Data

Evolve Fundamentals Track, Chapter 10

Overview

Data storage is an important part of any application's architecture – an application might store anything from user preferences, documents and photos to datasets downloaded from web services.

If the amount of data is small then it might be stored as simple key-value pairs (which is supported on iOS and Android), or as a file on the file system.

Larger data sets usually require a database and a data layer in the application to manage database access. iOS and Android both ship with the SQLite database engine, and access to the data is simplified by Xamarin's platform. In this chapter we are going to learn how to store information in a database on the iOS and Android mobile platforms.

Xamarin.iOS and Xamarin.Android support the following database access APIs for the built-in SQLite database engine:

- SQLite-NET 3rd party library.
- ADO.NET framework.

There are two sample solutions for this chapter:

- **DataiOS_demo1** – Implement SQLite-NET on iOS.
- **DataAndroid_demo1** – Implement SQLite-NET on Android.
- **TaskyPro** – Simple example with data updates and deletion.

The code introduced in this chapter is already present in the samples, but commented out. Use the **Tasks** window in your IDE to quickly find the commented-out code that is marked with a `//TODO:` task identifier.

Storing Data

When to use a Database

While the storage and processing capabilities of mobile devices are increasing, phones and tablets still lag behind their desktop & laptop counterparts. For this reason it is worth taking some time to plan the data storage architecture for your app rather than just assuming a database is the right answer all the time. There are a number of different options that suit different requirements, such as:

- **Preferences** – Both iOS and Android offer a built-in mechanism for storing simple key-value pairs of data. If you are storing simple user settings or small pieces of data (such as personalization information) then use the platform's native features for storing this type of information. For iOS you can also take advantage of iCloud synchronization for this data, both for backup and synchronization for users with multiple devices.

- Text Files – User input or caches of downloaded content (eg. HTML) can be stored directly on the file-system. Use an appropriate file-naming convention to help you organize the files and find data.
- Serialized Data Files – Objects can be persisted as XML or JSON on the file-system. The .NET framework includes libraries that make serializing and de-serializing objects easy. Use appropriate names to organize data files.
- Database – The SQLite database engine is available on both iOS and Android platforms, and is useful for storing structured data that you need to query, sort or otherwise manipulate. Database storage is suited to lists of data with many properties.
- Image files – Although it's possible to store binary data in the database on a mobile device, it is recommended that you store them directly in the file-system. If necessary you can store the filenames in a database to associate the image with other data. When dealing with large images, or lots of images, it is good practice to plan a caching strategy that deletes files you no longer need to avoid consuming all the user's storage space.

If a database is the right storage mechanism for your app, the remainder of this document discusses how to use SQLite in a cross-platform application on the Xamarin platform.

Advantages of using a Database

There are a number of advantages to using an SQL database in your mobile app:

- SQL databases allow efficient storage of structured data.
- Specific data can be extracted with complex queries.
- Query results can be sorted.
- Query results can be aggregated.
- Developers with existing database skills can utilize their knowledge to design the database and data access code.
- The data model from the server component of a connected application may be re-used (in whole or in part) in the mobile application.

SQLite Database Engine

SQLite is an open-source database engine that has been adopted by both Apple and Google for their mobile platforms. The SQLite database engine is built-in to both operating systems so there is no additional work for developers to take advantage of it. SQLite is well suited to cross-platform mobile development because:

- The database engine is small, fast and easily portable.
- A database is stored in a single file, which is easy to manage on mobile devices.

- The file format is easy to use across platforms: whether 32- or 64-bit, and big- or little-endian systems.
- It implements most of the SQL92 standard.

Because SQLite is designed to be small and fast, there are some caveats on its use:

- Some OUTER join syntax is not supported.
- Only table RENAME and ADDCOLUMN are supported. You cannot perform other modifications to your schema.
- Views are read-only.

You can learn more about SQLite on the website - http://SQLite.org - however all the information you need to use SQLite with Xamarin is contained in this document and associated samples. The SQLite database engine is built-in to all versions of iOS and was added to Android 2.

Although not covered in this chapter, SQLite is also available for use on Windows Phone and Windows Store (8 & RT) applications. There is more information on the Xamarin documentation website.

Configuration

The only configuration required to use SQLite in your application is to select the location for the database file.

