Guide de Présentation - Projet de Caching

📋 Table des Matières

🏗️ Architecture Globale

Stack Technologique

Backend (Go):

- **Framework:** Chi Router
- **Base de données:** MySQL (via GORM)
- **Cache:** Redis
- **Authentification:** JWT
- **Architecture:** Clean Architecture (Handler → Service → Repository)

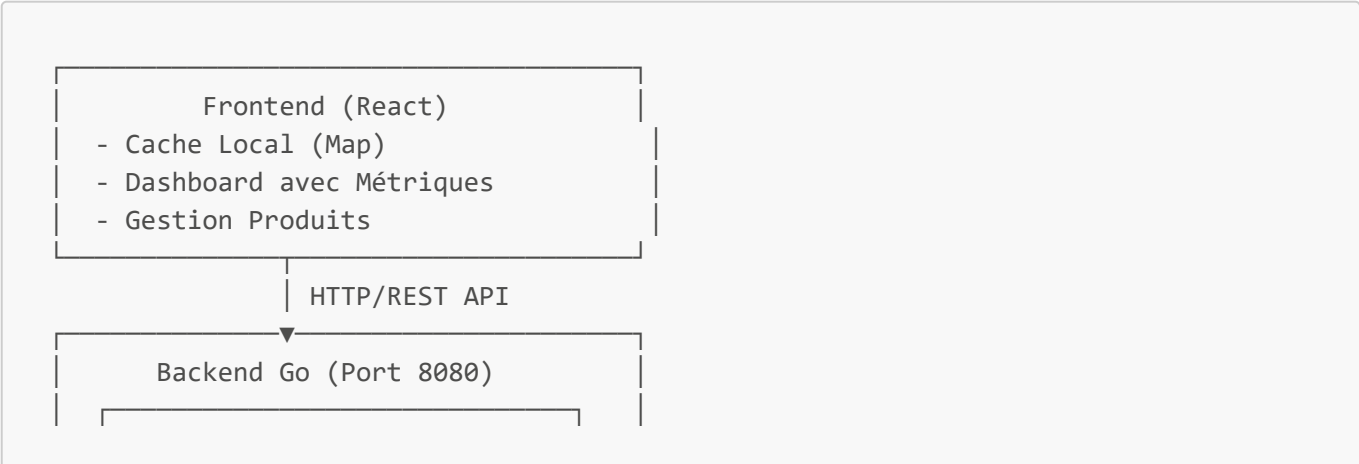
Frontend (React + TypeScript):

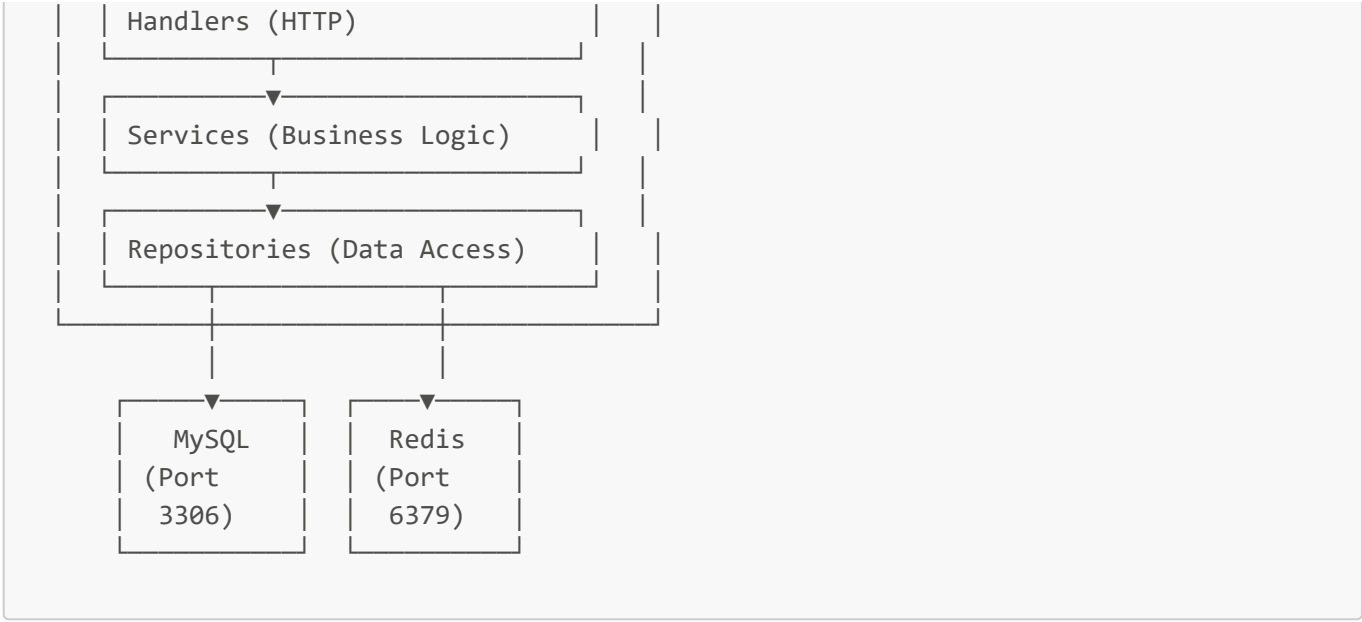
- **Framework:** React 18 + Vite
- **State Management:** Zustand (avec persistence)
- **UI:** Tailwind CSS + Radix UI
- **Cache Local:** Map JavaScript (simulation RAM)

