Data Access

Xamarin Level 2, Chapter 3

Overview

Many applications will have the requirement to save data on the device locally. This will require a database and a data layer in the application to manage the database. For performance, flexibility, and cross-platform compatibility developers are strongly encouraged to use SQLite. Support is available on the following mobile platforms:

- **iOS** – Built in to the operating system.
- **Android** – Built in to the operating system since Android 2.2 (API Level 10).
- **Windows Phone** – Can support SQLite by shipping the open-source C# SQLite assembly with the application. More information can be found at C# SQLite on Google Code.

Even with the database engine available on all platforms, the native methods to access the database are different. Android has it's own built-in API's to access an SQLite database. Xamarin.Android does provide bindings for these Android specific API's, but if you use them in an application it makes it much harder to write cross platform data access code. Instead, there are two other API's that can be used to increase code re-use:

- **ADO.NET** – Xamarin.Android comes out of the box with an ADO.NET Data Provider for SQLite.
- **SQLite-NET** – This is a cross platform *Object-Relational Mapper* (ORM) targeting SQLIte.

This chapter will briefly discuss using ADO.NET for managing and accessing an SQLite database in an Android application. It will then move on to SQLite-NET, and explain how to add SQLite-NET to a Xamarin.Android application, how to create tables based on data model classes, and then how to perform standard CRUD operation.

ADO.NET

Xamarin.Android has built in support for SQLite via ADO.NET. By referencing the namespaces `System.Data` and `Mono.Data.Sqlite`. These namespaces contain the classes that will allow an application manage and access SQLite databases. Edit the project references to include both `System.Data.DLL` and `Mono.Data.Sqlite.DLL` and add these using statements to your code:

```
using System.Data;
using Mono.Data.Sqlite;
```

The ADO.NET Data Provider for SQLite is very much like any other data provider and if you already understand how to use ADO.NET then you will.

The connection string used for SQLite is:

```
var connectionString = "Data Source=<Path-database-file>"
```

The following code is an example of how create an SQLite database with a table, insert a row, and then read records from the table:

```
string dbPath = Path.Combine (
        Environment.GetFolderPath (Environment.SpecialFolder.Personal),
        "items.db3");
bool exists = File.Exists (dbPath);
if (!exists) {
    // Need to create the database and seed it with some data.
    SqliteConnection.CreateFile (dbPath);
    var connection = new SqliteConnection ("Data Source=" + dbPath);
   connection.Open ();
   var commands = new[] {
        "CREATE TABLE [Items] (Key ntext, Value ntext);",
        "INSERT INTO [Items] ([Key], [Value]) VALUES ('sample', 'text')"
    };
    foreach (var command in commands) {
        using (var c = connection.CreateCommand ()) {
            c.CommandText = command;
            c.ExecuteNonQuery ();
        }
    }
}

// use `connection`... here, we'll just append the contents to a TextView
using (var contents = connection.CreateCommand ()) {
    contents.CommandText = "SELECT [Key], [Value] from [Items]";
    var r = contents.ExecuteReader ();
    while (r.Read ())
        Console.Write("\n\tKey={0}; Value={1}",
                r ["Key"].ToString (),
                r ["Value"].ToString ());
}
connection.Close ();
```

Real world implementations of ADO.NET would obviously be split across different methods classes. This example is for demonstration purposes only.

SQLite-ORM

An ORM attempts to simplify storage the storage of data modeled in classes. Rather than manually writing the SQL statements to CREATE TABLEs, or to SELECT / DELETE / INSERT / UPDATE data, an ORM is an architecture layer that does this for you. An ORM will examine the structure of the classes in the data model of an application, and then create tables and columns to mirror the classes. This allows applications to send and retrieve object instances from the ORM, which takes care of all of the SQL operations under the hood.

SQLite-NET is one such example of an ORM. It is an open source, minimal library that is easy to integrate in a C# project. Some of the features of SQLite-NET:

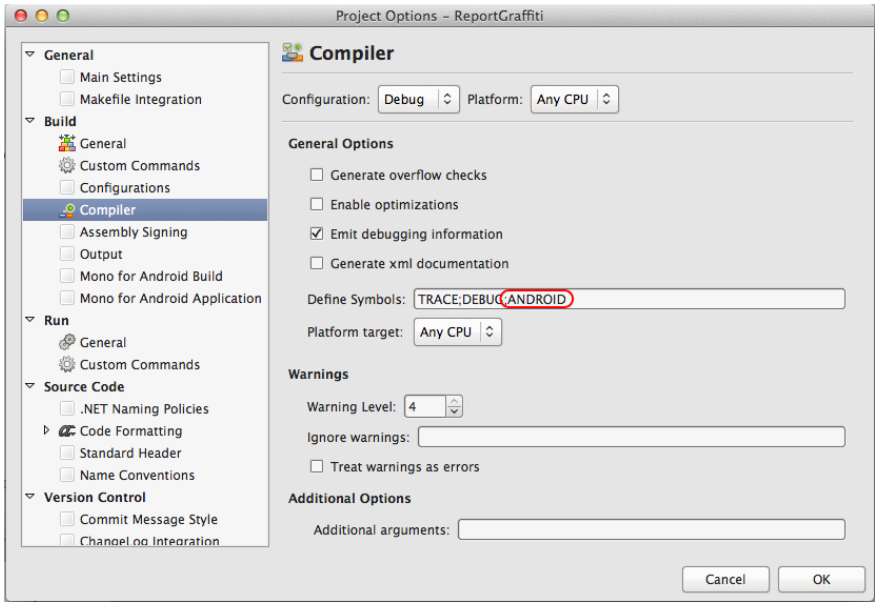
- ➔ Tables are defined by adding attributes to data model classes.

