Lab 01: Mobile Data using ADO.NET

Prerequisites

You will need a development environment, either a Mac or Windows PC with the Android SDK and Xamarin tools installed.

The lab instructions use Android as the example, but the same steps can be performed in the included iOS project if you prefer to work in that environment. Before starting the exercise, make sure your environment is properly setup and capable of building your target application style (either Android or iOS).

Downloads

The starter project and completed example are both included with this lab manual and can be downloaded from http://university.xamarin.com.

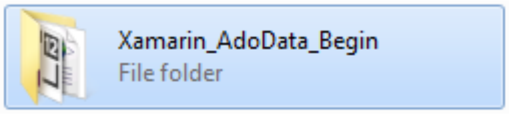
Lab Goals

The goal of this lab will be to introduce integrating local data storage on mobile devices using ADO.NET. We will demonstrate this by implementing some persistent storage in a stock viewer application.

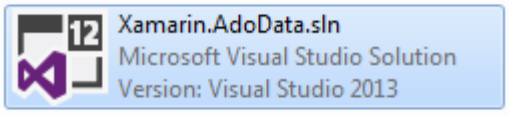
Steps

Open the Starting Solution

1. Launch **Xamarin Studio** or **Visual Studio** (Windows).
2. Click **Open…** on the Welcome Screen and navigate to the **Lab 01 Resources** folder included with this document.
3. Locate the **Xamarin_AdoData_Begin** folder and navigate into the folder (make sure it's the **begin** and not the **completed** folder).


Xamarin_AdoData_Begin
File folder

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


Xamarin.AdoData.sln
Microsoft Visual Studio Solution
Version: Visual Studio 2013

5. Go ahead and build the solution and make sure it compiles and can run in the Android emulator or iOS simulator.

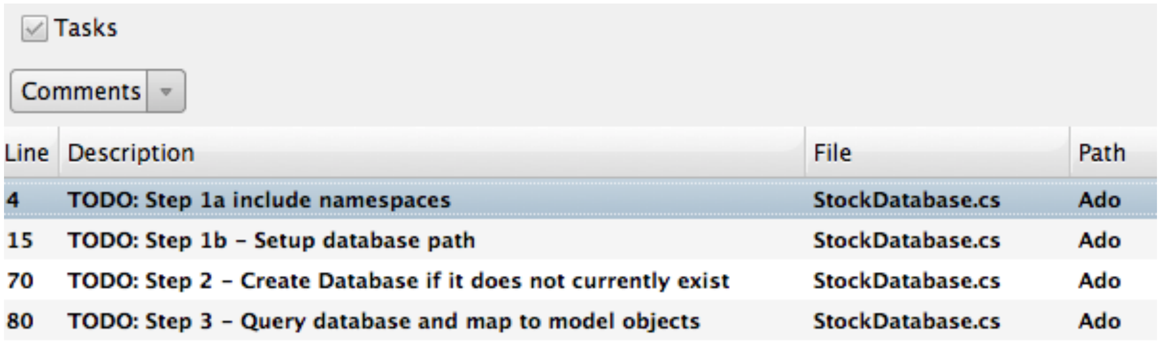
Examine the Beginning Solution

This lab exercise will use a Stock metaphor to demonstrate ADO.NET data access. A simple data object with an Id and Symbol represents the stock itself. The application then randomly generates stocks and adds them to a local SQLite database using the ADO.NET library. Each time you launch the application it will reload these saved stocks and display them. You can also insert additional stocks using a refresh button (Android) or gesture (iOS).

1. The beginning solution has three projects in it
    a. Xamarin.AdoData.Core – a library project with all the shared files used with both the iOS and Android versions of the exercise. This is where most of our data-access code will reside. The solution is using file linking to share the source files, although it could also be structured using a Portable Class Library with a little more work.
    b. Xamarin.AdoData.Droid – the Android version of the UI.
    c. Xamarin.AdoData.iOS – the iOS version of the UI.

Examine the three projects before you start so you have a general understanding of the pieces of the application. You will primarily be uncommenting code in order to add the database support.

The code is marked with `TODO` comments, which can be easily located using the Tasks pane in Xamarin Studio or Visual Studio – double clicking on a comment will navigate you to the appropriate location in the code. For example, here is the tasks pane in Xamarin Studio:

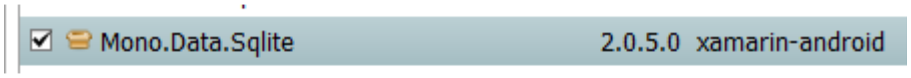


Line	Description	File	Path
4	TODO: Step 1a include namespaces	StockDatabase.cs	Ado
15	TODO: Step 1b – Setup database path	StockDatabase.cs	Ado
70	TODO: Step 2 – Create Database if it does not currently exist	StockDatabase.cs	Ado
80	TODO: Step 3 – Query database and map to model objects	StockDatabase.cs	Ado