Infrastructure:

- **Docker Compose:** Orchestration des services (Go API, MySQL, Redis)
- **Réseau:** Réseau Docker isolé (`app_net`)

Architecture en Couches





📖 Explication du README.md

1. Qu'est-ce que le Caching ?

Définition du README:

> "Le caching est une technique utilisée pour stocker des données fréquemment accédées dans une couche de stockage rapide afin de réduire le temps d'accès futur."

Explication:

- **Problème résolu:** Les accès à la base de données sont lents (latence réseau, I/O disque)
- **Solution:** Stocker une copie des données dans la RAM (Redis) pour accès ultra-rapide
- **Métaphore:** Un bloc-notes où on garde les détails clés plutôt que de chercher dans les archives

Dans votre code:

- Redis stocke les produits en mémoire (accès microsecondes)
- MySQL stocke les données persistantes (accès millisecondes)
- Le frontend a aussi un cache local (Map) pour démonstration

---

2. Types de Caches

A. In-Memory Cache (Votre Implémentation)

README dit:

> "Les données sont stockées directement dans la RAM du serveur"

Votre implémentation:

- **Backend:** Redis (stockage en RAM)
- **Frontend:** `Map<string, Product>` dans Zustand (simulation RAM)
- **Limite:** 1 MB dans le frontend (`MAX_CACHE_SIZE = 1024 * 1024`)

Code correspondant:

```
// Frontend/src/store/cacheStore.ts
const MAX_CACHE_SIZE = 1024 * 1024 // 1 MB
cache: Map<string, Product> // Stockage en mémoire
```

```
// Backend/pkg/cache/redisconnect.go
rdb := redis.NewClient(&redis.Options{
    Addr: "redis:6379", // Redis en mémoire
})
```

B. Distributed Cache (Votre Implémentation)

README dit:

> "Approprié pour les applications scalables où plusieurs instances doivent partager les mêmes données"

Votre implémentation:

- Redis est un cache distribué
- Plusieurs instances Go pourraient partager le même Redis
- Les locks Redis permettent la synchronisation

---

3. Stratégies de Chargement (Loading Strategies)

A. Cache-Aside (Implémenté dans votre code)

README dit:

> "Si les données sont demandées et pas dans le cache, alors charger ces données dans le cache"

Votre implémentation Backend:

Fichier: `Backend/internal/repositories/product_repositorie.go`

```
// GetProductByID - Cache-Aside Pattern
func (pr *ProductRepositorie) GetProductByID(ctx context.Context, id uint)
(*models.Product, error) {
    pKey := fmt.Sprintf("product:%d", id)

    // 1. CHECK CACHE FIRST
    err := pr.cache.HGetAll(ctx, pKey).Scan(&product)
    if err == nil && product.ID != 0 {
        fmt.Println("Cache HIT for product:", id) // ✅ TROUVÉ DANS CACHE
        return &product, nil
    }
```

```
    // 2. CACHE MISS - Fetch from Database
    fmt.Println("Cache MISS for product:", id)
    err = pr.db.WithContext(ctx).First(&product, id).Error

    // 3. STORE IN CACHE for future requests
    pr.cache.HSet(ctx, pKey, "id", product.ID, "name", product.Name, "price",
product.Price)

    return &product, nil
}
```

Flux:

1. ✅ **Cache HIT:** Données trouvées dans Redis → Retour immédiat
2. ❌ **Cache MISS:** Pas dans Redis → Fetch MySQL → Stocke dans Redis → Retour

Frontend - Même Pattern:

```
// Frontend/src/store/cacheStore.ts
getProduct: async (id: string) => {
    // 1. Check local cache
    if (cache.has(id)) {
        // CACHE HIT
        return cache.get(id)!
    }

    // 2. CACHE MISS - Fetch from backend
    const response = await productAPI.getById(parseInt(id))

    // 3. Store in cache
    cache.set(id, product)
    return product
}
```

Pourquoi Cache-Aside?

