MAUI > SQLite avec EF Core

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

1. Installation du package EF Core SQLite

Pour utiliser SQLite avec Entity Framework Core dans une application MAUI, il faut d'abord installer le package NuGet approprié.

```
dotnet add package Microsoft.EntityFrameworkCore.Sqlite
```

2. Le modèle de données avec EF Core

Théorie : Dans Entity Framework Core, une propriété nommée `Id` est conventionnellement reconnue comme clé primaire. Nous pouvons ajouter l'attribut `[Key]` pour être explicite. Avec EF Core, la clé primaire peut être configurée comme auto-incrémentée, ce qui 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;
using System.ComponentModel.DataAnnotations;

namespace CardsApp.Models
{
    public class Card
    {
        [Key]
        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 avec EF Core

Théorie : Le DbContext d'EF Core est une classe qui coordonne la fonctionnalité d'Entity Framework pour un modèle de données. Il :

- Établit une connexion à un fichier de base de données SQLite
- Crée la base de données si elle n'existe pas déjà
- Définit des ensembles d'entités (DbSet) pour chaque type d'entité dans le modèle
- Fournit des méthodes pour interroger et sauvegarder des instances de ces entités
- Gère le suivi des changements et les opérations asynchrones

Implémentation du CardDbContext :

```
using Microsoft.EntityFrameworkCore;
using CardsApp.Models;

namespace CardsApp.Data
{
    public class CardDbContext : DbContext
    {
        private string _databasePath;

        public CardDbContext(string databasePath)
        {
            _databasePath = databasePath;
        }

        public DbSet<Card> Cards { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlite($"Data Source={_databasePath}");
        }
    }
}
```

4. Le service avec instance statique

Théorie : Au lieu d'utiliser l'injection de dépendances (ioc), nous utilisons ici une instance statique pour accéder au service depuis n'importe où dans l'application. Cette approche :

- Simplifie l'accès au service depuis les ViewModels et autres composants
- Évite la configuration du conteneur de services dans MauiProgram.cs
- Utilise `EnsureCreated()` pour initialiser la base de données
- Offre une implémentation simple et directe

Implémentation du CardService avec instance statique :

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

namespace CardsApp.Services
{
    public class CardService
    {
        private readonly string _dbPath;

        // Instance statique directement créée
```

```
public static CardService Instance = new CardService(
    Path.Combine(FileSystem.AppDataDirectory, "cards.db3")
);

public CardService(string dbPath)
{
    _dbPath = dbPath;

    // Assurer que la base de données existe
    using (var context = new CardDbContext(_dbPath))
    {
        context.Database.EnsureCreated();
    }
}

public async Task<List<Card>> GetCardsAsync()
{
    using (var context = new CardDbContext(_dbPath))
    {
        return await context.Cards.ToListAsync();
    }
}

public async Task<Card> GetCardAsync(int id)
{
    using (var context = new CardDbContext(_dbPath))
    {
        return await context.Cards.FindAsync(id);
    }
}

public async Task<bool> AddCardAsync(Card card)
{
    using (var context = new CardDbContext(_dbPath))
    {
        context.Cards.Add(card);
        int result = await context.SaveChangesAsync();
        return result > 0;
    }
}

public async Task<bool> UpdateCardAsync(Card card)
{
    using (var context = new CardDbContext(_dbPath))
    {
        context.Cards.Update(card);
        int result = await context.SaveChangesAsync();
        return result > 0;
    }
}

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

```
        {
            using (var context = new CardDbContext(_dbPath))
            {
                context.Cards.Remove(card);
                int result = await context.SaveChangesAsync();
                return result > 0;
            }
        }

        public async Task<bool> DeleteCardAsync(int id)
        {
            using (var context = new CardDbContext(_dbPath))
            {
                var card = await context.Cards.FindAsync(id);
                if (card == null)
                    return false;

                context.Cards.Remove(card);
                int result = await context.SaveChangesAsync();
                return result > 0;
            }
        }
    }
}
```

5. Utilisation du service dans les ViewModels

Théorie : Avec l'instance statique directe, les ViewModels n'ont plus besoin de recevoir le service via l'injection de dépendances. Ils peuvent simplement accéder à l'instance via la propriété statique `Instance`.

Exemple d'utilisation dans un ViewModel :

```
public class CardListViewModel : BaseViewModel
{
    // Accès au service via le singleton
    private readonly CardService _cardService = CardService.Instance;

    public ObservableCollection<Card> Cards { get; set; } = new ObservableCollection<Card>
();

    public async Task LoadCardsAsync()
    {
        var cards = await _cardService.GetCardsAsync();
        Cards.Clear();
        foreach (var card in cards)
        {
            Cards.Add(card);
        }
    }
```

```
    // Autres méthodes...
}
```

Résumé

En résumé, les modifications apportées sont :

1. **Installation du package** : Passage de sqlite-net-pcl à Microsoft.EntityFrameworkCore.Sqlite
2. **Modèle Card** : Ajout de l'attribut [Key] pour la clé primaire
3. **DbContext** : Création d'une classe DbContext spécifique à EF Core avec DbSet
4. **Service statique** : Implémentation d'une instance statique directe pour le service
5. **Contextes jetables** : Utilisation d'instructions `using` pour chaque opération de base de données

Ces changements permettent de transformer l'application en une solution qui utilise EF Core et un pattern Singleton tout en conservant les mêmes fonctionnalités. Les données sont toujours conservées entre les redémarrages de l'application, maintenant l'expérience utilisateur, mais avec une approche architecturale différente.