Database File Path

Depending on what platform you are targeting the file location will be different. For iOS and Android you can use `Environment` class to construct a valid path, as shown in the following code snippet:

```
// dbPath contains a valid file path for the database file to be stored
string dbPath = Path.Combine (
        Environment.GetFolderPath (Environment.SpecialFolder.Personal),
        "database.db3");
```

There are other things to take into consideration when deciding where to store the database file. On iOS you may want the database to backed-up automatically, while on Android you can choose whether to use internal or external storage. These factors may require slightly different code on each platform.

If you wish to use a different location on each platform you can use a compiler directive as shown to generate a different path for each platform:

```
var sqliteFilename = "MyDatabase.db3";
#if __ANDROID__
// Just use whatever directory SpecialFolder.Personal returns
string libraryPath =
Environment.GetFolderPath(Environment.SpecialFolder.Personal); ;
#else
// we need to put in /Library/ on iOS5.1 to meet Apple's iCloud terms
// (they don't want non-user-generated data in Documents)
```

```
string documentsPath = Environment.GetFolderPath
(Environment.SpecialFolder.Personal); // Documents folder
string libraryPath = Path.Combine (documentsPath, "..", "Library"); //
Library folder instead
#endif
var path = Path.Combine (libraryPath, sqliteFilename);
```

Refer to the Working with the File System article in the Xamarin Developer Center for more information on what file locations to use on iOS. For hints on using the file system in Android, see the Browse Files recipe. Also, see the Building Cross Platform Applications document for more information on using compiler directives to write code specific to each platform.

Using SQLite.NET ORM

ORM stands for Object Relational Mapping – libraries that automatically let you save and retrieve "objects" from a database without writing SQL statements generally fit into this category. There are many different levels of support, from basic storage and retrieval to complex relationship modeling and query performance optimizations. The SQLite.NET library that Xamarin recommends is a basic ORM that lets you easily store and retrieve objects in the local SQLite database on an Android or iOS device.

The SQLite.NET library also works on Windows Phone and Windows Store applications written in Visual Studio. Some additional setup may be required on those platforms (such as downloading the **Community.CsharpSqlite** component), but it is beyond the scope of this document.

About SQLite.Net

To use SQLite.NET you must download the source and include it in your Xamarin project. The code is in a single file `SQLite.cs` file, available at https://github.com/praeclarum/sqlite-net/blob/master/src/SQLite.cs. Because SQLite.NET binds directly to the SQLite database engine on each platform you do not need to include any additional assembly references for it to work.

Creating a Blank Database

Passing the file path the `SQLiteConnection` class constructor can create a database reference. You do not need to check if the file already exists. It will automatically be created if required. Otherwise, the existing database file will be opened, as shown below:

```
var db = new SQLiteConnection (dbPath);
```

The `dbPath` variable should be determined according the rules discussed earlier in this document.

Model Classes

With SQLite a database table is defined by adding attributes to a class rather than a CREATE TABLE command. For example, the following class is used to define a `Session` table:

```
public class Session
```

```
{
    [PrimaryKey, AutoIncrement, Column("_id")]
    public int Id { get; set; }
    public string SpeakerName { get; set; }
    public string Title { get; set; }
    public string Abstract { get; set; }
    public string Location { get; set; }
    public DateTime Begins { get; set; }
    public DateTime Ends { get; set; }
}
```

By default, the underlying database table will have the same name as the class (in this case, "Session"). The actual table name is important if you write SQL queries directly against the database rather than use the ORM data access methods. The `PrimaryKey` and `AutoIncrement` attributes cause an auto-incremeting primary key to be created. The `[Column("_id")]` attribute is optional. If absent, a column will be added to the table with the same name as the property in the class.

Changing a Model Class

As noted earlier, SQLite has some limitations when it comes to changing the database schema: you can only add or rename columns. This affects what changes you can make to your Model classes, especially when you release subsequent versions of your application to existing customers:

- If you remove a property, the column will not be removed from databases that already exist on users devices.
- If you change a property name, there is no way for the SQLite.NET to know about the data under the old column name in your new version.
- If you attempt to change the Primary Key or other configuration done via attributes, that code will fail on existing installations.

It is possible to manage these limitations by careful planning of your data structures (especially as you extend your app and release new versions). You can also create "migration" SQL commands to bring older data into a new schema using ADO.NET, which is discussed towards the end of this chapter.

