IT6035
Mobile Development
Database Tutorial

Course Level: 6

Course Credits: 15

Activity: Working with a local database

In this tutorial activity you will learn to set up database connection to SQLite database, write and execute code to perform basic database operations.

Create new Xamarin.Forms project named **LocalDatabaseTutorial**. In this project you will work with the local CustomerDatabase to implement functionality for working with customer records.
Adding functionality to your app to perform database operations can be divided into several steps. They are:
1. Adding appropriate NuGet packages for working with databases

2. Creating database connection for each project that will be working with databases

3. Creating data models corresponding to the database tables

4. Setting up CRUD (Create, Read, Update and Delete) operations behavior

5. Set up user interface to support the database operations

You will follow the above steps in this activity.

1. Adding appropriate NuGet packages for working with databases

To add a NuGet package to the solution:

1. Right click on the solution in the Solution Explorer and select the Manage NuGet packages for solution option,

2. In the dialog select the Browse tab to see the available packages,

3. Click on the required package to add it to the solution.

Follow the instructions above to add the **sqlite-net-pcl** package to the LocalDatabaseTutorial solution. This package has the following description: "SQLite-net is an open source and light weight library providing easy SQLite database storage for .NET, Mono, and Xamarin applications. This version uses SQLitePCLRaw to provide platform independent versions of SQLite."

After installation, the package should appear under the Installed tab. You should also see the sqlite-net-pcl package in Solution Explorer under Dependencies - NuGet in the *LocalDatabaseTutorial* project and under References in the *LocalDatabaseTutorial.Android* and *LocalDatabaseTutorial.iOS* projects.

2. Creating database connection

The procedure for creating database connection varies based on the project type. iOS project has a different process from the Android one.

Adding IDatabaseConnection interface

1. First, add new interface named **iDatabaseConnection** in the shared project (*LocalDatabaseTutorial*). Creating this interface will make sure that every class implementing the interface must have the DbConnection() method with the same method signature. It is an easy way to make sure that all the projects in the solution provide the correct method for connecting to the database.

To add an interface right click the *LocalDatabaseTutorial* project in Solution Explorer and select **Add** – **New Item**. Find **Interface** type under **Visual C# Items** and specify the interface name as **IDatabaseConnection**.

2. Add the appropriate code so that your interface looks like the one shown below:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace LocalDatabaseTutorial
{
public interface IDatabaseConnection
{
SQLite.SQLiteConnection DbConnection();
}
}
```

Implementing iDatabaseConnection interface and setting up connection methods
In this section you will create classes in the iOS and Android projects responsible for creating database connection. Those classes will implement the **IDatabaseConnection** interface.

3. In the *LocalDatabaseTutorial.Android* project, add the **DatabaseConnection_Android** class.

To add new class to the project right click the project in Solution Explorer and select **Add** – **New Item**. Find **Class** type under **Visual C# Items** and specify the class's name.

4. Add the appropriate code so that your class looks like the one shown below:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
```

```
using LocalDatabaseTutorial.Droid;
using SQLite;
[assembly: Xamarin.Forms.Dependency(typeof(DatabaseConnection_Android))]
namespace LocalDatabaseTutorial.Droid
{
public class DatabaseConnection_Android: IDatabaseConnection
{
public SQLiteConnection DbConnection()
{
var dbName = "CustomersDatabase.db3";
var path = Path.Combine(System.Environment.GetFolderPath(System.Environment.SpecialFolder.Personal), dbName);
return new SQLiteConnection(path);
}
}
}
```

5. In the iOS project, add the **DatabaseConnection_iOS** class and add the appropriate code so that your class looks like the one shown below:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Foundation;
using UIKit;
using SQLite;
using System.IO;
namespace LocalDatabaseTutorial.iOS
{
class DatabaseConnection_iOS: IDatabaseConnection
{
public SQLiteConnection DbConnection()
{
var dbName = "CustomersDatabase.db3";
string personalFolder = System.Environment.GetFolderPath(Environment.SpecialFolder.Personal);
string libraryFolder = Path.Combine(personalFolder, "..", dbName);
var path = Path.Combine(libraryFolder, dbName);
return new SQLiteConnection(path);
}
}
}
```

See that both classes implement the **IDatabaseConnection** interface and have the **DbConnection()** method created.

The DbConnection() method has the return type of SQLiteConnection. In the method implementations you are returning the SQLiteConnection object using the SQLiteConnection(String) constructor, where the String parameter specifies the connection string.

The connection string is made up of the database name and database location.
The database name is specified with the *dbName* variable. In this tutorial you will create the database named CustomerDatabase. This database will hold customer information. The database path is specified through the use of System.Environment.

Environment class provides information about, and means to manipulate, the current environment and platform. GetFolderPath() method gets the path to the system special folder that is identified by the specified enumeration. Environment.SpecialFolder.Personal is the directory that serves as a common repository for documents.

The System.Environment.SpecialFolder.Personal type maps to the path /data/data/[your.package.name]/files. This is a private directory to your application so you will not be able to see these files using a file browser unless it has root privileges.

3. Creating data models corresponding to the database tables

In this section you will create a Customer model class to represent a record template for the Customers table in the CustomerDatabase.

In the *LocalDatabaseTutorial* project, add a class named **Customer**. This class will represent the customer information that will provide template for the customer records to be stored in the Customers table in the database.

This class must implement the INotifyPropertyChanged interface to notify callers of changes in the data it stores. It will use special attributes in the SQLite namespace to annotate properties with validation rules and other information.

1. Add the appropriate code so that your class looks like the one shown below:

```
using System.ComponentModel;
using SQLite;
namespace LocalDatabaseTutorial
{
class Customer: INotifyPropertyChanged
{
private int _id;
[PrimaryKey, AutoIncrement]
public int Id
{
get
{
return _id;
}
set
{
this._id = value;
```

```
OnPropertyChanged(nameof(Id));
}
}
private string _companyName;
[NotNull]
public string CompanyName
{
get
{
return _companyName;
}
set
{
this._companyName = value;
OnPropertyChanged(nameof(CompanyName));
}
}
private string _physicalAddress;
[MaxLength(50)]
public string PhysicalAddress
{
get
{
return _physicalAddress;
}
set
{
this._physicalAddress = value;
OnPropertyChanged(nameof(PhysicalAddress));
}
}
private string _country;
public string Country
{
get
{
return _country;
}
set
{
this._country = value;
OnPropertyChanged(nameof(Country));
}
}
public event PropertyChangedEventHandler PropertyChanged;
```

```
private void OnPropertyChanged(string propertyName)
{
this.PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
}
}
```

In this class you specify that the Customer object will have the following fields:

- Id (primary key that is set to auto increment)
- CompanyName (has to be entered, cannot be null)
- PhysicalAddress (cannot be longer than 50 characters)
- Country

4. Setting up database CRUD operations

Add new class named **CustomerDataAccess** to the *LocalDatabaseTutorial* project. In this class, you will set up methods that will specify behavior for adding new records to the Customers table, retrieve specific Customer records, update and delete customer records from the Customers table.

1. First, add the following code to the class:

```
private SQLiteConnection database;
private static object collisionLock = new object();
public ObservableCollection<Customer> Customers { get; set; }
public CustomerDataAccess()
{
database = DependencyService.Get<IDatabaseConnection>().DbConnection();
database.CreateTable<Customer>();
this.Customers = new ObservableCollection<Customer>(database.Table<Customer>());
AddNewCustomer();
}
```

In the CustomerDataAccess() constructor you first set up the database used by this class to be the CustomerDatabase. DependencyService.Get<IDatabaseConnection>().DbConnection() practically finds the class in the project that implements the IDatabaseConnection (DatabaseConnection_Android or DatabaseConnection_iOS) and calls the DbConnection() method that will return the SQLiteConnection pointing to the CustomerDatabase.

Here you also create collection named Customers that will contain objects of type Customer.

2. Next, add the method for **adding new customers** to the Customers table. This method adds new customer with the default values.

```
public void AddNewCustomer()
{
this.Customers.Add(new Customer
{
CompanyName = "Enter company name here",
PhysicalAddress = "Enter address here",
Country = "Enter country here"
});
}
```

3. Add the following three methods are related to **reading database records**:

GetFilteredCustomers(string countryName) method retrieves customer records where the country is equal to the countryName specified.

GetFilteredCustomers() method retrieves customer records where the country is Italy.

GetCustomer(int id) method returns Customer record based on specific id specified.

```
public IEnumerable<Customer> GetFilteredCustomers(string countryName)
{
lock (collisionLock)
{
var query = from cust in database.Table<Customer>()
where cust.Country == countryName
select cust;
return query.AsEnumerable();
}
}
public IEnumerable<Customer> GetFilteredCustomers()
{
lock (collisionLock)
{
return database.Query<Customer>("SELECT * FROM Item WHERE Country = 'Italy'").AsEnumerable();
}
}
public Customer GetCustomer(int id)
{
lock (collisionLock)
{
return database.Table<Customer>().FirstOrDefault(customer => customer.Id == id);
}
}
```

4. Add the following two methods are related to **updating database records**:

SaveCustomer(Customer CustomerInstance) method checks Customer's id. If the customer record already exists, it updates the record, otherwise it inserts new record to the Customers table.

SaveAllCustomers() method checks id's of all the customer objects in the Customers ObservableCollection. For each customer it checks if the customer record already exists. If it does, the customer record gets updated, otherwise new record is inserted to the Customers table.