- ✅ Ne charge que les données demandées (pas de pré-chargement inutile)
- ✅ Cache et DB sont découplés
- ✅ Si Redis tombe, l'app fonctionne toujours (fallback DB)

B. Refresh-Ahead (Non implémenté - mentionné dans README)

README explique:

> "Rafraîchit les données populaires avant qu'elles expirent"

Pourquoi pas implémenté:

- Complexité supplémentaire

- Votre projet se concentre sur Cache-Aside et Write-Through

---

4. Stratégies d'Écriture (Writing Strategies)

A. Write-Through (Implémenté)

README dit:

> "Met à jour simultanément le cache et la base de données"

Votre implémentation:

Fichier: `Backend/internal/repositories/product_repositorie.go`

```
// CreateProduct - Write-Through Pattern
func (pr *ProductRepositorie) CreateProduct(ctx context.Context, product
*models.Product) error {
    // 1. WRITE TO DATABASE FIRST (Source of Truth)
    err := pr.db.WithContext(ctx).Create(product).Error
    if err != nil {
        return err
    }

    // 2. IMMEDIATE CACHE UPDATE
    pKey := fmt.Sprintf("product:%d", product.ID)
    pr.cache.HSet(ctx, pKey, "id", product.ID, "name", product.Name, "price",
product.Price)
    pr.cache.SAdd(ctx, "products:all_ids", product.ID) // Index pour
GetAllProducts

    return nil
}
```

Flux Write-Through:

```
Application Write Request
    ↓
Write to MySQL (DB) ✅
    ↓
Write to Redis (Cache) ✅
    ↓
Return Confirmation
```

Avantages:

- ✅ **Cohérence forte:** Cache et DB toujours synchronisés
- ✅ **Pas de données obsolètes** dans le cache
- ❌ **Coût:** 2 écritures au lieu d'1 (mais garantit la cohérence)

UpdateProduct avec Redis Lock:

```
func (pr *ProductRepositorie) UpdateProduct(ctx context.Context, product
*models.Product) error {
    lockKey := fmt.Sprintf("lock:product:%d", product.ID)

    // 1. ACQUIRE REDIS LOCK (Prevent Race Conditions)
    ok, err := pr.cache.SetNX(ctx, lockKey, "1", 5*time.Second).Result()
    if err != nil || !ok {
        return errors.New("product is being updated, try again")
    }
    defer pr.cache.Del(ctx, lockKey) // Release lock

    // 2. UPDATE DATABASE
    pr.db.WithContext(ctx).Model(&models.Product{}).Where("id = ?",
product.ID).Updates(...)

    // 3. UPDATE CACHE (Write-Through)
    pr.cache.HSet(ctx, pKey, "id", product.ID, "name", product.Name, "price",
product.Price)

    return nil
}
```

Pourquoi le Lock?

- **Problème:** 2 requêtes simultanées peuvent modifier le même produit
- **Solution:** Lock Redis (`SetNX`) garantit qu'une seule mise à jour à la fois
- **Timeout:** 5 secondes (évite les deadlocks)

B. Write-Aside (Non implémenté - mentionné dans README)

README dit:

> "Écrit seulement dans la base de données, ignore le cache"

Pourquoi pas implémenté:

- Votre projet privilégie la cohérence (Write-Through)

C. Write-Behind (Non implémenté - mentionné dans README)

README dit:

> "Écrit dans le cache, puis met à jour la DB de manière asynchrone"

Pourquoi pas implémenté:

- Risque de perte de données si crash avant écriture DB

- Complexité supplémentaire

---

5. Politiques d'Éviction (Eviction Policies)

A. LRU - Least Recently Used (Implémenté)

README dit:

> "La clé qui n'a pas été accédée depuis le plus longtemps est évincée"

Votre implémentation Frontend:

Fichier: `Frontend/src/store/cacheStore.ts`

```
// Evict LRU items if needed
while (currentSize + product.size > MAX_CACHE_SIZE && updatedCache.size > 0) {
    // Find LRU item (lowest lastAccessed)
    let lruKey = ''
    let lruTime = Infinity

    updatedCache.forEach((p, key) => {
        if ((p.lastAccessed || 0) < lruTime) {
            lruTime = p.lastAccessed || 0
            lruKey = key
        }
    })

    // Remove LRU item
    if (lruKey) {
        const evictedProduct = updatedCache.get(lruKey)!
        evictionEvents.push({
            type: 'eviction',
            productId: lruKey,
            productName: evictedProduct.name,
        })
        currentSize -= evictedProduct.size
        updatedCache.delete(lruKey)
    }
}
```

Comment ça marche:

1. Chaque produit a un `lastAccessed` (timestamp)
2. Quand la mémoire est pleine, on trouve le produit avec le `lastAccessed` le plus ancien
3. On le supprime pour faire de la place

Exemple:

- Produit A: `lastAccessed = 1000` (ancien)
- Produit B: `lastAccessed = 2000` (récent)

- Produit C: `lastAccessed = 1500` (moyen)
- **Éviction:** Produit A (le moins récemment utilisé)

Mise à jour de `lastAccessed`:

```
// Quand on accède à un produit (Cache HIT)
const updatedProduct = {
    ...product,
    lastAccessed: Date.now(), // ✅ Mise à jour du timestamp
    accessCount: product.accessCount + 1,
}
```

B. LFU, FIFO, Random (Mentionnés dans README, non implémentés)

Pourquoi seulement LRU?