Executing Commands

Once you have created a `SQLiteConnection` object, database commands are executed by calling its methods, such as `CreateTable` and `Insert` like this:

```
db.CreateTable<Session> ();
db.Insert (session); // after creating the session object, insert it into
the database
```

More Complex Queries

The following methods on `SQLiteConnection` can be used to perform other data operations:

- Insert – Adds a new object to the database.
- Get<T> – Attempts to retrieve an object using the primary key.

- Table<T> – Returns all the objects in the table.
- Delete – Deletes an object using its primary key.
- Query<T> - Perform an SQL query that returns a number of rows (as objects).
- Execute – Use this method (and not `Query`) when you don't expect rows back from the SQL (such as INSERT, UPDATE and DELETE instructions).

Getting an object by the primary key

SQLite.Net provides the `Get` method to retrieve a single object based on its primary key.

```
var session = db.Get<Session>(1);
```

Selecting an object using Linq

Methods that return collections support `IEnumerable<T>` so you can use Linq to query or sort the contents of a table. The entire table is loaded into a collection prior to the Linq query executing, so performance of these queries could be slow for large amounts of data.

The following code shows an example using Linq to filter out all entries that begin with the letter "A":

```
var sessions = from s in db.Table<Session>()
    where s.Title.StartsWith ("A")
    select s;
Console.WriteLine ("-> " + sessions.FirstOrDefault ().Title);
```

Selecting an object using SQL

Even though SQLite.Net can provide object-based access to your data, sometimes you might need to do a more complex query than Linq allows (or you may need faster performance). You can use SQL commands with the `Query` method, as shown here:

```
var sessionsStartingWithA = db.Query<Stock>("SELECT * FROM Items WHERE
Title = ?", "A");
foreach (var s in sessionsStartingWithA) {
    Console.WriteLine ("a " + s.Title);
}
```

> Note: When writing SQL statements directly you create a dependency on the names of tables and columns in your database, which have been generated from your classes and their attributes. If you change those names in your code you must remember to update any manually written SQL statements.

Deleting an object

An object can be deleted from the database by passing it to the `Delete` method. The primary key is used to delete the row, so you can pass an object that was previously retrieved from a Query or Table or you can create a new object and set the primary key, as shown here:

```
var rowcount = db.Delete (new Sessions(){Id=1});
```

You can check the `rowcount` to confirm how many rows were affected (deleted in this case).

Threading

You should not use the same SQLite database connection across multiple threads. Be careful to open, use and then close any connections you create on the same thread.

To ensure that your code is not attempting to access the SQLite database from multiple threads at the same time, manually take a lock whenever you are going to access the database, like this:

```
lock (locker){
    // Do your query or insert here
}
```

All database access (reads, writes, updates, etc) should be wrapped with the same lock, taking care not to create a deadlock situation by ensuring that the work inside the lock clause is kept simple and does not call out to other methods that may also take a lock.

Using Data In An Application

The sample code for this chapter looks like this when running on iOS and Android respectively:

The example adds the session data retrieved from the web (from chapter 9) to a database, and subsequently reads and present it on screen. In doing so, the

example demonstrates a database interaction using the SQLite.NET library to encapsulate the underlying database access.

It shows:

- Creating the database file
- Inserting data by creating objects and then saving them
- Querying the data
- Using shared code for the database interactions

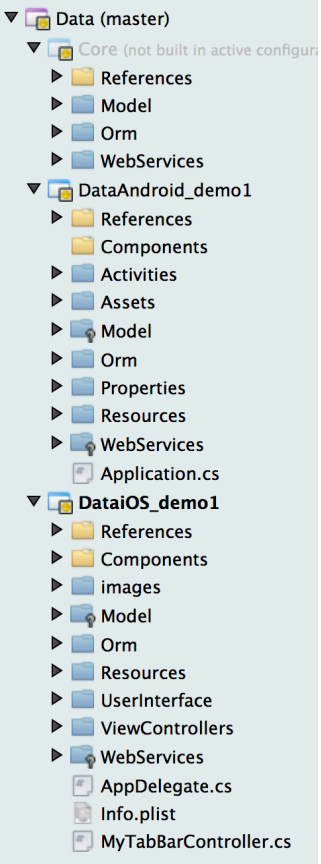
Namespaces

SQLite requires the following namespaces:

```
using System.Data;
using SQLite; // from SQLite.cs
```

Projects

The projects are shown below – the Core project contains all the shared files relating to SQLite. The application projects link to these files to provide database functionality on iOS and Android with the same code.



