

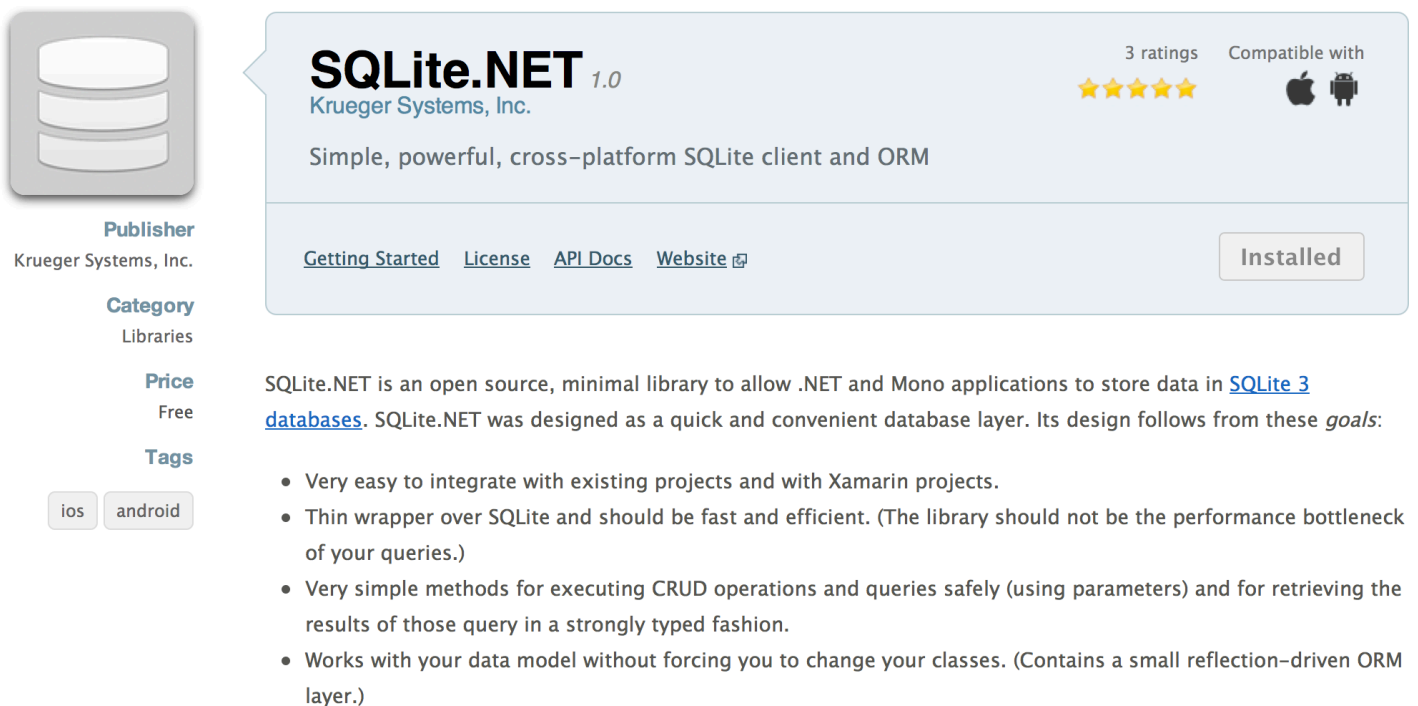
Part 3 - Using SQLite.NET ORM

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 very basic ORM that lets you easily store and retrieve objects in the local SQLite database on an Android or iOS device.

Using SQLite.NET


There are two ways to include SQLite.NET in your Xamarin project:


- **Github** – The code is in a single file [SQLite.cs file on Github](#). 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. This C# file uses compiler directives to support a variety of platforms in the same codebase, including Windows platforms.
- **Component Store** – SQLite.NET is available for iOS and Android from the Xamarin Component Store, as shown in this screenshot:



SQLite.NET 1.0
Krueger Systems, Inc.

Simple, powerful, cross-platform SQLite client and ORM

3 ratings Compatible with 

[Getting Started](#) [License](#) [API Docs](#) [Website](#) 

Installed

Publisher
Krueger Systems, Inc.

Category
Libraries

Price
Free

Tags
ios android

SQLite.NET is an open source, minimal library to allow .NET and Mono applications to store data in [SQLite 3 databases](#). SQLite.NET was designed as a quick and convenient database layer. Its design follows from these *goals*:

- Very easy to integrate with existing projects and with Xamarin projects.
- Thin wrapper over SQLite and should be fast and efficient. (The library should not be the performance bottleneck of your queries.)
- Very simple methods for executing CRUD operations and queries safely (using parameters) and for retrieving the results of those query in a strongly typed fashion.
- Works with your data model without forcing you to change your classes. (Contains a small reflection-driven ORM layer.)

