Lab 02: Mobile Data using SQLite-NET

Prerequisites

You will need a Windows PC with the Android SDK and Xamarin tools installed. We will be using the Android emulator to test the code we are building, so make sure to have a virtual device already configured and ready to run. See the Xamarin.Android setup documentation if you need help getting your environment setup.

Downloads

Links to lab sample code (if available).

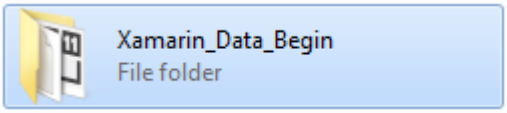
Lab Goals

The goal of this lab will be to introduce integrating local data storage on mobile devices using the SQLite-NET ORM.

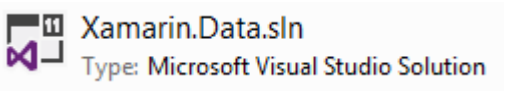
Steps

Open the Starting Solution

1. Launch **Xamarin Studio**
2. Click **Open...** on the Xamarin Studio Welcome and navigate to the **Lab 02 Resources** folder included with this document.
3. Locate the **Xamarin_Data_Begin** folder – make sure it's the **begin** and not the **completed** folder.



4. Inside the **Xamarin_Data_Begin** folder, you will find a **Xamarin.Data.sln** file – double click on this file to open the starter solution

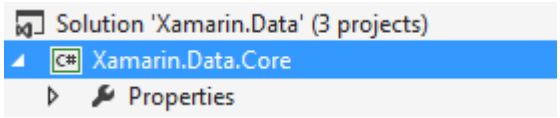


5. Go ahead and build and run the application in the emulator to make sure it compiles and your environment is ready. Let the instructor know if you have any trouble.

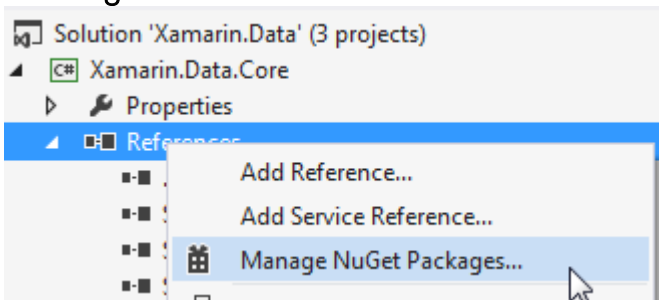
Installing SQLite.NET

We are going to reference the SQLite.Net ORM library to help streamline development for local data storage

1. In Xamarin Studio/Visual Studio, locate the project named Xamarin.Data.Core

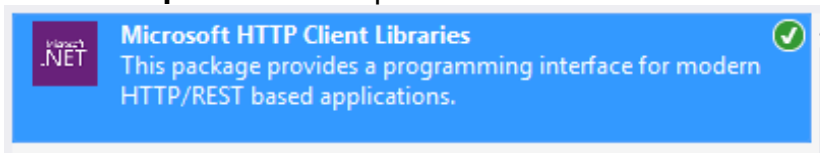


2. Right click on the components folder and select Manage NuGet Packages…



3. In the NuGet Package Manager Dialog, search for SQLite-NET and install SQLite.Net.Async PCL. Since SQLite.Net PCL is a dependency, it will be added as well.

4. In order to send HTTP requests and handle the responses that come back within a Portable Class Library, we'll also need an additional package. While still in the NuGet, search for **Microsoft.Net.Http** and install **Microsoft HTTP Client Libraries** after agreeing to the licenses. You will also get **Microsoft BCL Portability Pack** and **Microsoft BCL Build Components** as dependencies.



5. We will also need to install some SQLite.Net packages for our platform-specific project. Right-click on **Xamarin.Data.Droid** and select **Manage NuGet Packages…**
6. Search for **SQLite.Net.Platform.XamarinAndroid** and install **SQLite.NET PCL - XamarinAndroid Platform**.

7. Also install the **SQLite.Net.Async PCL** package as we did above for **Core** by searching for and installing SQLite.Net.Async-PCL.

Getting started with SQLite.Net

With the packages installed, we can start setting up our core project to allow us to inject platform-specific implementations.

