CacheLab

Cahier des charges techniques

Yanis Bennadji

1. Choix Technologiques

Conformément aux exigences du projet, le choix s'est porté sur **Node.js avec TypeScript**.

1.1 Justification du Langage : Node.js & TypeScript

- **Performance des Entrées/Sorties (I/O) :** Node.js est conçu sur une architecture non-bloquante (Event Loop), idéale pour une API qui doit gérer un grand nombre de requêtes HTTP simultanées.

- **Sécurité et Robustesse (TypeScript) :** L'utilisation de TypeScript ajoute un typage statique fort. Dans le cadre de ce projet où nous manipulons des structures de données complexes (listes chaînées, références d'objets), le typage garantit l'intégrité de la mémoire et réduit drastiquement les bugs d'exécution.

- **Écosystème :** L'accès à des outils robustes comme Express pour le serveur et les outils de test facilite le développement rapide du MVP.

2. Architecture Applicative

L'application suit une architecture modulaire en couches pour garantir la maintenabilité et la séparation des responsabilités.

2.1 Schéma de l'Architecture

I. **Couche Interface (API REST) :** Reçoit les requêtes HTTP, valide les entrées et formate les réponses JSON.

II. **Couche Logique (Controller/Service) :** Orchestre les appels et gère les règles métier.

III. **Couche Données (In-Memory Engine) :** Le module CacheStore qui contient l'algorithme de stockage pur.

2.2 Composants du système

- **Serveur API :** Framework Express.js (léger et performant).

- **Module de stockage :** Classe CacheStore (Singleton) persistant les données en RAM tant que le processus est vivant.

- **Système de Logging :** Middleware personnalisé interceptant chaque requête pour logger la méthode, l'URL, le code retour et le temps de réponse (latence).
- **Documentation :** Intégration de Swagger UI (/api-docs) pour une documentation interactive.

3. Algorithmique et Structure de Données

C'est le cœur du projet. L'objectif est de garantir une complexité temporelle optimale.

3.1 Structure retenue : HashMap (Table de Hachage)

Nous avons choisi d'implémenter une **HashMap**.

Justification vs Recherche Dichotomique :

- **HashMap :** Offre une complexité moyenne de **O(1)** pour les lectures ET les écritures. C'est impératif pour un cache haute performance.
- **Recherche Dichotomique :** Bien que la lecture soit rapide (O(logn)), l'écriture nécessite de décaler les éléments du tableau (O(n)), ce qui est inacceptable pour un système de cache dynamique recevant beaucoup d'écritures.

3.2 Détails de l'Implémentation "Low-Level"

Pour maximiser la performance et la maîtrise mémoire (et respecter les contraintes strictes du projet), nous n'utilisons **aucune structure de haut niveau** (Map, Set) ni de méthodes itératives (map, filter).

- **Buckets :** Un tableau (Array) de taille fixe.
- **Fonction de Hachage :** Algorithme transformant la clé (string) en index numérique via la somme des codes ASCII modulo la taille du tableau.
- **Gestion des Collisions (Chaining) :** Utilisation de **Listes Chaînées**. Chaque case du tableau contient une référence vers un objet Entry ({ key, value, next }). En cas de collision (deux clés menant au même index), le nouvel élément est chaîné au précédent via le pointeur next.

4. Spécifications de l'API (Endpoints)

L'API respecte les standards REST et le format JSON. Les endpoints suivants sont implémentés:

Méthode	Endpoint	Description	Payload (Corps)	Réponse Succès
POST	`/keys`	Créer une clé/valeur	`{ "key": "...", "value": ... }`	`201 Created`
GET	`/keys/:key`	Lire une valeur	-	`200 OK` + JSON
PUT	`/keys/:key`	Modifier une valeur	`{ "value": ... }`	`200 OK`
DELETE	`/keys/:key`	Supprimer une clé	-	`200 OK`
GET	`/keys`	Lister tout (Admin)	-	`200 OK` + Array

5. Sécurité et Qualité

5.1 Mesures de Sécurité

- **Validation des entrées :** Vérification systématique de la présence des champs key et value avant tout traitement. Rejet des requêtes malformées (400 Bad Request).
- **Gestion des erreurs :** Les erreurs internes (500) sont capturées pour ne jamais exposer la stack trace ou la structure interne du code au client.
- **Mémoire Volatile :** Les données étant stockées en RAM, aucune donnée sensible n'est écrite sur le disque dur (protection contre le vol de disque), mais elles sont perdues au redémarrage (comportement attendu d'un cache).

5.2 Éco-conception

- **Efficacité Algorithmique :** L'algorithme en O(1) minimise les cycles CPU nécessaires par requête, réduisant la consommation électrique du processeur.
- **Gestion Mémoire :** L'utilisation de listes chaînées pour les collisions évite de devoir réallouer et copier des grands tableaux contigus en mémoire.

6. Risques et Améliorations Futures

- **Risque identifié :** Saturation de la mémoire RAM si le nombre de clés explose.
- **Amélioration (V2) :** Implémentation d'une politique d'éviction **LRU (Least Recently Used)** pour supprimer automatiquement les clés les moins utilisées quand la mémoire est pleine.