Setting up ADO.NET for Mobile Data

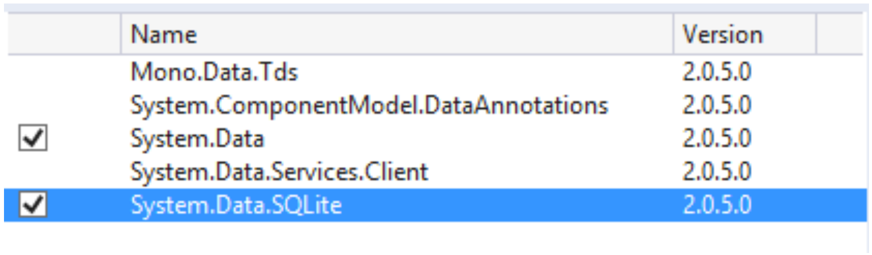
The first step in using ADO.NET in a mobile application is to include the proper references.

1. Right-Click on the Xamarin.AdoData.Core project and select Edit References… (Or Add Reference… in Visual Studio).
2. Add Mono.Data.Sqlite and System.Data as references



☑ System.Data 2.0.5.0 xamarin-android

If you are using Visual Studio, the assemblies are called `System.Data` and `System.Data.SQLite`:



	Name	Version
	Mono.Data.Tds	2.0.5.0
	System.ComponentModel.DataAnnotations	2.0.5.0
✓	System.Data	2.0.5.0
	System.Data.Services.Client	2.0.5.0
✓	System.Data.SQLite	2.0.5.0

3. Repeat steps 1-2 for the `Xamarin.AdoData.Droid` and `Xamarin.AdoData.iOS` projects so they will continue to compile properly when we begin uncommenting and adding code. All three projects should be referencing these two assemblies.

Writing your Data Layer

Next, you will want to abstract your data access code into a shared library or file so that you can reuse it across both Android and iOS (and potentially even Windows or Windows Phone).

1. Open the Ado\StockDatabase.cs file. We will be uncommenting several blocks in this file. First, at the top, locate the comment `TODO: Step 1a - include namespaces` and uncomment the `using` statement below it.

```
// TODO: Step 1a include namespaces
using Mono.Data.Sqlite;
```

2. Locate the comment `TODO: Step 1b - Setup database path` and uncomment the block of code that follows it to generate the path and filename for our database file, create a connection, and execute a SQL textual command. It is a large block so feel free to examine the code – it's pretty standard ADO.NET logic.

```
//TODO: Step 1b - Setup database path
private static string _databaseFilePath;

public static string DatabaseFilePath
{
    get
    {
        if (!string.IsNullOrEmpty(_databaseFilePath))
            return _databaseFilePath;

        const string sqliteFilename = "StockDB.db3";

        // Get the documents folder.
        string path = Environment.GetFolderPath(Environment.SpecialFolder.Pers
onal);
#if __IOS__
        // In iOS we want to use the library path so this file is not
        // backed up to iCloud.
        path = Path.Combine(path, "..", "Library"); // Library folder
#endif

        _databaseFilePath = Path.Combine(path, sqliteFilename);

        return _databaseFilePath;
    }
}

private static SqliteConnection CreateConnection()
{
    return new SqliteConnection("Data Source=" + DatabaseFilePath);
}

private async Task<int> ExecuteNonQueryAsync(string commandText)
{
    var updates = 0;
    using (await Mutex.LockAsync().ConfigureAwait(false))
    {
        using (var connection = CreateConnection())
        {
            connection.Open();
            using (var transaction = connection.BeginTransaction())
            {
                using (var c = connection.CreateCommand())
                {
                    c.CommandText = commandText;
                    updates = await c.ExecuteNonQueryAsync().ConfigureAwait(fa
lse);
                }
                transaction.Commit();
            }
            connection.Close();
        }
    }

    return updates;
}
```

Tip: If your application has multiple threads accessing your database, make sure to lock access or restrict multiple connections. In the newly uncommented code shown above we are using a `Mutex` for this purpose.

3. Locate the comment `//TODO: Step 2 - Create Database if it does not currently exist`

```
//TODO: Step 2 - Create Database if it does not currently exist
public async Task<bool> CreateDatabaseIfNotExistsAsync()
{
    if (File.Exists(DatabaseFilePath))
        return true;
    return await ExecuteNonQueryAsync(
               "CREATE TABLE [Stock] (_id ntext, Symbol ntext);")
                  .ConfigureAwait(false) > 0;
}
```