1. Locate `//TODO: Step 1a - using SQLite in ConferenceDatabaseAsync` in **ConferenceDatabaseAsync.cs** and uncomment the SQLite `using` statement.

```
using SQLite.Net.Async;
```

2. Below there is `//TODO: Step 1b - Inherit from SQLiteAsyncConnection`. Either comment out or delete the original **ConferenceDatabaseAsync** constructor and add one inheriting from **SQLiteAsyncConnection**.

```
public class ConferenceDatabaseAsync : SQLiteAsyncConnection
```

3. To allow injecting a platform-specific connection into our database class, locate `//TODO: Step 1c - Allow injection of function to get platform-specific connection` and add the constructor accepting a function that returns a SQLite connection.

```
public ConferenceDatabaseAsync(Func<SQLite.Net.SQLiteConnectionWithLock> connection) : base(connection) { }
```

4. To avoid using compiler directives, we will be implementing an interface on any platforms. Locate `//TODO: Step 2a - Use SQLite in IPlatform` in IPlatform.cs and uncomment the SQLite using statement.

```
using SQLite.Net.Interop;
```

5. Let's put the final property on the interface. Locate `//TODO: Step 2b - Add ISQLitePlatform to IPlatform` and add the SQLitePlatform property.

```
ISQLitePlatform SQLitePlatform { get; }
```

SQLite.Net table creation

Our model classes will instruct SQLite on the creation of our underlying tables and any migrations needed through property attributes.

1. Navigate to the Model\Session.cs class by locating `//TODO: Step 3a - Using SQLite in Session model`. Uncomment the SQLite using statement.

```
using SQLite.Net.Attributes;
```

2. Add the following code under `//TODO: Step 3b - Add a primary key in Session model` and add in attributes to define a primary key

```
[SQLite.PrimaryKey, SQLite.AutoIncrement, SQLite.Column("_id")]
```

3. Add the following code under `//TODO: Step 3c - Mark column that you will not persist`

```
[SQLite.Ignore()]
```

> **Tip:** SQLite-NET has a lot more features than we have seen here. Please reference the SQLite-NET wiki for more information.

4. Switch to the Speaker.cs model at `//TODO: Step 4a - Using SQLite in Speaker model`. Uncomment the SQLite using statement.

```
using SQLite.Net.Attributes;
```

5. Add the following code under `//TODO: Step 4b - Add a primary key in Speaker model`.

```
[PrimaryKey, AutoIncrement]
```

6. Navigate to `//TODO: Step 5 - Create tables` and add in the following code. This code will create any tables that may not be there when we connect to the database

```
await CreateTableAsync<Session>().ConfigureAwait(false);
await CreateTableAsync<Speaker>().ConfigureAwait(false);
```

> There are a number of return values present at the upcoming tasks. These were put in place just to ensure the solution would compile when first loaded. We will be removing them by either commenting them out or deleting them.

7. Move down to `//TODO: Step 6a - Remove placeholder return`. Comment out or delete the placeholder return value.
8. Under `//TODO: Step 6b - Select All Records from Table` add the following code to query for all data in the table.

```
return await Table<Session>().ToListAsync().ConfigureAwait(false);
```

9. Move down to `//TODO: Step 7a - Remove placeholder return`. Comment out or delete the placeholder return value.
10. Under `//TODO: Step 7b - Select records from table using Linq` add in the following code to query for the first record and return the first match

```
return await Table<Session>().Where(s => s.Id == id).FirstOrDefaultAsy
nc().ConfigureAwait(false);
```

11. Move down to `//TODO: Step 8a - Remove placeholder return`. Comment out or delete the placeholder return value.
12. Under `//TODO: Step 8b - Perform an insert or update to the database` add in the following code to perform an insert or update

```
if (item.Id <= 0)
    return await InsertAsync(item).ConfigureAwait(false);

await UpdateAsync(item).ConfigureAwait(false);
return item.Id;
```

13. Under `//TODO: Step 8c - Perform a bulk insert or update to the database`, add the following code to insert a collection in one call.

```
await InsertAllAsync(sessions).ConfigureAwait(false);
```

14. Move down to `//TODO: Step 9a - Remove placeholder return`. Comment out or delete the placeholder return value.
15. Navigate to `//TODO: Step 9b - Delete a record from the table` and in the following code to perform a deletion

```
return await DeleteAsync(session).ConfigureAwait(false);
```

16. Navigate to `//TODO: Step 10 - Delete all data using SQL commands` and add in the following code to delete all data from a table

```
await this.ExecuteAsync("DELETE FROM Session;").ConfigureAwait(false);
```

Tip: You can still call standard SQL code when using the SQLite-NET code.

17. Save and Build the project and ensure that there are no errors

Platform-Specific Implementation

Since we are injecting our platform details into our SQLite core code, let's start putting the implementation together.

