Ce document décrit les étapes pour mettre en place et requêter une base de données locale SQLite depuis une application Xamarin Forms (XF). Ce guide se base sur la documentation Microsoft correspondante, il a été réalisé sous Visual Studio (VS) 2019 et prend comme support l'application SIPXami.

Outils requis

Pour utiliser SQLite dans XF, il est nécessaire d'installer le paquet NuGet ***sqlite-net-pcl***. Il existe plusieurs paquets avec le même nom, celui recherché possède les attributs suivants :

- Auteurs: SQLite-net
- Lien NuGet: sqlite-net-pcl

La base de données est enregistrée dans un fichier .db3 dans le système de fichiers propre à l'application (donc protégé) de l'appareil cible. Par conséquent, au moment de la rédaction de ce document, il n'est pas possible de consulter la base de données en direct. On doit donc enregistrer le fichier `.db3` sur un PC disposant d'un explorateur de base de données SQLite.

Pour explorer le système de fichier interne à l'appareil je conseille *Android Studio* et *DB Browser for SQLite* comme explorateur de base de données.

Instanciation de la BDD avec XF

Dans le projet, créer une classe statique (ex: Constantes.cs) qui contient des données communes de configuration de la BDD :

```
public const string DatabaseFilename = "SIPXamiSQLite.db3";

public const SQLite.SQLiteOpenFlags Flags =
    // open the database in read/write mode
    SQLite.SQLiteOpenFlags.ReadWrite |
    // create the database if it doesn't exist
    SQLite.SQLiteOpenFlags.Create |
    // enable multi-threaded database access
    SQLite.SQLiteOpenFlags.SharedCache;

public static string DatabasePath
{
    get
    {
        return Path.Combine(DataDirectoryPath, DatabaseFilename);
    }
}
```

Le fichier de constantes spécifie plusieurs constantes statiques :

- Le nom du fichier de BDD, "`DatabaseFileName`"
-
- "`Flags`" de type *enum* qui sert à initialiser la BDD SQLite.
    - ReadWrite : La connexion peut lire et écrire des données.
    - Create : La connexion crée le fichier de BDD s'il n'existe pas
    - SharedCache : La connexion participera au cache partagé, s'il est activé.

  Dans le cas où l'on souhaite spécifier d'autres *flags* selon la façon dont la base de données sera utilisée, voir la partie "Configurer des constantes d'application" sur la documentation Microsoft.

- "`DatabasePath`" qui correspond au chemin d'accès au fichier. Dans l'exemple ci-dessus, il s'agit de la concaténation du nom du fichier avec `DataDirectoryPath` qui correspond au dossier de stockage des données de l'application (accessible grâce à la constante `FileSystem.AppDataDirectory`.)

Créer la classe d'accès à la BDD

Cette classe va envelopper les fonctions d'accès à la BDD ce qui permet d'abstraire la couche d'accès aux données du reste de l'application. Cette classe centralise la logique de requête et simplifie la gestion de l'initialisation de la base de données. Dans le projet SIPXami, on trouve la classe `LocalDatabase.cs` à cette fin.

Le "Lazy Initialization"

L'initialisation asynchrone *Lazy* permet d'initialiser la base de données au moment du premier accès. De plus cette initialisation ne bloque pas l'exécution et laisse l'opportunité de gérer les exceptions. Ceci est rendu possible grâce à la classe générique `AsyncLazy<T>` :

```
public class AsyncLazy<T>
{
    readonly Lazy<Task<T>> instance;

    public AsyncLazy(Func<T> factory)
    {
        instance = new Lazy<Task<T>>(() => Task.Run(factory));
    }

    public AsyncLazy(Func<Task<T>> factory)
    {
        instance = new Lazy<Task<T>>(() => Task.Run(factory));
    }

    public TaskAwaiter<T> GetAwaiter()
    {
        return instance.Value.GetAwaiter();
    }

    public void Start()
    {
        var unused = instance.Value;
    }
}
```

Pour plus d'informations, voir AsyncLazy.

Création des modèles

Les tables de la BDD SQLite sont créées selon le modèle des objets à manipuler. La bibliothèque SQLite.NET permet de spécifier quelle propriété correspond à la clé primaire grâce à l'attribut `[PrimaryKey]`. Cet attribut est obligatoire pour l'utilisation de certaines fonctions de la bibliothèque qui ont besoin d'une clé primaire sur les objets manipulés.