The **Orm** directory includes a `SessionDatabase` class, discussed in the following sections, to perform all data access.

The native UI code for the View Controllers and Activities can be found in the iOS and Android projects respectively, and has been covered in previous chapters.

Creating the Database

Using the file path that was discussed earlier, we can set-up the SQLite-NET database as a singleton on both iOS and Android with the following code:

```
public static ConferenceDatabase Database { get { return databaseInstance;
} }
static readonly ConferenceDatabase databaseInstance = new
ConferenceDatabase (ConferenceDatabase.DatabaseFilePath);
```

In the sample code this property is created on the `AppDelegate` in iOS and an `Application` subclass in Android. The Database instance can then be easily referenced throughout the application.

iOS

For example, once the Database property has been set on the AppDelegate, we can refer to the database using code like this in the `MyTabBarController.cs` file:

```
AppDelegate.Database.DeleteSessions ();
AppDelegate.Database.SaveSessions (sessions);
```

Android

Similarly, once the Database property has been set in the Android Application subclass, we can access the database using code like this in the `SessionsActivity.cs` file:

```
EvolveApp.Database.DeleteSessions ();
EvolveApp.Database.SaveSessions (sessions);
```

Read

There are a couple of read operations in the sample:

- Reading the list of sessions
- Reading individual sessions

The two methods in the `SessionDatabase` class are:

```
public List<Session> GetSessions ()
{
    lock (locker) {
        return (from i in Table<Session> () select i).ToList ();
    }
}
public Session GetSession (int id)
{
    lock (locker) {
        return Table<Session>().FirstOrDefault(x => x.Id == id);
    }
}
```

Each platform renders the data differently: iOS uses a `UITableView` and Android uses a `ListView`.

Create and Update

To save a single session, a method is provided that does an `Insert` or `Update` depending on whether the `PrimaryKey` has been set:

```
public int SaveSession (Session item)
{
    lock (locker) {
        if (item.Id != 0) {
            Update (item);
            return item.Id;
        } else {
            return Insert (item);
        }
    }
}
```

Because the `Id` property is marked with a `PrimaryKey` attribute you should not set it in your code.

Additionally, multiple sessions can be saved using the `InsertAll` method:

```
public void SaveSessions (List<Session> sessions)
{
    lock (locker) {
        InsertAll (sessions);
    }
}
```

Real world applications will usually require some validation (such as required fields, minimum lengths or other business rules). Good cross-platform applications implement as much of the validation logical as possible in shared code, passing validation errors back up to the UI for display according to the platform's capabilities.

Delete

Unlike the `Insert` and `Update` methods, the `Delete<T>` method can accept just the primary key value rather than a complete `Session` object. In this example a `Session` object is passed into the method but only the `Id` property is passed on to the `Delete<T>` method.

```
public int DeleteSession (Session session)
{
    lock (locker) {
        return Delete<Session> (session.Id);
    }
}
```

There is also a `DeleteAll <T>` method, which will clear a table, as shown:

```
public void DeleteSessions()
{
    lock (locker) {
        DeleteAll<Session> ();
    }
```

```
}
```

Updating and Deleting Objects

Updating and deleting objects is better demonstrated in an application that has some data entry, such as the Tasky to do list sample. The following two methods from the `TaskDatabase` class are used for editing and deleting `Task` objects:

```
public int SaveItem<T> (T item) where T : Contracts.IBusinessEntity
{
    lock (locker) {
        if (item.ID != 0) {
            Update (item);
            return item.ID;
        } else {
            return Insert (item);
        }
    }
}
public int DeleteItem<T>(int id) where T : Contracts.IBusinessEntity,
        new ()
{
    lock (locker) {
        return Delete<T> (id);
        //return Delete (new Task() {ID = id}); would also have worked
    }
}
```

iOS

In the iOS project, saving changes to a task simply requires the data entry values from the user interface to be set on the `currentTask` object, which is then saved using SQLite.NET. The `currentTask` object's `Id` property will be set if the data was previously loaded from the database, otherwise it will be -1. The SaveTask method will use the Id to determine whether to INSERT or UPDATE the row.

```
// currentTask is stored in the view
currentTask.Name = taskDialog.Name; // data entered in MonoTouch.Dialog
form
currentTask.Notes = taskDialog.Notes;
TaskManager.SaveTask(currentTask);
```

The code itself is trivial but demonstrates how easy it is to add or change data with SQLite.NET.

