MAUI > SQLite

Ce document présente les modifications nécessaires pour passer d'un stockage en mémoire à une persistance SQLite dans l'application de gestion de cartes.

> ⚠ ATTENTION : ce document présente une alternative à celle présentée dans le tutorial qui possède des avantages et incovénients...

1. Installation du package SQLite

Pour utiliser SQLite dans une application MAUI, il faut d'abord installer le package NuGet approprié qui fournit les fonctionnalités nécessaires pour interagir avec une base de données SQLite.

```
dotnet add package sqlite-net-pcl
```

2. Le modèle de données avec SQLite

Théorie : Dans SQLite-net-pcl, une propriété nommée `Id` de type `int` est automatiquement configurée comme clé primaire auto-incrémentée. Cela signifie que :

- Vous n'avez pas besoin d'initialiser cette valeur manuellement
- SQLite attribuera automatiquement un identifiant unique à chaque nouvel enregistrement
- Après l'insertion d'un objet, sa propriété Id contiendra la valeur générée par SQLite

Modification du modèle Card :

```
using System;

namespace CardsApp.Models
{
    public class Card
    {
        // Id en int, auto-incrémenté par SQLite
        public int Id { get; set; }

        public string Title { get; set; }
        public string Content { get; set; }
        public DateTime CreatedAt { get; set; } = DateTime.Now;
    }
}
```

3. Le DbContext - Pont entre l'application et la base de données

Théorie : Le DbContext est un concept fondamental dans la persistance des données. Il sert de pont entre votre application et la base de données SQLite en :

- Établissant une connexion à un fichier de base de données SQLite
- Créant les tables nécessaires lors de l'initialisation si elles n'existent pas
- Fournissant des méthodes CRUD qui convertissent vos objets C# en opérations SQL
- Gérant les opérations asynchrones pour éviter de bloquer l'interface utilisateur

Le `SQLiteAsyncConnection` utilisé dans le DbContext permet d'effectuer toutes les opérations de manière asynchrone, ce qui est essentiel pour maintenir la réactivité d'une application mobile.

Implémentation du DbContext :

```
using SQLite;
using System.Collections.Generic;
using System.Threading.Tasks;
using CardsApp.Models;

namespace CardsApp.Data
{
    public class CardDbContext
    {
        private readonly SQLiteAsyncConnection _database;

        public CardDbContext(string databasePath)
        {
            _database = new SQLiteAsyncConnection(databasePath);
            _database.CreateTableAsync<Card>().Wait();
        }

        // Create - Ajoute une nouvelle carte dans la base de données
        public Task<int> AddCardAsync(Card card)
        {
            return _database.InsertAsync(card);
        }

        // Read (All) - Récupère toutes les cartes
        public Task<List<Card>> GetCardsAsync()
        {
            return _database.Table<Card>().ToListAsync();
        }

        // Read (Single) - Récupère une carte spécifique par son Id
        public Task<Card> GetCardAsync(int id)
        {
            return _database.Table<Card>().Where(c => c.Id == id).FirstOrDefaultAsync();
        }

        // Update - Met à jour une carte existante
        public Task<int> UpdateCardAsync(Card card)
        {
            return _database.UpdateAsync(card);
```

```
        }

        // Delete - Supprime une carte par objet
        public Task<int> DeleteCardAsync(Card card)
        {
            return _database.DeleteAsync(card);
        }

        // Delete - Supprime une carte par son Id
        public Task<int> DeleteCardAsync(int id)
        {
            return _database.DeleteAsync<Card>(id);
        }
    }
}
```

4. Adaptation du service pour utiliser SQLite

Théorie : Le service de gestion des cartes sert d'intermédiaire entre le ViewModel et la couche de données. En adaptant le service pour utiliser le DbContext au lieu d'une liste en mémoire, nous modifions l'implémentation interne tout en conservant la même interface publique. Cela nous permet de changer la source de données sans avoir à modifier les ViewModels ou l'interface utilisateur.

Les méthodes du service retournent désormais des résultats de type `Task<T>` car les opérations de base de données sont asynchrones. La valeur de retour `bool` indique si l'opération a réussi ou échoué.

