Stratégie de Mise en Cache - PRD (Product Requirements Document)

Version: 1.0
Date de dernière mise à jour: 18 mai 2024

1. Introduction et Objectif

PokeWallet est une application Ionic/Angular permettant aux utilisateurs de gérer leur collection de cartes Pokémon. Actuellement, l'application effectue des requêtes à Firebase à chaque changement de page ou navigation, ce qui mobilise des ressources inutilement puisque les données ne changent que lors d'actions spécifiques de l'utilisateur.

L'objectif de cette fonctionnalité est d'implémenter une stratégie de mise en cache efficace qui permettra de :

- Réduire significativement le nombre d'appels à Firebase
- Améliorer la réactivité et les performances de l'application
- Fournir une meilleure expérience utilisateur avec des chargements plus rapides
- Optimiser l'utilisation des ressources (particulièrement importante sur mobile)
- Réduire potentiellement les coûts liés à l'utilisation de Firebase

La stratégie choisie repose sur un cache en mémoire via les Services Angular utilisant le pattern BehaviorSubject de RxJS, qui représente le meilleur compromis entre facilité d'implémentation et bénéfices immédiats.

2. Fonctionnalités Requises (User Stories)

US-1: Cache des cartes Pokémon

En tant qu'utilisateur, je souhaite que ma collection de cartes se charge instantanément lorsque je navigue entre les pages, **afin de** ne pas avoir à attendre chaque fois que je consulte mes cartes.

Critères d'acceptation:

- La première consultation des cartes après connexion récupère les données depuis Firebase
- Les consultations suivantes utilisent les données en cache sans appel réseau
- Le temps de chargement après la première consultation est proche de zéro

US-2: Mise à jour du cache lors d'ajout de carte

En tant qu'utilisateur, je souhaite que lorsque j'ajoute une nouvelle carte à ma collection, celle-ci apparaisse immédiatement dans toutes les vues sans rechargement complet, **afin de** bénéficier d'une expérience fluide et cohérente.

Critères d'acceptation:

- L'ajout d'une carte met à jour Firebase puis le cache local
- La nouvelle carte apparaît immédiatement dans toutes les vues sans appel supplémentaire à Firebase
- Toutes les vues abonnées à la liste des cartes sont mises à jour automatiquement

US-3: Mise à jour du cache lors de suppression/modification de carte

En tant qu'utilisateur, je souhaite que lorsque je supprime ou modifie une carte dans ma collection, ces changements se reflètent immédiatement partout dans l'application, **afin de** maintenir une cohérence des données.

Critères d'acceptation:

- La suppression/modification d'une carte met à jour Firebase puis le cache local
- Les changements apparaissent immédiatement dans toutes les vues sans appel supplémentaire à Firebase
- Toutes les vues abonnées à la liste des cartes sont mises à jour automatiquement

US-4: Réinitialisation du cache lors de la déconnexion

En tant qu'utilisateur, je souhaite que mes données soient correctement effacées du cache lorsque je me déconnecte, **afin de** protéger ma vie privée et d'éviter des mélanges de données entre utilisateurs.

Critères d'acceptation:

- Le cache est vidé lors de la déconnexion
- À la reconnexion d'un autre utilisateur, ses propres données sont chargées correctement sans contamination

3. Exigences Non Fonctionnelles

3.1 Performance

- Le temps de chargement des données depuis le cache doit être inférieur à 100ms
- La réactivité de l'application après la mise en cache doit être perçue comme "instantanée" par l'utilisateur
- Le mécanisme de cache ne doit pas augmenter l'empreinte mémoire de l'application de plus de 5%

3.2 Fiabilité

- Le système doit gérer correctement les cas de données invalides ou corrompues dans le cache
- En cas d'échec du chargement depuis le cache, un fallback vers Firebase doit être implémenté
- La cohérence entre le cache et Firebase doit être garantie après chaque opération de modification