Android

The Android code looks identical to the iOS version, except we are now retrieving the data from Android's user interface controls.

```
// currentTask is stored in the activity
task.Name = nameTextEdit.Text;  // data entered in Activity
task.Notes = notesTextEdit.Text;
TaskManager.SaveTask(task);
```

Using ADO.NET

Xamarin.iOS has built-in support for the SQLite database that is available on iOS and Android, exposed using familiar ADO.NET-like syntax. Using these APIs requires you to write SQL statements that are processed by SQLite, such as CREATE TABLE, INSERT and SELECT statements.

ADO.NET is an alternative to the SQLite.NET ORM discussed above. ADO.NET requires you to write SQL statements for all interactions with the database, which means you will be responsible for translating your objects to and from SQL statements when you write or read from the database.

> **NOTE:** One important aspect of constructing SQL statements is ensuring you correctly escape special characters like the apostrophe to prevent user-input from causing invalid SQL commands (also known as SQL injection).

Assembly References

To use access SQLite via ADO.NET you must add `System.Data` and `Mono.Data.Sqlite` references to your iOS or Android project.

To add reference in Xamarin Studio, right-click **References > Edit References...** then tick to select the required assemblies.

About Mono.Data.Sqlite

This section discusses how to use the `Mono.Data.Sqlite.SqliteConnection` class to create a blank database file and then to instantiate `SqliteCommand` objects to execute SQL instructions against the database.

Creating a Blank Database

The following code snippet creates a database by calling the `CreateFile` method with a valid, writeable file path:

```
Mono.Data.Sqlite.SqliteConnection.CreateFile (dbPath);
```

You should check whether the file already exists before calling this method, otherwise a new database will be created over the top of the old one, and the data in the old file will be lost.

The `dbPath` variable should be determined according the rules discussed earlier in this document.

Creating a Database Connection

After the SQLite database file has been created you can create a connection object to access the data. The connection is constructed with a connection string which takes the form of `Data Source=file_path`, as shown here:

```
var connection = new SqliteConnection ("Data Source=" + dbPath);
connection.Open();
// do stuff
connection.Close();
```

You should never re-use a connection across different threads. If in doubt, create the connection as required and close it when you're done. However, be mindful of doing this more often than required too.

Creating and Executing a Database Command

Once you have a connection, you can execute arbitrary SQL commands against it. The code below shows a CREATE TABLE statement being executed.

```
using (var command = connection.CreateCommand ()) {
    command.CommandText = "CREATE TABLE [Items] ([_id] int, [Symbol]
ntext, [Name] ntext);";
    var rowcount = command.ExecuteNonQuery ();
}
```

When executing SQL directly against the database, take the normal precautions not to make invalid requests, such as attempting to create a table that already exists. Keep track of the structure of your database so that you don't cause a `SqliteException` with a message such as "SQLite error table [Items] already exists."

Basic Data Access

The following code sample shows a typical database interaction, including:

