# Database Example

Review the notes “Persistent Storage” under the heading “SQLite Database Access”.

## Installation and Setup of Database Layer

Go to Tools, Nuget Packet Manager -> Package Manager Console. At the PM> Prompt, Type:

Install-Package sqlite-net-pcl -Version 1.4.118

Right click the solution, Manage Nuget Packages for Solution, click the installed tab, choose the sqlite-net-pcl package and check both other projects and choose install.

Add interface to App class at the end in the same file as App:

public interface IFileHelper

{

string GetLocalFilePath(string filename);

}

Add FileHelper.cs to each other project:

Both projects need a FileHelper.cs file which will allow a platform specific approach to locating the actual database file. These two classes will implement an interface that we will define in our shared project called IFileHelper. The Dependency Service is used later on to determine which file to use for each platform, the assembly statement below sets up that link for each platform.

Android:

using System.IO;

using Xamarin.Forms;

[assembly: Dependency(typeof(XamarinFuelTracker.Droid.FileHelper))]

namespace XamarinFuelTracker.Droid

{

public class FileHelper : IFileHelper

{

public string GetLocalFilePath(string filename)

{

string path = System.Environment.GetFolderPath(System.Environment.SpecialFolder.Personal);

return Path.Combine(path, filename);

}

}

}

iOS:

using System.IO;

using Xamarin.Forms;

[assembly: Dependency(typeof(XamarinFuelTracker.iOS.FileHelper))]

namespace XamarinFuelTracker.iOS

{

public class FileHelper : IFileHelper

{

public string GetLocalFilePath(string filename)

{

string docFolder = Environment.GetFolderPath(Environment.SpecialFolder.Personal);

string libFolder = Path.Combine(docFolder, "..", "Library", "Databases");

if (!Directory.Exists(libFolder))

{

Directory.CreateDirectory(libFolder);

}

return Path.Combine(libFolder, filename);

}

}

}

## Fuel Purchase Class

Object Relational Mapping (ORM) is used to map object oriented types to specific database table. For example, below define a FuelPurchase type. This type will be linked with an SQLite table with the same name. The PrimaryKey and AutoIncrement keywords indicate that the ID field of the object will be used as a primary key that will auto increment in the associated table.

using SQLite;

namespace XamBookDatabase

{

public class FuelPurchase

{

[PrimaryKey, AutoIncrement]

public int ID { get; set; }

public DateTime date { get; set; }

public double litres { get; set; }

public double cost { get; set; }

}

}

## Fuel Helper Database Class

Review the heading “Database Helper Class” in the notes. We will be implementing a layer that will handle communication with the SQLite database and hide that implementation from our user interface.

This can just go inside the App class, or another file. Doesn’t really matter:

public class Fuel\_Database

{

readonly SQLiteConnection database;

The above line is a connection to the database file that can be used to query and interact with the data. It is setup in the constructor below. The constructor for this example is also creating a hard coded row in the table if there isn’t one already, just so we have something to look at later before we are completely finished our interface.

public Fuel\_Database(string dbPath)

{

database = new SQLiteConnection(dbPath);

//database.DropTable<FuelPurchase>(); // can call this to drop if needed

database.CreateTable<FuelPurchase>(); // won’t do anything if already exists

if(database.Table<FuelPurchase>().Count()==0) //if no records make one

{

// this will get the next key

FuelPurchase purchase = new FuelPurchase();

purchase.cost = 15;

purchase.date = new DateTime(2018, 1, 15);

purchase.litres = 15;

SaveItem(purchase);

}

}

public List<FuelPurchase> GetItems()

{

return database.Table<FuelPurchase>().ToList<FuelPurchase>();

}

The above method is using the Language Integrated Query (LINQ) extension method ToList() to convert the results of the Table method into a standard list collection. The Table method is used to query a table and return all rows, though through chaining the result can be further filtered (see GetItem(int id) below).

public List<FuelPurchase> GetItemsOverTen()