- ➔ A database instance is represented by a subclass of `SQLiteConnection`, the main class in the SQLite-NET library.
- ➔ Data can be inserted, queried, and deleted using objects. No SQL statements are required.
- ➔ Basic LINQ queries can be performed on the collections returned by SQLite-NET.
- ➔ Automatic migrations that will add new columns to tables when new properties are added to the Model classes.

The source code and documentation for SQLite-NET is available on SQLite-NET on GitHub.

Getting Started

Getting started with SQLite-Net involves downloading the file SQLite.cs from GitHub and adding it to your Xamarin.Android project. Next define an `ANDROID` compiler directive as shown in the following screenshot:



Once this is done, the code in the file `SQLite.cs` has been integrated with your application and is ready for use.

Creating a Model

Each table in an SQLite database has a corresponding class that matches a table in an SQLite database. Each instance of the data model class corresponds to a row in the table. A simple example of a data model class is shown below:

```
[Table("stocks")]
public class Stock
{
    [PrimaryKey, AutoIncrement, Column("_id")]
    public int Id { get; set; }
```

```
    [MaxLength(8)]
    public string Symbol { get; set; }
}
```

The attributes that adorn the properties declare meta-data to help SQLite-NET with creating the database schema. So, in the example above, one instance of a Stock object will correspond to one row in an SQLite named `stocks`. This table has two columns:

- **_id** – This is the primary key to the table, and is an auto-incrementing integer column. It will map to the property `Id`.
- **Symbol** – This is a string column.

There are several more attributes that can be used to mark up a data model class in SQLite. The best source of documentation is to read the source code in the file SQLite.cs that was added to the project.

Once the model class is created, SQLite-NET can automatically generate the database and the tables within by calling `CreateTable` as shown in the following code snippet:

```
var db = new SQLiteConnection(connectionString);
db.CreateTable<Stock>();
```

Each time `CreateTable<T>` is called, SQLite-NET will add any new columns to the database table that have been added to the data model class. SQLite-NET will not perform any other schema changes such as deleting columns, renaming columns, or changing datatypes.

Create, Read, Update, and Delete

SQLite-NET has an intuitive and easy to use API that allows you to quickly interact with tables in a database. Using the Stock data model class from the previous section, here is how to perform some common database tasks. For these examples, assume that an instance of `SQLiteConnection` has already been defined:

```
var = new SQLiteConnnection(pathToDatabase);
```

- **Create** – To insert a new record into the stocks table, use the Insert method:

```
var newStock = new Stock();
newStock.Name = "APPL";
db.Insert (newStock);
```

When the new row is inserted into the database, `newStock.Id` will be updated with the value of the primary key that the database created.

- **Read** – To retrieve a specific instance of Stock from the database, `SQLIteConnection` has a `GetItem<T>` method for example:

```
var existingItem = db.GetItem<Stock>(11);
```

The most straightforward way to query for data is to use the `Table` method, as shown in the following example:

```
var stocksStartWithB = db.Table<Stock>.Where(stock =>
stock.Symbol.StartsWith("B"));
```

If it is necessary to use SQL, perhaps for performance reasons, then `SQLiteConnection` provides a `Query` method that can be use:

```
var stocksStartingWithA = db.Query<Stock>("SELECT * FROM Stock  WHERE
Symbol = ?", "A");
```

➔ **Update** – To update an existing stock record in the Stock table:

```
db.Update (existingStock); // Stock.Id must be populated for Update to
work
```

➔ **Delete** - Finally to delete a record from the database, there is a Delete method:

```
db.Delete<Stock>(existingInstance);
```

For more information about SQLite-NET, consult the source code or the SQLite-NET wiki page.

Summary

This chapter discussed data access in a Xamarin.Android application using SQLite as a database. The open source database SQLite-NET was introduced. Then we went on to briefly talk about how to use ADO.NET to create and manipulate data in an SQLite database. As an alternative to ADO.NET, an Object-Relational Mapper named SQLite-NET was discussed. We saw how to add SQLite-NET to a Xamarin.Android application, and then some examples were given on how to use SQLite-NET to create a database and perform basic CRUD actions.