- LRU est la politique la plus courante
- Simple à implémenter
- Efficace pour la plupart des cas d'usage

---

6. Stratégies d'Invalidation

A. Manual (Implémenté)

README dit:

> "Quand l'application met à jour le stockage, elle met à jour l'entrée du cache"

Votre implémentation:

- `UpdateProduct` met à jour DB + Cache manuellement
- `CreateProduct` ajoute dans DB + Cache manuellement

B. TTL Based (Non implémenté)

Pourquoi pas?

- Votre projet se concentre sur Write-Through (cohérence immédiate)
- TTL serait utile pour des données moins critiques

C. Write-Event Based (Implémenté indirectement)

Votre implémentation:

- Chaque write déclenche une mise à jour cache (Write-Through)
- Équivalent à Write-Event Based

---

7. Métriques de Performance

A. Cache Hit Rate

Formule du README:

```
Cache Hit Rate = (Cache hits / Total Requests) × 100
```

Votre implémentation:

Frontend: `Frontend/src/store/cacheStore.ts`

```
metrics: {
    totalRequests: 0,
    cacheHits: 0,
    cacheMisses: 0,
    hitRate: 0, // Calculé automatiquement
    missRate: 0,
}

// Quand Cache HIT
const newMetrics = {
    ...state.metrics,
    totalRequests: state.metrics.totalRequests + 1,
    cacheHits: state.metrics.cacheHits + 1,
    hitRate: (state.metrics.cacheHits + 1) / (state.metrics.totalRequests + 1), //
✅ Calcul
    missRate: state.metrics.cacheMisses / (state.metrics.totalRequests + 1),
}
```

Affichage: `Frontend/src/pages/Dashboard.tsx`

```
<MetricsCard
    title="Hit Rate"
    value={formatPercentage(metrics.hitRate)} // Ex: "75.5%"
    subtitle={`${metrics.cacheHits} hits out of ${metrics.totalRequests}
requests`}
/>
```

Interprétation:

- **> 70%:** Excellent (vert, "Good")
- **30-70%:** Modéré (jaune, "Moderate")
- **< 30%:** Faible (rouge, "Low")

B. Cache Miss Rate

Formule:

```
Cache Miss Rate = (Cache Miss / Total Requests) × 100
```

Implémentation: Même principe que Hit Rate

C. Eviction Rate

Formule:

```
Eviction Rate = Number of evictions / Time Period
```

Votre implémentation:

- Compteur total d'évictions (pas de période spécifique)
- Affiché dans le dashboard

```
<MetricsCard
    title="Total Evictions"
    value={metrics.evictions}
    subtitle="Items removed due to memory limit"
/>
```

D. Data Freshness

Formule:

```
Data Age = Current Time - Last Update Time
```

Non implémenté explicitement, mais:

- Write-Through garantit que les données sont toujours à jour
- Pas besoin de calculer l'âge si on met à jour immédiatement

---

8. Cache Locks

README dit:

> "Verrouille une certaine clé pour éviter les conditions de course"

Votre implémentation:

Fichier: Backend/internal/repositories/product_repositorie.go

```
func (pr *ProductRepositorie) UpdateProduct(...) error {
    lockKey := fmt.Sprintf("lock:product:%d", product.ID)

    // ACQUIRE LOCK
    ok, err := pr.cache.SetNX(ctx, lockKey, "1", 5*time.Second).Result()
    if err != nil || !ok {
        return errors.New("product is being updated, try again")
    }

    defer pr.cache.Del(ctx, lockKey) // RELEASE LOCK (même en cas d'erreur)

    // ... Update DB and Cache ...
}
```

Comment ça marche:

1. **SetNX** (Set if Not eXists): Crée la clé seulement si elle n'existe pas
2. **Timeout:** 5 secondes (évite les deadlocks)
3. **Defer Del:** Libère le lock même en cas d'erreur

Scénario sans lock:

```
Requête 1: Read product.price = 100
Requête 2: Read product.price = 100
Requête 1: Update product.price = 110 → Write DB + Cache
Requête 2: Update product.price = 120 → Write DB + Cache
Résultat: 120 (perte de la mise à jour de la requête 1)
```