4. Next, locate the comment `TODO: Step 3 - Query database and map model objects` and uncomment the method that follows. This will use ADO.NET to execute a `SELECT` query and then create a set of `Stock` objects from the Model folder.

```
// TODO: Step 3 - Query database and map to model objects
public async Task<IList<Stock>> SelectStockAsync()
{
    var stockInDatabase = new List<Stock>();

    using (await Mutex.LockAsync().ConfigureAwait(false))
    {
        using (var connection = CreateConnection())
        {
            connection.Open();

            using (var contents = connection.CreateCommand())
            {
                contents.CommandText = "SELECT [_id], [Symbol] from [Stock]";
                using (var reader = await contents.ExecuteReaderAsync()
                                                  .ConfigureAwait(false))
                {
                    while (await reader.ReadAsync().ConfigureAwait(false)) {
                        stockInDatabase.Add(
                            new Stock {
                                Id = Convert.ToInt32(reader["_id"]),
                                Symbol = reader["Symbol"].ToString()
                            });
                    }

                    reader.Close();
                }
            }
            connection.Close();
        }
    }
    return stockInDatabase;
}
```

Architecture Tip: Try to create your database methods so that they receive and return model objects or primitive types. This way, your UI is further separated from any database implementation.

5. Locate the comment `TODO: Step 4 - Perform an insert` and uncomment the method that follows which will create a new row in the table from a `Stock` model object:

```
//TODO: Step 4 - Perform an Insert
public async Task<bool> InsertStockAsync(Stock stock)
```

```
{
    var updates = 0;
    using (await Mutex.LockAsync().ConfigureAwait(false))
    {
        using (var connection = CreateConnection())
        {
            connection.Open();
            using (var transaction = connection.BeginTransaction())
            {
                using (var c = connection.CreateCommand())
                {
                    c.CommandText = "INSERT INTO Stock (_id, Symbol) values (@
id, @symbol);";
                    c.Parameters.Add ("@id", System.Data.DbType.String).Value
= stock.Id;
                    c.Parameters.Add ("@symbol", System.Data.DbType.String).Va
lue = stock.Symbol;

                    updates = await c.ExecuteNonQueryAsync().ConfigureAwait(fa
lse);
                }
                transaction.Commit();
            }
            connection.Close();
        }
    }

    return updates > 0;
}
```

Notice that we are using *parameterized queries* here to avoid a SQL injection attack. In this instance (since we control all the code paths and aren't using user input), it's not as likely an issue, but it's always a good practice to use this style of code for inserts or updates to a database.

That completes the database class we will be using. Now we will be adding support into our Android project to use this class. Our platform project will not need to use

Adding Data Access code to your project

Finally, we will need to utilize the data access code in our Android and iOS application.

Note: from this point forward, the instructions refer to the Android version of the lab, you can use the same instructions for the iOS version as well, and the same comments and TODO markers are in that project.

1. Locate the comment `TODO: Step 5 - Android - Integrate the ADO.Net client into your UI` and uncomment the method that follows it which will load our stocks from the database.

```
//TODO: Step 5 - Android - Integrate the ADO.Net client into your UI
private async Task LoadStockDataAsync()
{
    try
    {
        if (loadDataMenuItem != null)
            loadDataMenuItem.SetEnabled(false);

        var adoDatabase = new Core.Ado.StockDatabase();

        //Generate some sample stock items
        for (var i = 0; i < 5; i++)
            await adoDatabase.InsertStockAsync(Core.StockHelper
                                 .GenerateStock() /* Generates a fake stock */);

        var stocks = await adoDatabase.SelectStockAsync();

        adapter.Stocks.AddRange(stocks);
        adapter.NotifyDataSetChanged();
    }
    finally
    {
        if (loadDataMenuItem != null)
            loadDataMenuItem.SetEnabled(true);
    }
}
```

Note: we are using the async/await keywords here to keep the UI thread responsive – this is always a good practice in mobile applications when dealing with I/O or other blocking operations that may take an indeterminate amount of time to execute.

2. Locate the comment `TODO: Step 6a - Android - Setup the database and query` in the `OnResume` method and uncomment the lines of code that follow – this will initialize our database and kick off the load.

```
// TODO: Step 6a - Android - Setup the database and query
var adoDatabase = new Core.Ado.StockDatabase();
await adoDatabase.CreateDatabaseIfNotExistsAsync();
await LoadStockDataAsync ();
```

3. Locate the comment `TODO: Step 6b - Android - query database` in the `OnMenuItemSelected` method and uncomment the line to refresh our UI from the database.

```
case Resource.Id.action_refresh:
    //TODO: Step 6b - Android - query database
    LoadStockDataAsync();
    break;
```

4. Run the application (whichever version you worked on) – you should see it populate with some sample data. Each time you press the Refresh button (or pull down to refresh on the iOS version), it will add 5 more records.

5. Stop the application and terminate it using the task list (double-tap in iOS, or use the Recent Applications list in Android) to make sure it is physically removed from memory.
6. Relaunch the application – your existing data should still be present.
7. Try adding some breakpoints to walk through the system loading the data.

Summary

In this lab, we learned how to integrate the ADO.Net components into our Mobile applications and perform simple SQL queries to create a database, insert and retrieve data.

There is a completed version of the solution in the `Xamarin_AdoData_Completed` folder if you would like to run or examine the final code.