Regardless of which method you use to include SQLite.NET in your application, the code to use it the same. Once you have the SQLite.NET library available, follow these three steps to use it to access a database:

1. Add a using statement

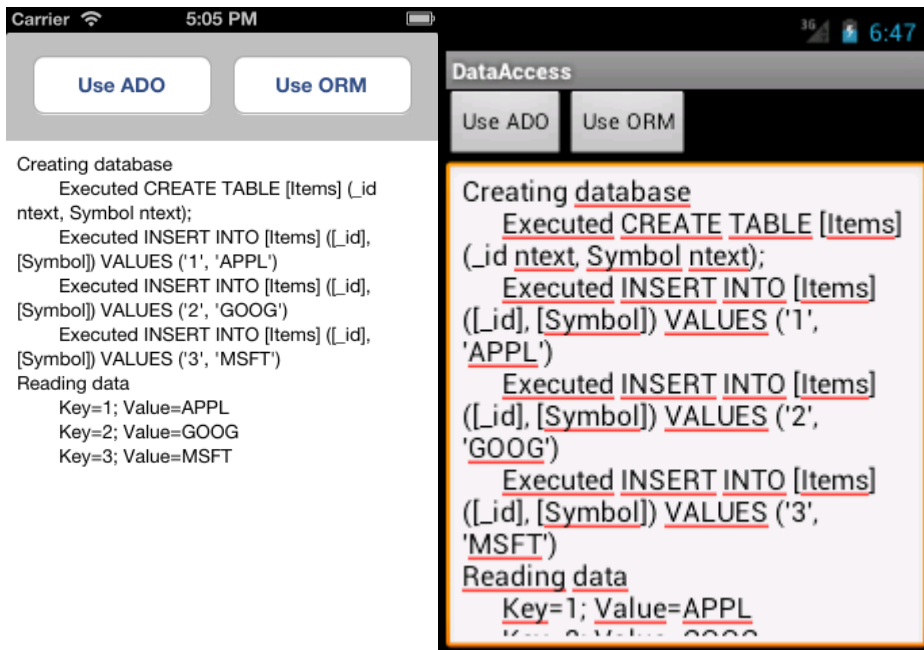
Add the following statement to the C# files where data access is required: `using SQLite;`

2. **Create a Blank Database** A database reference can be created by passing the file path the `SQLiteConnection` class constructor. 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. `var db = new SQLiteConnection (dbPath);` The `dbPath` variable should be determined according the rules discussed earlier in this document.

3. **Save Data** Once you have created a `SQLiteConnection` object, database commands are executed by calling its methods, such as `CreateTable` and `Insert` like this: `db.CreateTable`
4. **Retrieve Data** To retrieve an object (or a list of objects) use the following syntax: `var stock = db.Get`

Basic Data Access

The *DataAccess_Basic* sample code for this document looks like this when running on iOS and Android respectively. The code illustrates how to perform simple SQLite.NET operations and shows the results in as text in the application's main window.



The following code sample shows an entire database interaction using the SQLite.NET library to encapsulate the underlying database access. It shows:

1. Creating the database file
2. Inserting some data by creating objects and then saving them
3. Querying the data

You'll need to include these namespaces:

```
using SQLite; // from the github SQLite.cs class
```

The last one requires that you have added SQLite to your project. Note that the SQLite database table is defined by adding attributes to a class (the `Stock` class) rather than a `CREATE TABLE` command.

```
[Table("Items")] public class Stock { [PrimaryKey, AutoIncrement, Column("_id")] public int Id {
get; set; } [MaxLength(8)] public string Symbol { get; set; } } public static void DoSomeDataAccess
() { Console.WriteLine ("Creating database, if it doesn't already exist"); string dbPath =
Path.Combine ( Environment.GetFolderPath (Environment.SpecialFolder.Personal), "ormdemo.db3"); var
db = new SQLiteConnection (dbPath); db.CreateTable
```

Using the `[Table]` attribute without specifying a table name parameter will cause the underlying database table to have the same name as the class (in this case, "Stock"). The actual table name is important if you write SQL queries directly against the database rather than use the ORM data access methods. Similarly the `[Column("_id")]` attribute is optional, and if absent a column will be

added to the table with the same name as the property in the class.

SQLite Attributes

Common attributes that you can apply to your classes to control how they are stored in the underlying database include:

- **[PrimaryKey]** – This attribute can be applied to an integer property to force it to be the underlying table's primary key. Composite primary keys are not supported.
- **[AutoIncrement]** – This attribute will cause an integer property's value to be auto-increment for each new object inserted into the database
- **[Column(name)]** – Supplying the optional `name` parameter will override the default value of the underlying database column's name (which is the same as the property).
- **[Table(name)]** – Marks the class as being able to be stored in an underlying SQLite table. Specifying the optional name parameter will override the default value of the underlying database table's name (which is the same as the class name).
- **[MaxLength(value)]** – Restrict the length of a text property, when a database insert is attempted. Consuming code should validate this prior to inserting the object as this attribute is only 'checked' when a database insert or update operation is attempted.
- **[Ignore]** – Causes SQLite.NET to ignore this property. This is particularly useful for properties that have a type that cannot be stored in the database, or properties that model collections that cannot be resolved automatically by SQLite.
- **[Unique]** – Ensures that the values in the underlying database column are unique.

Most of these attributes are optional, SQLite will use default values for table and column names. You should always specify an integer primary key so that selection and deletion queries can be performed efficiently on your data.

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 existingItem = db.Get
```

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 apple = from s in db.Table
```

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 stocksStartingWithA = db.Query
```

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

The primary key is used to delete the row, as shown here:

```
var rowcount = db.Delete
```

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