- Creating the database file
- Inserting data
- Querying data

These operations would normally appear in multiple places throughout your code. For example you may create the database file and tables when your application first starts and perform data reads and writes in individual screens in your app. In the example below groups things into a single method for brevity:

```
public static SqliteConnection connection;
public static string DoSomeDataAccess ()
{
    // determine the path for the database file
    string dbPath = Path.Combine (
    Environment.GetFolderPath (Environment.SpecialFolder.Personal),
    "mydatabase.db3");

    bool exists = File.Exists (dbPath);

    if (!exists) {
        // Create the database before seeding it with some data
        Mono.Data.Sqlite.SqliteConnection.CreateFile (dbPath);
        connection = new SqliteConnection ("Data Source="+ dbPath);

        var commands = new[] {
        "CREATE TABLE [Items] (_id ntext, Things ntext);",
        "INSERT INTO [Items] ([_id], [Item]) VALUES ('1', item1)",
        "INSERT INTO [Items] ([_id], [Item]) VALUES ('2', item2)",
        "INSERT INTO [Items] ([_id], [Item]) VALUES ('3', item3)"
```

```
            };

            // Open the database connection and create table with data
            connection.Open ();
            foreach (var command in commands) {
                using (var c = connection.CreateCommand ()) {
                    c.CommandText = command;
                    var rowcount = c.ExecuteNonQuery ();
                }
            }
        } else {
            // Open connection to existing database file
            connection = new SqliteConnection ("Data Source="+ dbPath);
            connection.Open ();
        }

        // query the database to prove data was inserted
        using (var contents = connection.CreateCommand ()) {
            contents.CommandText = "SELECT [_id], [Item] from [Items]";
            var r = contents.ExecuteReader ();
        while (r.Read ())
                Console.WriteLine("\tKey={0}; Value={1}",
                    r ["_id"].ToString (),
                    r ["Item"].ToString ());
        }
        connection.Close ();
    }
```

Complex Queries

Because SQLite allows arbitrary SQL commands to be run against the data, you can perform whatever CREATE, INSERT, UPDATE, DELETE or SELECT statements you like. You can read about the SQL commands supported by SQLite at the SQLite website at www.sqlite.org/lang.html. The SQL statements are run using one of three methods on an `SqliteCommand` object:

- **ExecuteNonQuery** – Typically used for table creation or data insertion. The return value for some operations is the number of rows affected, otherwise it's -1.
- **ExecuteReader** – Used when a collection of rows should be returned as a `SqlDataReader`.
- **ExecuteScalar** – Retrieves a single value (for example an aggregate).

EXECUTENONQUERY

INSERT, UPDATE and DELETE statements will return the number of rows affected. All other SQL statements will return -1. The following snippet shows an example of an INSERT statement called with `ExecuteNonQuery`:

```
using (var c = connection.CreateCommand ()) {
    c.CommandText = "INSERT INTO [Items] ([_id], [Item]) VALUES ('1',
'Item1')";
    var rowcount = c.ExecuteNonQuery (); // rowcount will be 1
```

```
}
```

EXECUTEREADER

The following method shows a WHERE clause in the SELECT statement:

```
public static string MoreComplexQuery ()
{
    var output = "";
    output += "\nComplex query example: ";
    string dbPath = Path.Combine (
        Environment.GetFolderPath (Environment.SpecialFolder.Personal),
"ormdemo.db3");

    connection = new SqliteConnection ("Data Source=" + dbPath);
    connection.Open ();
    using (var contents = connection.CreateCommand ()) {
        contents.CommandText =
               "SELECT * FROM [Items] WHERE Symbol = 'MSFT'";
        var r = contents.ExecuteReader ();
        output += "\nReading data";
        while (r.Read ())
            output += String.Format ("\n\tKey={0}; Value={1}",
                    r ["_id"].ToString (),
                    r ["Symbol"].ToString ());
    }
    connection.Close ();

    return output;
}
```

The code creates a complete SQL statement, therefore it must escape reserved characters such as the quote (') around strings.

The `ExecuteReader` method returns a `SqliteDataReader` object. In addition to the `Read` method shown in the example, other useful properties include:

- **RowsAffected** – Count of the rows affected by the query.
- **HasRows** – Whether any rows were returned.

EXECUTESCALAR

The `ExecuteScalar` method is used for SELECT statements that return a single value (such as an aggregate), as shown below:

```
using (var contents = connection.CreateCommand ()) {
    contents.CommandText = "SELECT COUNT(*) FROM [Items] WHERE Item <>
Item1";
    var i = contents.ExecuteScalar ();
}
```

The `ExecuteScalar` method's return type is `object`. You should cast the result depending on the database query. The result could be an integer from a COUNT query or a string from a single column SELECT query. Note that this is different to other `Execute` methods that return a reader object or a count of the number of rows affected.

Summary

This chapter discussed data access in Xamarin.iOS and XamarinAndroid using SQLite as the database engine. We explained how to implement the SQLite.NET ORM to perform data operations and added SQLite functionality to the sample application.

The ADO.NET framework was also introduced as an alternative to the SQLite.NET ORM.

For additional examples of cross-platform data access see our pre-built apps Field Service and Employee Directory, as well as the Tasky Pro and MWC 2012 case studies.