Scénario avec lock:

```
Requête 1: Acquire lock → Update 110 → Release lock
Requête 2: Wait... → Acquire lock → Update 120 → Release lock
Résultat: 120 (cohérent, pas de perte)
```

💻 Implémentation dans le Code

Architecture Backend

1. Models (Données)

Fichier: Backend/internal/models/product.go

```
type Product struct {
    ID    uint   `json:"id" redis:"id" gorm:"primaryKey;autoIncrement"`
    Name  string `json:"name" redis:"name" gorm:"size:100;unique"
validate:"required"`
    Price string `json:"price" redis:"price" gorm:"size:100" validate:"required"`
}
```

Tags:

- `json:` → Sérialisation JSON pour API
- `redis:` → Mapping pour Redis Hash
- `gorm:` → Mapping pour MySQL
- `validate:` → Validation des données

---

2. Repository (Accès aux Données)

Fichier: `Backend/internal/repositories/product_repositorie.go`

Structure:

```
type ProductRepositorie struct {
    db    *gorm.DB      // MySQL
    cache *redis.Client // Redis
}
```

Méthodes:

- `CreateProduct` → Write-Through
- `GetAllProducts` → Cache-Aside
- `GetProductByID` → Cache-Aside
- `UpdateProduct` → Write-Through + Lock

Structure Redis:

- **Hash:** `product:{id}` → `{id, name, price}`
- **Set:** `products:all_ids` → `{1, 2, 3, ...}` (index pour GetAllProducts)
- **Lock:** `lock:product:{id}` → `"1"` (avec TTL 5s)

---

3. Service (Logique Métier)

Fichier: `Backend/internal/services/product_service.go`

Rôle: Couche intermédiaire (peut ajouter de la logique métier)

```go
type ProductService struct {
    repo ProductRepository
}

func (ps *ProductService) GetAllProducts(ctx context.Context) ([]models.Product,
error) {
    return ps.repo.GetAllProducts(ctx) // Délègue au repository
}
```

Pourquoi cette couche?

- Séparation des responsabilités
- Facilite les tests
- Permet d'ajouter de la logique métier sans toucher au repository

---

4. Handler (HTTP)

Fichier: `Backend/internal/handlers/product_handler.go`

Rôle: Gère les requêtes HTTP

```go
func (ph *ProductHandler) GetAllProducts(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()
    w.Header().Set("Content-Type", "application/json")

    products, err := ph.serv.GetAllProducts(ctx)
    if err != nil {
        utils.Error(w, http.StatusInternalServerError, err)
        return
    }

    utils.Success(w, products) // JSON response
}
```

Endpoints:

- `GET /api/products/all` → GetAllProducts
- `GET /api/products/getbyid/{id}` → GetProductByID
- `POST /api/products/create` → CreateProduct
- `PUT /api/products/update` → UpdateProduct

---

5. Router

Fichier: `Backend/internal/router/router.go`

```
func MainRoutes() *chi.Mux {
    apiroute := chi.NewRouter()
    apiroute.Use(middlewares.CORSMiddleware) // CORS pour frontend

    apiroute.Route("/api", func(r chi.Router) {
        r.Mount("/users", UserRoutes())
        r.Mount("/products", ProductRoutes())
    })

    return apiroute
}
```

Architecture Frontend

1. Store (Zustand)

Fichier: `Frontend/src/store/cacheStore.ts`

Structure:

```
interface CacheState {
    cache: Map<string, Product>        // Cache local (simulation RAM)
    metrics: CacheMetrics              // Métriques
    events: CacheEvent[]               // Journal des événements
    loadingProducts: Set<string>       // Produits en cours de chargement

    // Actions
    getProduct: (id: string) => Promise<Product | null>
    addProduct: (product) => Promise<Product>
    updateProduct: (id, updates) => Promise<Product | null>
    deleteProduct: (id) => void
    clearCache: () => Promise<void>
    resetMetrics: () => void
}
```

Persistence:

```
persist(
    (set, get) => ({ ... }),
    {
        name: 'cache-store',
        partialize: (state) => ({
            cache: Array.from(state.cache.entries()),
            metrics: state.metrics,
            events: state.events,
        }),
```

```
    }
)
```

Pourquoi Zustand?

