Day 28: Using SQLite.Net with Xamarin.Android

Today, we will explore on how to use SQLite.Net with Xamarin.Android. SQLite.Net is a open source library for SQLite written in .NET to allow light-weight mobile applications (or others) save data in a local SQLite database.

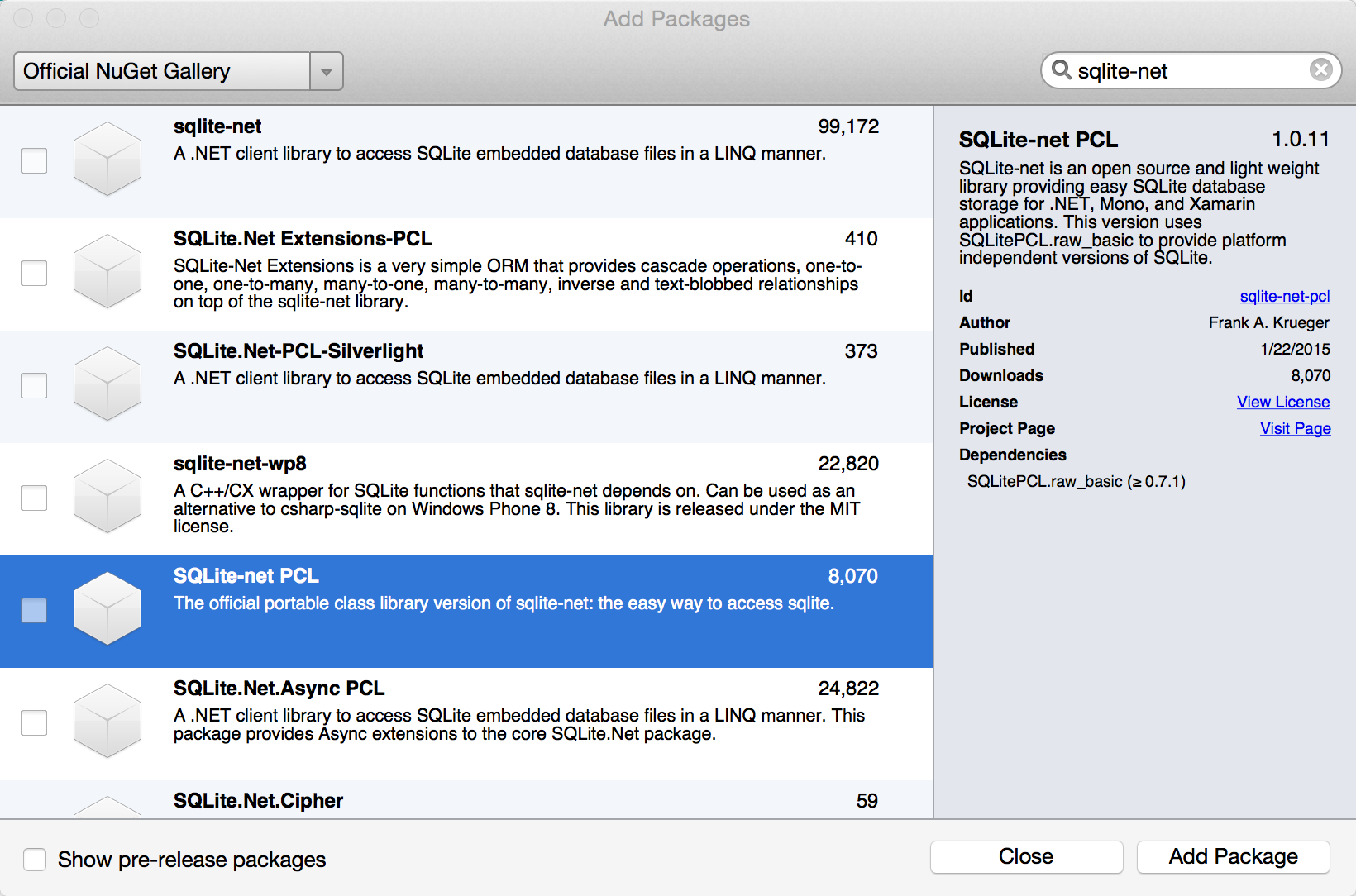
SQLite.Net implemented is written by Frank A Krueger, and the source of the project is all up on [Github](https://github.com/praeclarum/sqlite-net).

To use SQLite.Net in your Xamarin.Android application, you need to add a Nuget package called “SQLite-net PCL”

|  |
| --- |
| PM> Install-Package sqlite-net-pcl |

Note: There are a lot of SQLite Nuget packages, make sure you pick the one whose author is Frank A. Krueger.

‘’



Once you add the Nuget package, if you expand your Packages folder in Xamarin Studio, you should see two references



One for sqlite-net-pcl and one for SQLitePCL.raw\_basic. The raw library is a C++ implementation of SQLite.Net that handles a lot of operations.