1. In the **Xamarin.Data.Droid** project, open the **Platform.cs** file
2. Navigate to `//TODO: Step 11a - Android - Using SQLite in Platform` and uncomment the SQLite using statement.

```
using SQLite.Net.Interop;
```

3. Below that, after `//TODO: Step 11b - Android - Implement the platform configurations`, add in the following code to provide an implementation for Android

```
public class Platform : IPlatform
{
    const string SQLiteFilename = "StockDB.db3";

    public Stream SessionsPath
    {
        get
        {
            return
                File.Open(
                    Path.Combine(
                        System.Environment.GetFolderPath(System.Enviro
nment.SpecialFolder.MyDocuments),
                        "Sessions.xml")
                    , FileMode.OpenOrCreate, FileAccess.ReadWrite);
        }
    }

    public string DatabasePath
```

```
    {
        get
        {
            return Path.Combine(
              System.Environment.GetFolderPath(System.Environment.Spec
ialFolder.Personal),
              SQLiteFilename);
        }
    }

    public ISQLitePlatform SQLitePlatform
    {
        get { return new SQLite.Net.Platform.XamarinAndroid.SQLitePlat
formAndroid(); }
    }
}
```

4. Open the XamarinDataApp.cs file
5. Navigate to `//TODO: Step 12a - Android - Remove placeholder return` and, inside the static DataManager property, remove the temporary return statement;
6. Then, under `//TODO: Step 12b - Android - Provide an instance of DataManager`, provide an instance of the DataManager for our Android application

```
return _DataManager ?? (_DataManager = new DataManager(new Platform())
);
```

7. Now we want to make a ConferenceDatabaseAsync instance available from our DataManager class. In the Core project, return to DataManager.cs by locating `//TODO: Step 13 - Offer a ConferenceDatabaseAsync through DataManager`. Add the ConferenceDatabase property below it.

```
public ConferenceDatabaseAsync ConferenceDatabase
{
    get
    {
        return _conferenceDatabase ??
        (
            _conferenceDatabase = new ConferenceDatabaseAsync(() =>
                new SQLite.Net.SQLiteConnectionWithLock(
                    _platform.SQLitePlatform,
                    new SQLite.Net.SQLiteConnectionString(_platform.Da
tabasePath, true)))
        );
    }
}
```

Creating our App

With all the database back-end work out of the way, let's put everything into use in an Android Activity.

1. Open the Activities/SessionsActivity.cs file
2. Navigate to `// TODO: Step 14 - Android - Prepare to load` and add the following code to clear and insert data into the sessions adapter

```
await XamarinDataApp.DataManager.ConferenceDatabase.SetupDatabaseAsync
();

LoadDataAsync();
```

3. Navigate to `//TODO: Step 15 - Android - Load data from database` and add the following code to query the database and load data into our adapter

```
var sessions = await XamarinDataApp.DataManager.ConferenceDatabase.Get
SessionsAsync();


adapter.Sessions.Clear();

adapter.Sessions.AddRange(sessions);

adapter.NotifyDataSetChanged();
```

4. Navigate to `//TODO: Step 16 - Android - Clear database and save new sessions` and add in the following code to clear the database and save new sessions

```
// Download all the data

await XamarinDataApp.DataManager.DownloadSessionXmlAsync();


var sessions = await XamarinDataApp.DataManager.GetSessionsAsync();


//Delete data that exists and re-import it into our database.

await XamarinDataApp.DataManager.ConferenceDatabase.DeleteSessionsAsyn
c();

await XamarinDataApp.DataManager.ConferenceDatabase.SaveSessionsAsync(
sessions);


await LoadLocalAsync();
```

5. Finally, to actually retrieve our data so it can be incorporated into our database, we need to put the Microsoft HTTP Client to work. In the Core project, navigate to `//TODO: Step 17a - Http using in Downloader` and uncomment the BCL Net `using` statement.

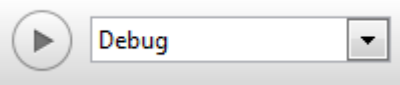
```
using System.Net.Http;
```

6. Below `//TODO: Step 17b - Download sessions`, add the code to retrieve our session XML data.

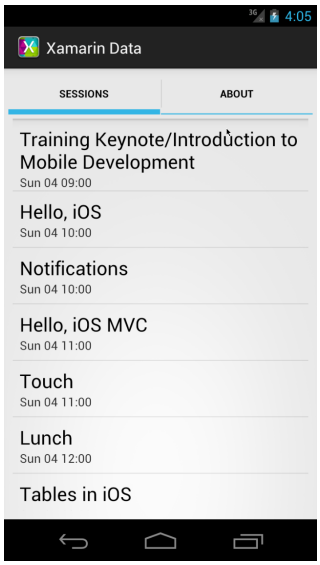
```
using (var httpClient = new HttpClient())
    await streamWriter.WriteAsync(await httpClient.GetStringAsync(sess
ionsXmlUrl)).ConfigureAwait(false);
```

Testing the Data App

1. Ensure your Android Emulator is running and from Xamarin Studio, Debug or Run the app



2. The application will load data from the database and present the results into the list view



Summary

In this lab, we learned how to integrate a SQLite database to assist us with persisting data locally in our applications. Using the SQLite-NET ORM, we generated our tables, created a client, queried for and took action against data in our tables.