NB : Lors de l'utilisation de web services (WS), ceux-ci peuvent utiliser des modèles qui leur sont propres. Il est important de créer un nouveau modèle dans le projet, correspondant à chaque modèle des WS pour rester cohérent dans la gestion des données.

Ci-dessous, l'implémentation du modèle *`Mouvement`* correspondant au modèle *`mouvement`* utilisé par le WS *SIPService*. Dans le code présenté, la spécification de l'attribut `[AutoIncrement]` permet de préciser que le champ *`IdMouvement`* est incrémenté à la création d'un nouveau `Mouvement` en BDD.

```
public class Mouvement
{
    [PrimaryKey, AutoIncrement]
    public int IdMouvement { get; set; }
    public DateTime DateMouvement { get; set; }

    // ...

    public int NouvelEffectif { get; set; }

    public Mouvement(mouvement _mouvement)
    {
        IdMouvement = _mouvement.idMouvement;
        DateMouvement = _mouvement.dateMouvement;

        // ...

        NouvelEffectif = _mouvement.nouvelEffectif;
    }

    public Mouvement()
    {

    }
}
```

Initialisation de la BDD

```
public class LocalDatabase
{
    static SQLiteAsyncConnection Database;

    public static readonly AsyncLazy<LocalDatabase> Instance = new
AsyncLazy<LocalDatabase>(async () =>
    {
        var instance = new LocalDatabase();
        CreateTableResult result;
        await Database.CreateTableAsync<Animal>();
        await Database.CreateTableAsync<Bassin>();

        // ...

        await Database.CreateTableAsync<Zone>();
        return instance;
    });

    public LocalDatabase()
    {
        Database = new SQLiteAsyncConnection(Constantes.DatabasePath,
Constantes.Flags);
    }
    // ...
}
```

Le champ Instance est utilisé pour créer les tables de base de données pour les objets `Animal`, `Bassin` et `Zone`, si elles n'existent pas déjà, et renvoie une `LocalDatabase` en tant que singleton. Le champ Instance, de type `AsyncLazy<LocalDatabase>` est construit la première fois qu'il est attendu.

NB : La connexion à la base de données est un champ statique qui garantit qu'une seule connexion à la base de données est utilisée pendant toute la durée de vie de l'application. L'utilisation d'une connexion statique offre de meilleures performances en comparaison à l'ouverture et la fermeture de plusieurs connexions, et cela plusieurs fois au cours d'une exécution de l'application.

Méthodes de manipulation des données

La bibliothèque SQLite.NET fournit un Mapping objet-relationnel (ORM) simple qui permet de stocker et de récupérer des objets sans avoir à écrire d'instructions SQL. Les fonctions disponibles offrent 4 types d'opérations : la création, la lecture, l'édition et la suppression.

```
public async Task<int> SaveOrReplaceZoneAsync(Zone _zone)
{
    await Database.InsertOrReplaceAsync(_zone);
    return 0;
}

public async Task<int> UpdateZoneAsync(Zone _zone)
{
    await Database.UpdateAsync(_zone);
    return 0;
}

public async Task<int> public async Task<int> DeleteZones()
{
    await Database.DeleteAllAsync<Zone>();
    return 0;
}

public async Task<List<Animal>> GetAnimalsByZone(int idZone)
{
    List<Animal> ret = await Database.Table<Animal>().Where(a => a.IdZone ==
idZone).ToListAsync();
    return ret;
}
```

Accès aux données dans XF

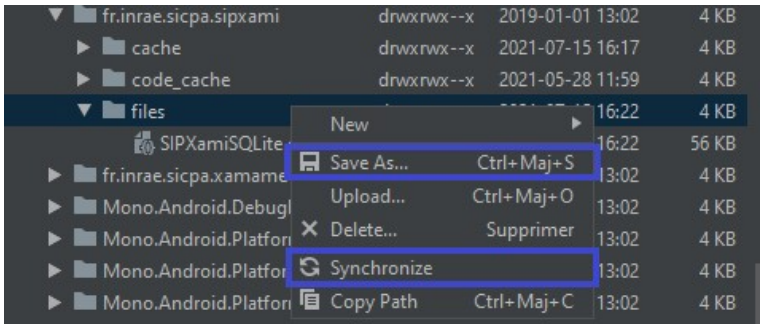
La classe LocalDatabase expose le champ Instance, par lequel les opérations d'accès aux données de cette même classe peuvent être invoquées. Par exemple, pour la mise à jour du champ RFID d'un Bassin, dans la DAO locale :

```
public class LocBassinDAO : IBassinDAO
{
    AsyncLazy<LocalDatabase> instance = LocalDatabase.Instance;

    // ...

    public async Task<int> SetBassinWithTagRFID(bassin bassin)
    {
        LocalDatabase database = await instance;
        int ret = await database.UpdateBassinAsync(bassin);
        return ret;
    }
}
```