We will be building a simple TODO List app that stores the data locally in today’s sample. Once of the packages are added, next step is to look at the API that we need to get familiar in order to work with SQLite.Net. Some of the APIs that we will be learning in today’s blogpost are –

* Column, Table level attributes
* SQLiteAsyncConnection
* CreateTable
* GetTable

## Column, Table level attributes

The first step that we need to do while working with SQLite.Net database is to define the class that holds the data. Let’s look at our ToDoItem class –

|  |
| --- |
| using System; using SQLite;  namespace SqliteSample {     [Table("Todo")]     public class ToDoItem     {         [PrimaryKey, AutoIncrement]         public int Id { get; set;}          [MaxLength(50)]         public string Title { get; set; }          public string Description { get; set; }          public DateTime CreatedAt { get; set; }          public bool IsComplete { get; set; }          [Ignore]         public string ReadableTitle         {             get             {                 return Title.ToString() + " Created At " + CreatedAt.ToLocalTime().ToShortDateString();             }                          }      } } |

Gist file link: <https://gist.github.com/vkoppaka/b1ce44ea777d5899c602>

Some of the attributes that we see in the above snippet are Table, PrimaryKey, AutoIncrement, MaxLength and Ignore. Let’s take a look at them in a little bit more detail –

**Table:** Table attribute defines that a given class that is decorated with this attribute is a SQLite Table. The Table attribute also takes a string parameter to denote what the name of the table is, in our case, it is “Todo”

**PrimaryKey:** PrimaryKey attribute indicates that a property that is decorated with PrimaryKey is the PrimaryKey for the given table.

**AutoIncrement:** AutoIncrement tells SQLite that the field needs to be automatically incremented by the database. This property is generally used along with PrimaryKey so the database handles the Id in on its own rather than us worrying about what the next Id would be.

**MaxLength:** MaxLength attribute tells the property that is decorated with it can only have a maximum length of the defined number.

**Ignore:** Ignore tells the SQLite database that the property is not managed by the database and it should not worry about persisting it.

## SQLiteAsyncConnection, CreatTable, Table

Now that we have defined our table, let’s learn how we can connect to the database so we can create this table. SQLite.Net PCL version uses .NET’s async pattern heavily so the operations that it performs are not blocking its calling threads. If for some reason, you don’t need the Async calls, simply remove “Async” from all the APIs we will be using and you should be good to go.

|  |
| --- |
| using SQLite; using System.Collections.Generic; using System.Threading.Tasks;  namespace SqliteSample {     public class TodoItemRepository     {         private readonly SQLiteAsyncConnection sqlConnection;          public TodoItemRepository()         {             sqlConnection = new SQLiteAsyncConnection(DatabaseFilePathRetriever.GetPath());             sqlConnection.CreateTableAsync<ToDoItem>();         }          *//Get all ToDoItems*         public async Task<List<ToDoItem>> GetToDoItems()         {             return  await sqlConnection.Table<ToDoItem>().ToListAsync(); *//Careful with ToList calls on databases!*         }          *//Get specific ToDoItem*         public async Task<ToDoItem> GetToDoItem(int id)         {             return await sqlConnection.Table<ToDoItem>().Where(tdi => tdi.Id == id).FirstOrDefaultAsync();         }          *//Delete specific ToDoItem*         public async Task DeleteToDoItem(ToDoItem todoItem)         {             await sqlConnection.DeleteAsync(todoItem);         }          *//Add new ToDoItem to DB*         public async Task AddToDoItem(ToDoItem todoItem)         {             await sqlConnection.InsertAsync(todoItem);         }     } } |

Before we explore the methods here, let’s first see the first line in the constructor where we are getting a Database File Path. Let’s take a look at this class as well –

|  |
| --- |
| using System; using System.IO;  namespace SqliteSample {     public static class DatabaseFilePathRetriever     {         public static string GetPath()         {             const string filename = "TodoDB.db3";             var documentspath = Environment.GetFolderPath(Environment.SpecialFolder.Personal);             var path = Path.Combine(documentspath, filename);             return path;         }     } } |

The first requirement of a SQLite database is that it has to be saved on the file system of the device. So our DatabaseFilePathRetriever class tells us where in the file system should we be saving the SQLite database. We are getting the path for documents personal folder and then appending TodoDB.db3 to it to get our full path. This full path is in turn used by the SQLiteAsyncConnection class to instatiantiate itself. We are then calling CreateTableAsync<ToDoItem> to create the ToDoItem table on our database.

The rest of the methods are what are called CRUD operations on database. To get a list of rows in the ToDoItem table, we would be calling the .Table<ToDoItem> call. Please be really careful on calling ToList on any database as this would bring all the records into memory. To get a single ToDoItem by item, we would use the same .Table call and apply LINQ on top of it to find the item by ID. Delete is performed by the DeleteAsync call and Insert is performed by the InsertAsync call.

Let’s see how we can use this code in our Android app –

|  |
| --- |
| using System; using Android.App; using Android.Widget; using Android.OS;  namespace SqliteSample {     [Activity(Label = "SqliteSample", MainLauncher = true, Icon = "@drawable/icon")]     public class MainActivity : Activity     {         private TodoItemRepository repository;          protected override void OnCreate(Bundle bundle)         {             base.OnCreate(bundle);              *// Set our view from the "main" layout resource*             SetContentView(Resource.Layout.Main);              repository = new TodoItemRepository();              *// Get our button from the layout resource,*             *// and attach an event to it*             Button createItemButton = FindViewById<Button>(Resource.Id.createItemButton);                          createItemButton.Click += CreateItemButton\_Click;         }          async void CreateItemButton\_Click(object sender, EventArgs e)         {             var newToDoItem = new ToDoItem             {                 Title = "New Item",                 Description = "Some Description",                 CreatedAt = DateTime.UtcNow,                 IsComplete = false             };             await repository.AddToDoItem(newToDoItem);               var items = await repository.GetToDoItems();             Toast.MakeText(this, "Item created, there are " + items.Count + " in the database", ToastLength.Long).Show();         }     } } |

And the UI would look like below –

That’s it for today, see you all tomorrow.