- ✅ Simple et léger
- ✅ Persistence automatique (localStorage)
- ✅ Réactivité automatique (React re-render)

---

2. API Client

Fichier: `Frontend/src/lib/api.ts`

Structure:

```
class ApiClient {
    private baseURL: string
    private token: string | null

    async request<T>(endpoint: string, options: RequestInit):
Promise<ApiResponse<T>> {
        // Ajoute Authorization header si token existe
        // Gère les erreurs
        // Retourne réponse JSON
    }
}

export const productAPI = {
    getAll: () => apiClient.get('/products/all'),
    getById: (id) => apiClient.get(`/products/getbyid/${id}`),
    create: (product) => apiClient.post('/products/create', product),
    update: (product) => apiClient.put('/products/update', product),
}
```

---

3. Dashboard

Fichier: `Frontend/src/pages/Dashboard.tsx`

Composants:

- `MetricsCard` → Affiche Hit Rate, Miss Rate, Evictions, Total Requests
- `MemoryGauge` → Jauge de mémoire (1 MB max)
- `EventLog` → Journal des 50 derniers événements
- `ProductTable` → Tableau des produits avec actions

Mise à jour en temps réel:

```
const metrics = useCacheStore((state) => state.metrics) // ✅ Réactif
const events = useCacheStore((state) => state.events)   // ✅ Réactif
```

🔄 Flux de Données

Scénario 1: Création d'un Produit (Write-Through)

```
1. User clique "Add Product" dans Frontend
   ↓
2. Frontend: productAPI.create({ name, price })
   ↓
3. Backend: POST /api/products/create
   ↓
4. Handler: CreateProduct()
   ↓
5. Service: CreateProduct() → délègue
   ↓
6. Repository: CreateProduct()
   ├─→ MySQL: INSERT INTO products (name, price) VALUES (...)
   └─→ Redis: HSET product:{id} id name price
   └─→ Redis: SADD products:all_ids {id}
   ↓
7. Response: { code: 200, data: { id, name, price } }
   ↓
8. Frontend: addProduct() → Ajoute au cache local
   ↓
9. Dashboard: Met à jour automatiquement (réactivité Zustand)
```

Scénario 2: Lecture d'un Produit (Cache-Aside)

Cas A: Cache HIT (dans Redis)

```
1. User clique "Fetch" sur un produit
   ↓
2. Frontend: getProduct(id)
   ├─→ Check cache local: MISS (pas encore chargé)
   └─→ productAPI.getById(id)
   ↓
3. Backend: GET /api/products/getbyid/{id}
   ↓
4. Repository: GetProductByID()
   ├─→ Redis: HGETALL product:{id}
   └─→ ✅ CACHE HIT → Retourne immédiatement
   ↓
5. Response: { code: 200, data: { id, name, price } }
   ↓
```

```
6. Frontend: Cache local → Ajoute au cache
   ↓
7. Dashboard: Affiche produit + Event "HIT"
```

Cas B: Cache MISS (pas dans Redis)

```
1. User clique "Fetch" sur un produit
   ↓
2. Frontend: getProduct(id) → productAPI.getById(id)
   ↓
3. Backend: GET /api/products/getbyid/{id}
   ↓
4. Repository: GetProductByID()
   ├─→ Redis: HGETALL product:{id} → ❌ EMPTY
   ├─→ MySQL: SELECT * FROM products WHERE id = ?
   ├─→ Redis: HSET product:{id} ... (remplit le cache)
   └─→ Retourne produit
   ↓
5. Response: { code: 200, data: { id, name, price } }
   ↓
6. Frontend: Cache local → Ajoute au cache
   ↓
7. Dashboard: Affiche produit + Event "MISS"
```

Scénario 3: Mise à Jour avec Lock

```
1. User clique "Edit Product"
   ↓
2. Frontend: updateProduct(id, { name, price })
   ↓
3. Backend: PUT /api/products/update
   ↓
4. Repository: UpdateProduct()
   ├─→ Redis: SETNX lock:product:{id} "1" EX 5
   │   └─→ ✅ Lock acquis
   ├─→ MySQL: UPDATE products SET name=?, price=? WHERE id=?
   ├─→ Redis: HSET product:{id} name price
   └─→ Redis: DEL lock:product:{id} (release lock)
   ↓
5. Response: { code: 200, data: { id, name, price } }
   ↓
6. Frontend: Met à jour le cache local
```

Si 2 requêtes simultanées:

```
Requête 1: SETNX lock:product:1 → ✅ OK (lock acquis)
Requête 2: SETNX lock:product:1 → ❌ FAIL (lock déjà pris)
           → Retourne erreur: "product is being updated, try again"
Requête 1: Update DB + Cache → DEL lock
Requête 2: Peut réessayer maintenant
```

Scénario 4: Éviction LRU

```
1. Cache local: 950 KB utilisés (limite: 1024 KB)
2. User ajoute un produit de 200 KB
   ↓
3. Frontend: addProduct()
   ├─→ Calcul: 950 + 200 = 1150 KB > 1024 KB ❌
   ├─→ Trouve LRU: Produit A (lastAccessed = 1000)
   ├─→ Supprime Produit A (libère 150 KB)
   ├─→ Calcul: 800 + 200 = 1000 KB < 1024 KB ✅
   └─→ Ajoute nouveau produit
   ↓
4. Dashboard: Event "EVICTION" pour Produit A
```

