `NodeJS`
Ainsi que `eslint` pour valider mon code