{

return database.Query<FuelPurchase>("SELECT \* FROM [FuelPurchase] WHERE [litres] > 10");

}

As demonstrated above, you may also use the Query method to run arbitrary SQL SELECT statements which will be returned as the object of the type specified.

public FuelPurchase GetItem(int id)

{

return database.Table<FuelPurchase>().Where(i => i.ID == id).FirstOrDefault();

}

public int SaveItem(FuelPurchase item)

{

if (item.ID != 0)

{

return database.Update(item);

}

else

{

return database.Insert(item);

}

}

When inserting, updating, and deleting, we should be using our ORM class to pass around as data – These items will then become parameters of the various CRUD operation methods.

public int DeleteItemAsync(FuelPurchase item)

{

return database.Delete(item);

}

}

## App Class Properties

static Fuel\_Database database;

public static Fuel\_Database Database

{

get

{

if (database == null)

{

database = new Fuel\_Database (DependencyService.Get<IFileHelper>().GetLocalFilePath("FuelSQLite.db3"));

}

return database;

}

}

This property will be accessible from anywhere in the application as App.Database. We will use the database variable to do queries and so on. We’ll be calling the “Dependency Service” to find out which platform’s file location to use.

## Basic User Interface

To create a basic user interface, we will use the code approach in the App constructor. This will include Entry elements to show/edit the data and buttons to run code. After the Initialize Component call add:

database = Database;

EntryCell eLitres = new EntryCell { Label = "Litres:" };

EntryCell eDate = new EntryCell { Label = "Date:" };

EntryCell eCost = new EntryCell { Label = "Cost:" };

EntryCell eID = new EntryCell { Label = "ID:" };

var btnSearch = new Button { Text = "Read" };

btnSearch.Clicked += (sender, e) =>

{

FuelPurchase purchase = database.GetItem(Convert.ToInt32(eID.Text));

eID.Text = purchase.ID.ToString();

eDate.Text = purchase.date.ToString();

eLitres.Text = purchase.litres.ToString();

eCost.Text = purchase.cost.ToString();

};

MainPage = new ContentPage

{

Content = new StackLayout

{

Spacing = 25, // this will alter distance between elements

Padding = 75, // this will alter distance from side of frame

Children =

{ new TableView{Intent = TableIntent.Form, Root =

new TableRoot{

new TableSection("Fuel Purchase"){eID, eDate, eLitres, eCost } }},

new StackLayout{Orientation = StackOrientation.Horizontal,

HorizontalOptions = LayoutOptions.Center,

Children = {btnNew, btnDelete, btnSave, btnSearch } }

},

},

};

Try it out. Should be able to search for ID #1 since it is added by default.

## Insert, Update, Delete

In the App constructor, after the search button is defined, add:

var btnNew = new Button { Text = "New" };

btnNew.Clicked += (sender, e) => {

eID.Text = "0"; eCost.Text = "";

eDate.Text = ""; eLitres.Text = "";

};

var btnDelete = new Button { Text = "Delete" };

btnDelete.Clicked += (sender, e) =>

{ database.DeleteItemAsync(database.GetItem(Convert.ToInt32(eID.Text)));

// add code to clear text boxes too maybe

};

var btnSave = new Button { Text = "Save" };

btnSave.Clicked += (sender, e) =>

{

FuelPurchase purchase = new FuelPurchase()

{

ID = Convert.ToInt32(eID.Text),

cost = Convert.ToDouble(eCost.Text),

litres = Convert.ToDouble(eLitres.Text),

date = Convert.ToDateTime(eDate.Text)

};

database.SaveItem(purchase);

};

Also will need to add to the layout somewhere.

Right now there are some issues with functionality, clearing of text boxes, trying to navigate when nothing there, deleting an item that doesn’t exist, etc. Lots of possible exceptions.

Work through the exercise in the same folder to “clean up” the functionality of the Fuel Tracker App.