Projet CacheLab

Sous-titre

Yanis Bennadji

1. Contexte et Enjeux

1.1 Contexte du projet

CacheLab, startup spécialisée dans la performance web, a signé un contrat majeur avec plusieurs clients e-commerce. Ces clients rencontrent des problèmes critiques de lenteur sur leurs sites lors des pics de trafic (soldes, Black Friday), car leurs bases de données relationnelles traditionnelles (SQL) ne parviennent pas à gérer le volume de requêtes.

1.2 Objectif de la mission

L'objectif est de développer un MVP (Minimum Viable Product) d'un système de base de données **clé/valeur en mémoire**, inspiré de Redis. Ce système servira de tampon pour stocker les données fréquemment consultées (prix, paniers, profils), réduisant ainsi la charge sur les bases de données principales.

2. Analyse des Besoins Client

2.1 Besoins Métier

Les clients pilotes ont besoin d'une solution capable de :

- **Accélérer le temps de réponse** de leurs applications web.
- **Supporter une forte charge** de lecture et d'écriture simultanée.
- **Stocker des données temporaires** structurées simplement (Clé -> Valeur).
- **Cas d'usage identifiés :**
    - Mise en cache des paniers d'achats utilisateurs.
    - Stockage rapide des informations de session utilisateur.
    - Consultation instantanée des prix produits.

2.2 Exigences de Performance

- Le système doit offrir des temps de réponse quasi-instantanés, indépendants du volume de données stocké (complexité algorithmique cible)

- Le système doit privilégier la vitesse d'accès (stockage RAM) par rapport à la capacité de stockage disque.

3. Périmètre Fonctionnel (Le MVP)

Le système se présente sous la forme d'un serveur API REST exposant les fonctionnalités CRUD (Create, Read, Update, Delete) suivantes[1][2].

3.1 Gestion des données (Clé/Valeur)

Le système doit permettre les interactions suivantes via des appels HTTP standard :

Fonctionnalité	Verbe HTTP	Endpoint	Description Fonctionnelle	Critère d'acceptation
Création	POST	/keys	Enregistrer une nouvelle paire clé/valeur.	Si la clé n'existe pas, elle est créée. Retour code 201.
Lecture	GET	/keys/:key	Récupérer la valeur associée à une clé spécifique.	Retourne l'objet JSON stocké. Retour code 404 si introuvable.
Mise à jour	PUT	/keys/:key	Modifier la valeur d'une clé existante.	La valeur est remplacée instantanément. Retour code 200.
Suppression	DELETE	/keys/:key	Supprimer définitivement une clé et sa valeur.	La mémoire est libérée. Retour code 200 ou 404.
Lister	GET	/keys	Obtenir la liste de toutes les clés présentes.	Retourne un tableau JSON des clés.

3.2 Gestion des erreurs

Le système doit renvoyer des messages d'erreur explicites au format JSON pour aider les développeurs clients :

- Erreur 400 : Données manquantes ou format invalide.
- Erreur 404 : Ressource (clé) non trouvée.
- Erreur 500 : Erreur interne du serveur.

4. Contraintes Techniques et Qualité

4.1 Contraintes de Performance et Algorithmique

- L'architecture doit reposer sur une structure de données optimisée pour la recherche rapide.
- Utilisation d'une **HashMap (Table de Hachage)** imposée pour garantir une complexité moyenne de $O(1)$ sur toutes les opérations, assurant ainsi la scalabilité lors des pics de trafic[19].

4.2 Contraintes de Sécurité

- Validation stricte des entrées API pour éviter les injections ou les corruptions de mémoire.
- Les messages d'erreur ne doivent pas exposer de détails sensibles sur l'infrastructure (Stack Traces masquées en production).

4.3 Spécialisation et Implications [21]

Le choix s'est porté sur la stack **Node.js / TypeScript**[22].

- **Implication fonctionnelle :** L'utilisation de TypeScript garantit une meilleure fiabilité du code (typage fort) et réduit les risques de bugs en production ("undefined is not a function").
- **Approche "Low-Level" :** Pour maximiser la compréhension et la performance, aucune structure de haut niveau (Map, Set) ou méthode itérative lourde (filter, reduce) n'est utilisée. L'algorithme gère la mémoire manuellement via des tableaux et des références.

4.4 Éco-conception [23]

- Le stockage étant exclusivement en mémoire vive (RAM), le système doit être économe en ressources.
- La structure de données choisie (HashMap avec gestion des collisions par chaînage) optimise l'allocation mémoire en évitant les duplications inutiles.