3.3 Maintenabilité

- Le code du système de cache doit être modulaire et facilement extensible
- La documentation du fonctionnement du cache doit être claire et détaillée
- Des tests unitaires doivent couvrir au moins 80% du code de la fonctionnalité de cache

3.4 Compatibilité

- La solution doit être compatible avec Angular 18+ et Ionic 8+
- La solution doit fonctionner sur toutes les plateformes supportées par l'application (iOS, Android, Web)

4. Stratégie de Gestion des Erreurs et Fallbacks

4.1 Gestion des erreurs de chargement Firebase

- Si une erreur survient lors du chargement initial depuis Firebase, le système doit :
  - Consigner l'erreur dans les logs (console)
  - Afficher un message d'erreur approprié à l'utilisateur
  - Proposer une option de rechargement manuel des données
  - Réinitialiser l'état du cache pour cet ensemble de données

4.2 Gestion des erreurs d'opérations d'écriture

- Si une erreur survient lors de l'écriture dans Firebase (ajout/modification/suppression), le système doit :
  - Conserver une copie de l'état précédent du cache
  - Restaurer cet état en cas d'échec de l'opération
  - Afficher un message d'erreur informatif à l'utilisateur
  - Proposer de réessayer l'opération ou d'annuler

4.3 Fallback en cas de cache invalide

- Le système doit détecter les incohérences potentielles entre le cache et Firebase
- Une stratégie de revalidation périodique peut être implémentée (ex. recharger les données en arrière-plan à des intervalles définis)
- En cas de détection d'invalidité du cache, recharger silencieusement les données depuis Firebase

5. Contraintes Techniques

5.1 Architecture et Technologies

- Utilisation du pattern BehaviorSubject de RxJS pour le stockage en mémoire
- Respect des principes d'immutabilité lors des mises à jour du cache
- Implémentation dans les services Angular existants sans modification majeure de l'architecture

5.2 Compatibilité avec Firebase

- Respect des patterns d'utilisation du SDK Firebase
- Prise en compte des mécanismes de mise en cache déjà présents dans Firebase
- Évaluation et prévention des éventuels conflits entre notre cache et celui de Firebase

5.3 Limitations de la Solution

- Le cache en mémoire est perdu à la fermeture de l'application ou au rafraîchissement de la page
- Les données extrêmement volumineuses (>10MB) devraient être gérées avec pagination plutôt qu'un chargement complet en cache
- La solution ne couvre pas le fonctionnement hors-ligne (serait une évolution future avec Ionic Storage)

6. Scénarios d'Utilisation

Scénario 1: Première utilisation après connexion

1. L'utilisateur se connecte à l'application
2. Il accède à la page MyWallet qui affiche sa collection de cartes
3. Le service CardStorageService vérifie son cache (vide à ce stade)
4. Le service effectue un appel à Firebase pour récupérer les cartes
5. Les cartes sont stockées dans le BehaviorSubject et affichées à l'utilisateur
6. Le temps de chargement est celui d'un appel Firebase standard

Scénario 2: Navigation répétée

1. L'utilisateur, déjà connecté, navigue depuis MyWallet vers une autre page
2. Puis il revient à la page MyWallet
3. Le service CardStorageService vérifie son cache (déjà rempli)
4. Les cartes sont immédiatement récupérées du BehaviorSubject sans appel Firebase
5. L'affichage est instantané

Scénario 3: Ajout d'une carte

1. L'utilisateur ajoute une nouvelle carte via la page AddCard
2. Le service CardStorageService sauvegarde la carte dans Firebase
3. Puis il met à jour son BehaviorSubject avec la nouvelle liste incluant la carte ajoutée
4. Toutes les vues abonnées (comme MyWallet) sont automatiquement mises à jour
5. L'utilisateur voit sa nouvelle carte apparaître sans rechargement

