# Documentation d'Implémentation de la Stratégie de Mise en Cache

**Version:** 0.2
**Date de dernière mise à jour:** 18 mai 2024

## Documentation Fonctionnelle

### Présentation Générale

La stratégie de mise en cache implémentée dans PokeWallet a pour objectif d'optimiser les performances de l'application en réduisant les appels inutiles à Firebase. Actuellement, l'application effectue des requêtes à la base de données à chaque navigation ou rechargement de page, ce qui mobilise des ressources et ralentit l'expérience utilisateur.

La solution mise en place exploite les BehaviorSubjects de RxJS pour stocker les données en mémoire après leur première récupération. Ces données sont ensuite réutilisées lors des consultations ultérieures, sans nécessiter de nouveaux appels à Firebase.

### Fonctionnalités Implémentées

À ce stade de l'implémentation (Phase 1 - Préparation et Architecture), les fonctionnalités suivantes ont été réalisées :

- **Analyse des services existants :** Les services CardStorageService, UserService, HistoryService et CollectionHistoryService ont été identifiés comme cibles principales pour l'optimisation par mise en cache.
- **Architecture de cache :** Une architecture standardisée basée sur BehaviorSubject a été définie pour tous les services, avec une gestion cohérente du cycle de vie du cache. L'interface `CacheableService<T>` et la classe abstraite `BaseCacheService<T>` ont été créées pour faciliter l'implémentation cohérente du pattern de cache.
- **Tests unitaires :** Une suite de tests unitaires a été mise en place pour valider le fonctionnement du cache, sa réinitialisation et sa gestion des erreurs.

### Bénéfices Attendus

L'implémentation complète de cette stratégie apportera les avantages suivants :

- **Réactivité accrue :** Les temps de chargement des pages après la première consultation seront pratiquement instantanés.
- **Économie de ressources :** Réduction significative des appels à Firebase, optimisant l'utilisation des quotas et la consommation de batterie sur mobile.
- **Expérience utilisateur fluide :** Les données modifiées seront immédiatement visibles dans toutes les vues, sans rechargement complet.
- **Robustesse :** Mise en place de mécanismes de gestion des erreurs et de fallback pour assurer la fiabilité.

## Documentation Technique

### État Actuel (Avant Optimisation)

#### Analyse des Services Firebase

L'analyse des services a révélé que plusieurs services effectuent des appels répétitifs à Firebase :

1. **CardStorageService :**
   - Gère déjà des BehaviorSubjects ( `cardsSubject` , `totalValueSubject` ) mais recharge depuis Firebase à chaque visite
   - Contient des méthodes de CRUD qui mettent à jour Firebase mais pas toujours le cache de manière optimale
   - Ne gère pas explicitement un cache avec identification de l'utilisateur courant
2. **UserService :**
   - Utilise un BehaviorSubject pour l'état d'authentification
   - Charge les données utilisateur à chaque changement d'état d'authentification
   - Ne met pas en cache les résultats des recherches ou les détails des amis
3. **HistoryService et CollectionHistoryService :**
   - Effectuent des appels répétitifs à Firebase même pour des données qui changent rarement
   - N'implémentent pas de mécanisme de cache

### Architecture de Cache Standardisée

Pour standardiser la mise en cache à travers les services, nous avons défini les principes architecturaux suivants et créé deux fichiers essentiels :

**1. Interface `CacheableService<T>` (src/app/interfaces/cacheable-service.interface.ts)**

Cette interface définit le contrat que tous les services avec cache doivent respecter :

```
export interface CacheableService<T> {
  // Observables pour les données et l'état du cache
  data$: Observable<T | null>;
  isLoading$: Observable<boolean>;
  hasError$: Observable<boolean>;

  // Méthodes principales
  getData(): Observable<T | null>;
  clearCache(): void;
  reloadData(): Promise<void>;
  hasCachedData(): boolean;
}
```

**2. Classe abstraite `BaseCacheService<T>` (src/app/services/base-cache.service.ts)**

Cette classe fournit une implémentation de base du pattern de cache :

```
export abstract class BaseCacheService<T> implements CacheableService<T> {
  // BehaviorSubjects pour stocker les données et l'état
  protected dataSubject = new BehaviorSubject<T | null>(null);
  protected loadingSubject = new BehaviorSubject<boolean>(false);
  protected errorSubject = new BehaviorSubject<boolean>(false);

  // Observables publiques
  public data$ = this.dataSubject.asObservable();
  public isLoading$ = this.loadingSubject.asObservable();
```

```
  public hasError$ = this.errorSubject.asObservable();

  // Méthode abstraite à implémenter dans chaque service
  protected abstract fetchFromSource(userId: string): Promise<T>;

  // Implémentation des méthodes de l'interface
  public getData(userId?: string): Observable<T | null> { ... }
  protected async loadData(userId: string): Promise<void> { ... }
  public async reloadData(): Promise<void> { ... }
  public clearCache(): void { ... }
  public hasCachedData(): boolean { ... }
}
```

#### Cycle de Vie du Cache

Le cycle de vie du cache est géré de manière standardisée :

1. **Initialisation :** Le cache commence vide ( `null` )
2. **Premier accès :** `getData()` vérifie si les données sont en cache, sinon appelle `loadData()` qui charge les données depuis la source via `fetchFromSource()`
3. **Accès suivants :** `getData()` retourne les données en cache si elles sont disponibles pour l'utilisateur demandé
4. **Rechargement forcé :** `reloadData()` force un rechargement des données depuis la source
5. **Réinitialisation :** `clearCache()` vide le cache et réinitialise l'état

#### Gestion des Erreurs

La gestion des erreurs est intégrée au cycle de vie du cache :

1. **Erreur de chargement initial :** L'erreur est capturée, logguée et signalée via `errorSubject`
2. **Erreur lors du rechargement :** Si des données existaient déjà en cache, elles sont conservées

### Tests Unitaires

Une suite complète de tests unitaires a été créée dans `src/app/services/base-cache.service.spec.ts` pour valider le comportement du cache :

- **Initialisation :** Vérification que le cache est initialement vide
- **Chargement :** Test du chargement des données depuis la source
- **Utilisation du cache :** Vérification que les requêtes suivantes utilisent le cache
- **Rechargement :** Test du rechargement forcé des données
- **Réinitialisation :** Vérification que le cache peut être vidé
- **Gestion des erreurs :** Test des scénarios d'erreur de chargement et de rechargement

### Prochaines Étapes

Les prochaines étapes de l'implémentation (Phase 2) consisteront à :

1. Refactorer CardStorageService pour étendre BaseCacheService et implémenter la méthode fetchFromSource
2. Optimiser les méthodes d'ajout, de suppression et de modification des cartes pour mettre à jour le cache
3. Intégrer la réinitialisation du cache au processus de déconnexion