```
public int SaveCustomer(Customer customerInstance)
{
lock (collisionLock)
{
if (customerInstance.Id != 0)
{
database.Update(customerInstance);
return customerInstance.Id;
}
else
{
database.Insert(customerInstance);
return customerInstance.Id;
}
}
}
public void SaveAllCustomers()
{
lock (collisionLock)
{
foreach (var customerInstance in this.Customers)
{
if (customerInstance.Id != 0)
{
database.Update(customerInstance);
}
else
{
database.Insert(customerInstance);
}
}
}
}
```

5. Finally, add the following two method allows **deleting database records**:

DeleteCustomer(Customer CustomerInstance) method checks Customer's id. If the customer record with the id already exists, it removes the record.

DeleteAllCustomers() method deletes the Customers table and create a blank table in its place.

```
public int DeleteCustomer(Customer customerInstance)
{
var id = customerInstance.Id;
```

```
if (id != 0)
{
lock (collisionLock)
{
database.Delete<Customer>(id);
}
}
this.Customers.Remove(customerInstance);
return id;
}
public void DeleteAllCustomers()
{
lock (collisionLock)
{
database.DropTable<Customer>();
database.CreateTable<Customer>();
}
this.Customers = null;
this.Customers = new ObservableCollection<Customer>(database.Table<Customer>());
}
```

5. Set up user interface to support the database operations

1. To test the code you have created, modify the **MainPage.xaml** to look like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
xmlns:local="clr-namespace:LocalDatabaseTutorial"
x:Class="LocalDatabaseTutorial.MainPage">
<ListView x:Name="CustomerView"
ItemsSource="{Binding Path=Customers}"
ListView.RowHeight ="250" >
<ListView.ItemTemplate>
<DataTemplate>
<ViewCell>
<StackLayout Orientation="Vertical">
<Entry Text="{Binding Id}" IsEnabled="False"/>
<Entry Text="{Binding CompanyName}"/>
<Entry Text="{Binding PhysicalAddress}"/>
<Entry Text="{Binding Country}"/>
</StackLayout>
</ViewCell>
</DataTemplate>
</ListView.ItemTemplate>
</ListView>
<ContentPage.ToolbarItems>
<ToolbarItem Name="Add" Activated="AddButton_Clicked"
```

```
Priority="0" Order="Secondary" />
<ToolbarItem Name="Remove" Activated="RemoveButton_Clicked"
Priority="1" Order="Secondary" />
<ToolbarItem Name="Remove all" Activated="RemoveAllButton_Clicked"
Priority="2" Order="Secondary" />
<ToolbarItem Name="Save" Activated="SaveButton_Clicked"
Priority="3" Order="Secondary" />
</ContentPage.ToolbarItems>
</ContentPage>
```

Here you create a ContentPage that contains the ListView named CustomerView, where each view cell represents a customer record. The ContentPage also contains a toolbar with the operations you can perform on the ListView items. Those operations refer to the AddButton_Clicked, RemoveButton_Clicked, RemoveAllButton_Clicked and SaveButton_Clicked methods in the MainPage.xaml.cs class. The Priority property for the toolbar items allows specifying the order in which the ToolBarItems appear on the toolbar.

2. Modify the **MainPage.xaml.cs** to look like this:

```
using System;
using System.Linq;
using Xamarin.Forms;
using Xamarin.Forms.Xaml;
namespace LocalDatabaseTutorial
{
[XamlCompilation(XamlCompilationOptions.Compile)]
public partial class MainPage : ContentPage
{
CustomerDataAccess dataAccess;
public MainPage()
{
InitializeComponent();
dataAccess = new CustomerDataAccess();
this.BindingContext = dataAccess;
}
private void SaveButton_Clicked(object sender, EventArgs e)
{
this.dataAccess.SaveAllCustomers();
}
private void RemoveButton_Clicked(object sender, EventArgs e)
{
var currentCustomer = this.CustomerView.SelectedItem as Customer;
if (currentCustomer != null)
{
this.dataAccess.DeleteCustomer(currentCustomer);
}
```

```
}
private async void RemoveAllButton_Clicked(object sender, EventArgs e)
{
if (this.dataAccess.Customers.Any())
{
var result = await DisplayAlert("Confirmation", "Are you sure you want to remove all customers?", "OK", "Cancel");
if (result == true)
{
this.dataAccess.DeleteAllCustomers();
this.BindingContext = this.dataAccess;
}
}
}
private void AddButton_Clicked(object sender, EventArgs
e)
{
this.dataAccess.AddNewCustomer();
}
}
}
```

The BindingContext property represents the data source for the current page.
XamlCompilation(XamlCompilationOptions.Compile) allows for the XAML compilation. Microsoft documentation outlines the following benefits:

- It performs compile-time checking of XAML, notifying the user of any errors.
- It removes some of the load and instantiation time for XAML elements.
- It helps to reduce the file size of the final assembly by no longer including .xaml files.

6. Running the project on Android emulator

1. Build and run the project on Android emulator. You should see one record with default values displayed.

Modify the record, then press the Save option from the tool bar. You should see the CustomerID value updated and the new customer record appearing under the existing one.
Test the rest of the tool bar options.