Visualiser l'état de la BDD

La visualisation de l'état de la BDD locale ne peut pas être réalisée "en direct". A chaque nouvel état de la BDD il est nécessaire d'effectuer la procédure évoquée dans la partie "Prérequis" : sauvegarder le fichier .db3 sur un PC et le visualiser dans l'explorateur SQLite.
Dans l'explorateur de fichier de l'appareil dans Android Studio, retrouver le chemin d'accès au fichier la base de données et le sauvegarder en effectuant un clic droit dessus puis "Save As…".

Attention : A chaque modification du fichier, l'explorateur Android Studio ne se met pas automatiquement à jour ! Pour avoir la bonne version du fichier, il est nécessaire de "synchroniser" le dossier qui contient ce fichier.

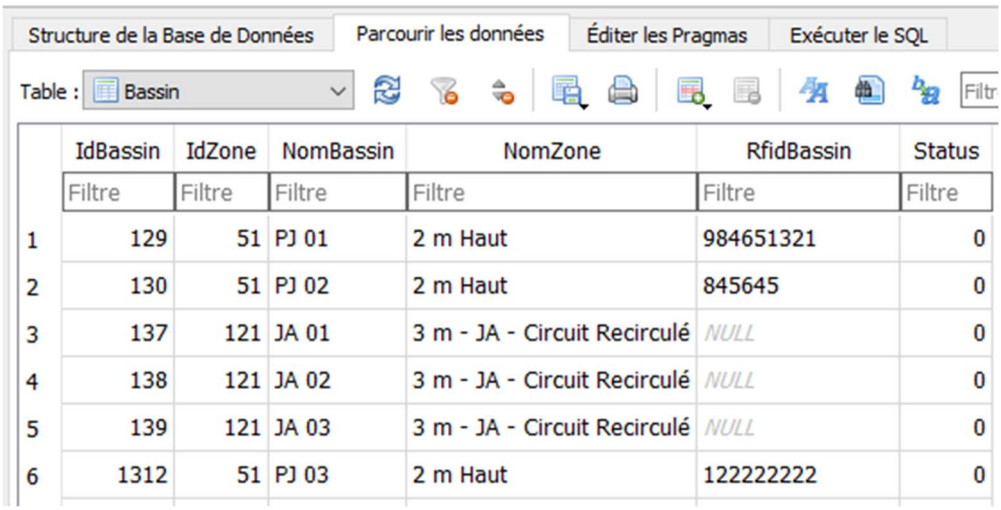


NB : Sauvegarder le fichier dans un chemin simple d'accès (par exemple C:\csharp) pour le retrouver depuis l'explorateur SQLite.

Dans l'explorateur SQLite, cliquer sur "Ouvrir une Base de Données" et retrouver le fichier .db3 dans l'explorateur de fichiers qui s'ouvre. Il est alors possible de consulter l'état de la base de données.

Attention : lorsqu'un fichier *db3* est ouvert dans SQLite, il n'est pas possible de l'écraser depuis Android Studio pour enregistrer une nouvelle version. Il faut alors créer un nouveau fichier avec un nom différent, et l'ouvrir ensuite dans l'explorateur SQLite.

Ci-dessous, un exemple de visualisation des données de type Bassin :



	IdBassin	IdZone	NomBassin	NomZone	RfidBassin	Status
	Filtre	Filtre	Filtre	Filtre	Filtre	Filtre
1	129	51	PJ 01	2 m Haut	984651321	0
2	130	51	PJ 02	2 m Haut	845645	0
3	137	121	JA 01	3 m - JA - Circuit Recirculé	NULL	0
4	138	121	JA 02	3 m - JA - Circuit Recirculé	NULL	0
5	139	121	JA 03	3 m - JA - Circuit Recirculé	NULL	0
6	1312	51	PJ 03	2 m Haut	122222222	0