Scénario 4: Déconnexion et changement d'utilisateur

1. L'utilisateur A se déconnecte
2. Le service CardStorageService vide son cache (BehaviorSubject réinitialisé)
3. L'utilisateur B se connecte
4. À sa première visite de MyWallet, le service chargera ses propres cartes depuis Firebase
5. Aucune donnée de l'utilisateur A ne persiste ou n'est visible

7. Critères d'Acceptation

La fonctionnalité de mise en cache sera considérée comme acceptée lorsque :

1. **Réduction des appels Firebase**
    - Le nombre d'appels à Firebase est réduit d'au moins 70% lors de l'utilisation normale de l'application
2. **Performance**
    - Après la première consultation, le temps de chargement des données depuis le cache est inférieur à 100ms
    - Pas de ralentissement perceptible de l'application avec le mécanisme de cache actif
3. **Cohérence des données**
    - Les données affichées sont toujours à jour après modification (ajout, suppression, édition)
    - Les changements sont visibles immédiatement dans toutes les vues concernées
4. **Gestion des erreurs**
    - Le système gère correctement toutes les erreurs potentielles identifiées
    - Les fallbacks fonctionnent comme prévu
    - L'utilisateur est informé de manière appropriée en cas d'erreur
5. **Tests**
    - Les tests unitaires couvrant les scénarios principaux (chargement, mise à jour, réinitialisation) passent avec succès
    - Les tests de performance confirment l'amélioration des temps de chargement

8. Mesure du Succès

Le succès de l'implémentation du système de cache sera mesuré par :

8.1 Métriques Quantitatives

- **Réduction des appels réseau** : Mesure du nombre d'appels à Firebase avant/après implémentation
- **Temps de réponse** : Mesure des temps de chargement des données avant/après implémentation
- **Utilisation des ressources** : Monitoring de l'utilisation de la mémoire et du CPU
- **Taux d'erreur** : Suivi du nombre d'erreurs liées au cache vs. autres erreurs

8.2 Métriques Qualitatives

- **Satisfaction utilisateur** : Feedback sur la réactivité perçue de l'application
- **Rapports de bugs** : Suivi des problèmes signalés liés à la synchronisation des données
- **Facilité de maintenance** : Évaluation de la complexité du code ajouté par les développeurs

8.3 KPIs Clés

- Diminution d'au moins 70% des appels à Firebase

- Augmentation de la vitesse de chargement perçue d'au moins 50%
- Maintien du taux d'erreur global en dessous de 1%
- Diminution des temps de chargement des pages de 60% en moyenne

Résumé des Fonctionnalités Angular Récentes et Pertinentes

Pour l'implémentation de cette stratégie de cache, plusieurs fonctionnalités récentes d'Angular sont particulièrement pertinentes :

RxJS et Observables

Angular 18 continue d'intégrer profondément RxJS, dont les BehaviorSubject sont un composant clé de notre stratégie. RxJS 7+ offre des améliorations de performance et une syntaxe plus claire pour les opérateurs.

Signals (Angular 18+)

Bien que notre implémentation initiale utilise BehaviorSubject, Angular 18 a introduit les "Signals" comme une alternative légère aux Observables pour la gestion d'état réactive. Ils pourraient être considérés pour une évolution future de notre système de cache.

Exemple :

```
// BehaviorSubject (approche actuelle)
private userCardsSubject = new BehaviorSubject<PokemonCard[] | null>(null);
public userCards$ = this.userCardsSubject.asObservable();

// Signal (approche potentielle future)
private userCards = signal<PokemonCard[] | null>(null);
```

Injection Légère

Angular 18 a amélioré le système d'injection de dépendances avec une syntaxe plus légère qui peut être utilisée dans notre implementation pour injecter les services nécessaires.

OnPush Change Detection

La détection de changement OnPush, particulièrement efficace avec les patterns d'immutabilité que nous utilisons dans notre stratégie de cache, peut améliorer davantage les performances des composants qui consomment les données en cache.