🎯 Stratégies de Cache Implémentées

Résumé

Stratégie	Implémentée?	Où?	Pourquoi?
Cache-Aside	✅ Oui	Backend + Frontend	Chargement à la demande
Write-Through	✅ Oui	Backend	Cohérence forte
Write-Aside	❌ Non	-	Pas de besoin
Write-Behind	❌ Non	-	Risque de perte de données
LRU Eviction	✅ Oui	Frontend	Gestion mémoire limitée
Redis Locks	✅ Oui	Backend	Prévention race conditions
TTL	❌ Non	-	Write-Through garantit fraîcheur
Refresh-Ahead	❌ Non	-	Complexité supplémentaire

❓ Questions Probables du Jury

Questions sur le README

Q1: "Expliquez la différence entre Cache-Aside et Write-Through"

Réponse:

- **Cache-Aside (Reads):** L'application vérifie le cache d'abord. Si MISS, elle va chercher dans la DB et remplit le cache. Le cache et la DB sont découplés.
- **Write-Through (Writes):** L'application écrit simultanément dans la DB et le cache. Garantit la cohérence mais coûte plus cher (2 écritures).

Dans votre code:

- Cache-Aside: `GetProductByID()` → Check Redis → Si MISS, fetch MySQL → Store Redis
- Write-Through: `CreateProduct()` → Write MySQL → Write Redis immédiatement

---

Q2: "Pourquoi utiliser LRU et pas LFU?"

Réponse:

- **LRU:** Supprime le moins récemment utilisé. Bon si les données récemment accédées sont plus susceptibles d'être réutilisées.
- **LFU:** Supprime le moins fréquemment utilisé. Bon si certaines données sont toujours populaires.

Votre choix: LRU car plus simple et efficace pour la plupart des cas. Les données récemment consultées sont souvent réutilisées.

Q3: "Qu'est-ce qu'un cache distribué?"

Réponse: Un cache partagé entre plusieurs serveurs/instances. Dans votre projet, Redis est un cache distribué car plusieurs instances Go pourraient se connecter au même Redis et partager les données.

Avantage: Cohérence entre toutes les instances.

Q4: "Pourquoi utiliser des locks Redis?"

Réponse: Pour éviter les **race conditions** lors de mises à jour concurrentes. Sans lock, 2 requêtes peuvent lire la même valeur, la modifier, et écraser les modifications l'une de l'autre.

Votre implémentation: `SetNX` crée un lock avec timeout. Une seule requête peut modifier un produit à la fois.

Questions sur le Code

Q5: "Expliquez la structure Redis dans votre code"

Réponse:

```
// Hash pour un produit
product:1 → { id: 1, name: "iPhone", price: "999" }
```

```
// Set pour l'index de tous les IDs
products:all_ids → { 1, 2, 3, 4, ... }

// Lock pour les mises à jour
lock:product:1 → "1" (avec TTL 5s)
```

Pourquoi Hash?

- Permet de stocker plusieurs champs (id, name, price) sous une seule clé
- `HGETALL` récupère tout d'un coup

Pourquoi Set pour all_ids?

- `GetAllProducts()` peut vérifier rapidement si le cache est vide
- Si le set est vide → Cache MISS → Fetch DB

Q6: "Comment fonctionne la gestion de la mémoire dans le frontend?"

Réponse:

```
const MAX_CACHE_SIZE = 1024 * 1024 // 1 MB

// Calcul de la taille actuelle
let currentSize = 0
cache.forEach(p => currentSize += p.size)

// Éviction si nécessaire
while (currentSize + newProduct.size > MAX_CACHE_SIZE) {
    // Trouve LRU
    // Supprime LRU
    // Réduit currentSize
}
```

Chaque produit a un `size` (512-1536 bytes aléatoire) pour simuler la taille réelle en mémoire.

Q7: "Pourquoi 2 caches? (Redis backend + Map frontend)"

Réponse:

- **Redis (Backend):** Cache distribué, partagé entre toutes les instances. Persiste même si le frontend se recharge.
- **Map (Frontend):** Cache local pour démonstration des concepts (LRU, métriques, éviction). Simule un cache in-memory.

Avantage: Le frontend peut fonctionner même si le backend est lent (cache local).

Q8: "Comment calculez-vous le Hit Rate?"

Réponse:

```
// À chaque requête
totalRequests++

// Si Cache HIT
cacheHits++
hitRate = cacheHits / totalRequests

// Si Cache MISS
cacheMisses++
missRate = cacheMisses / totalRequests
```

Mise à jour en temps réel: Zustand déclenche un re-render React automatiquement.