Implémentation du CardService :

```
using System.Collections.Generic;
using System.Threading.Tasks;
using CardsApp.Data;
using CardsApp.Models;

namespace CardsApp.Services
{
    public class CardService
    {
        private readonly CardDbContext _dbContext;

        public CardService(CardDbContext dbContext)
        {
            _dbContext = dbContext;
        }

        public async Task<List<Card>> GetCardsAsync()
        {
            return await _dbContext.GetCardsAsync();
        }

        public async Task<Card> GetCardAsync(int id)
        {
            return await _dbContext.GetCardAsync(id);
        }

        public async Task<bool> AddCardAsync(Card card)
```

```
        {
            int result = await _dbContext.AddCardAsync(card);
            return result > 0;
        }

        public async Task<bool> UpdateCardAsync(Card card)
        {
            int result = await _dbContext.UpdateCardAsync(card);
            return result > 0;
        }

        public async Task<bool> DeleteCardAsync(Card card)
        {
            int result = await _dbContext.DeleteCardAsync(card);
            return result > 0;
        }

        public async Task<bool> DeleteCardAsync(int id)
        {
            int result = await _dbContext.DeleteCardAsync(id);
            return result > 0;
        }
    }
}
```

5. Configuration de la base de données dans l'application

Théorie : Pour que l'application puisse utiliser SQLite, nous devons configurer le chemin de la base de données et enregistrer notre DbContext dans le conteneur de services.

Le chemin `FileSystem.AppDataDirectory` est un emplacement spécifique à la plateforme qui garantit que notre application a les permissions nécessaires pour créer et accéder au fichier de base de données. Cela fonctionne sur toutes les plateformes (Android, iOS, Windows) sans avoir à écrire de code spécifique à chaque plateforme.

L'injection de dépendances assure que le même DbContext et le même service sont utilisés partout dans l'application.

Configuration dans MauiProgram.cs :

```
// Définir le chemin de la base de données SQLite
string dbPath = Path.Combine(FileSystem.AppDataDirectory, "cards.db3");

// Enregistrer CardDbContext
builder.Services.AddSingleton(s => new CardDbContext(dbPath));

// Enregistrer le service de gestion des cartes
builder.Services.AddSingleton<CardService>();
```

Pourquoi le ViewModel ne change pas

Théorie : Le ViewModel n'a pas besoin d'être modifié car il suit le principe de "séparation des préoccupations" du pattern MVVM. Le ViewModel :

- Interagit avec les services via leur interface publique, pas leur implémentation
- Manipule des objets de modèle (Card) indépendamment de la façon dont ils sont stockés
- Expose les mêmes propriétés et commandes à l'interface utilisateur

La seule différence pour l'utilisateur est que les données sont maintenant persistantes entre les sessions de l'application. Les cartes qu'il crée seront disponibles lors de sa prochaine utilisation de l'application.

L'importance des services dans l'évolution de l'application

Théorie : L'utilisation d'un service comme couche intermédiaire entre le ViewModel et les données constitue l'un des plus grands avantages de cette architecture. Ce pattern a permis `d'ajouter SQLite en minimisant les modifications` et permet de manière générale de :

- **Isoler les changements** : L'évolution du stockage de la mémoire RAM vers SQLite n'affecte que le service, sans nécessiter de modifications du ViewModel ou de l'interface utilisateur
- **Faciliter les tests** : Les services peuvent être facilement remplacés par des mocks dans les tests unitaires
- **Permettre des évolutions futures** : Si vous souhaitez plus tard utiliser un service web, Azure, Firebase ou tout autre mécanisme de stockage, vous n'aurez qu'à modifier l'implémentation du service
- **Favoriser la réutilisation** : Le même service peut être injecté dans plusieurs ViewModels différents

Cette approche illustre parfaitement le principe de "dépendance vers l'abstraction" des principes SOLID. En dépendant de l'interface du service (ce qu'il fait) plutôt que de son implémentation (comment il le fait), votre application devient plus flexible et plus facile à maintenir.

Résumé

En résumé, les modifications apportées sont :

1. **Modèle Card** : Changement du type d'Id de string à int pour bénéficier de l'auto-incrémentation
2. **Ajout du DbContext** : Création d'une classe pour gérer les opérations SQLite
3. **Adaptation du Service** : Modification pour utiliser le DbContext au lieu du stockage en mémoire
4. **Configuration** : Ajout du chemin de la base de données et enregistrement du DbContext

Ces changements permettent de transformer l'application de stockage en mémoire en une application avec persistance complète des données, tout en préservant l'architecture MVVM existante. Les données sont désormais conservées entre les redémarrages de l'application, ce qui améliore considérablement l'expérience utilisateur...