Q9: "Que se passe-t-il si Redis tombe?"

Réponse:

- **Backend:** Continue de fonctionner (fallback MySQL). Les requêtes seront plus lentes mais l'app ne crash pas.
- **Frontend:** Le cache local continue de fonctionner. Seules les nouvelles données nécessitent le backend.

Amélioration possible: Ajouter un circuit breaker ou retry logic.

Q10: "Pourquoi utiliser Clean Architecture (Handler → Service → Repository)?"

Réponse:

- **Séparation des responsabilités:** Chaque couche a un rôle précis
- **Testabilité:** On peut mocker chaque couche indépendamment
- **Maintenabilité:** Facile d'ajouter de la logique sans toucher aux autres couches
- **Réutilisabilité:** Le service peut être utilisé par d'autres handlers (ex: GraphQL)

Questions Techniques Avancées

Q11: "Comment optimiseriez-vous ce système?"

Réponses possibles:

1. **TTL sur Redis:** Ajouter expiration automatique pour éviter données obsolètes
2. **Cache Warming:** Pré-charger les produits populaires au démarrage
3. **Compression:** Compresser les données dans Redis pour économiser la mémoire
4. **Monitoring:** Ajouter Prometheus/Grafana pour surveiller les métriques
5. **Sharding Redis:** Si très grande échelle, sharder les données

---

Q12: "Quand ne faut-il PAS utiliser de cache?"

Réponse (selon README):

1. **High-write workloads:** Si plus d'écritures que de lectures, le cache sera invalidé trop souvent
2. **Petits systèmes:** Complexité supplémentaire inutile
3. **Low reuse:** Si les données ne sont accédées qu'une fois, le cache n'apporte rien

Dans votre projet: Les produits sont lus fréquemment → Cache utile ✅

📝 Points Clés pour la Présentation

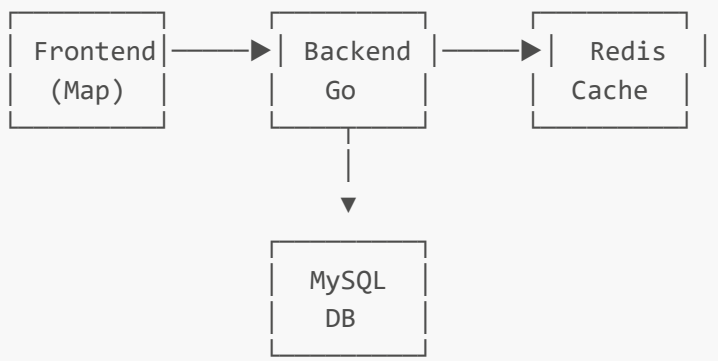
1. Démonstration Live

Scénario à montrer:

1. **Clear Cache** → Montrer que les produits disparaissent
2. **Fetch from Backend** → Montrer Cache MISS (orange) + métriques
3. **Fetch again** → Montrer Cache HIT (vert) + Hit Rate augmente
4. **Add Product** → Montrer Write-Through (ajout immédiat)
5. **Fill cache** → Montrer LRU Eviction quand mémoire pleine

---

2. Diagrammes à Dessiner



---

3. Métriques à Expliquer

- **Hit Rate > 70%:** Cache efficace ✅
- **Miss Rate élevé:** Beaucoup d'accès DB (lent) ⚠
- **Évictions fréquentes:** Cache trop petit, augmenter la taille

---

4. Code à Montrer

Backend - Cache-Aside:

```
// GetProductByID - Ligne 81-106
// Montrer: Check Redis → If MISS → Fetch MySQL → Store Redis
```

Backend - Write-Through:

```
// CreateProduct - Ligne 26-41
// Montrer: Write MySQL → Write Redis
```

Backend - Lock:

```
// UpdateProduct - Ligne 108-139
// Montrer: SetNX lock → Update → Del lock
```

Frontend - LRU:

```
// cacheStore.ts - Ligne 194-219
// Montrer: Find LRU → Evict → Add new
```

🎓 Conclusion

Votre projet démontre:

1. ✅ **Cache-Aside** pour les lectures (efficace, découplé)
2. ✅ **Write-Through** pour les écritures (cohérence forte)
3. ✅ **LRU Eviction** pour la gestion mémoire
4. ✅ **Redis Locks** pour la concurrence
5. ✅ **Métriques en temps réel** pour le monitoring
6. ✅ **Architecture propre** (Clean Architecture)

Points forts:

- Implémentation complète et fonctionnelle
- Code bien structuré
- Dashboard interactif
- Documentation détaillée

Améliorations possibles:

- TTL sur Redis
- Cache warming
- Monitoring avancé
- Tests unitaires

Bonne chance pour votre présentation! 🚀