How to use LocalDatabase in Windows Phone

This article is a step-by-step guide to start using a local database in a Windows Phone application.

Introduction

Some time ago I've written a how-to guide to install and use the SQLite as the database of a Windows Phone application. SQLite is good, but it not the native solution to store structured data and it is evidenced with the hard steps needed to install the SQLite support in an existing application. Moreover, the developer cannot use the LINQ to SQL component to perform queries and must learn the SQLite SQL sintax.

Install the local database support is much easier (in fact, the local database component is already in your solution, as you'll see in this article) and it also provides the LINQ to SQL language to perform database queries (much more easier than SQL sintax).

This article is basically a how-to guide to use a local database in an existing Windows Phone application. The sample application is exactly the same shown in the SQLite article, but using the local database as storage mechanism. Some new sections were append to the end in order to show how to implement relationship between entities.

Adding local database support to the Windows Phone 8 application

Here is the best thing about local database approach. The local database classes are already in any Windows Phone application and there is no need to install any external plugin. If you go to the same section of How_to_use_SQLite_in_Windows_Phone article you can see how this step is extensive in the SQLite approach.

Using the database

To keep the code simple, Every code was implemented in two files, the **MainPage.xaml.cs** and the **TaskDataContext.cs**. But in real world, a good practice should be to separate the Data layer from UI layer, as MVVM pattern says. The creation of a separated `DataContext` subclass is close to this guide.

Creating a DataContext subclass

First of all, you have to create a `DataContext` subclass and define your database structure in there. In this sample project was created a `TaskDataContext` class.

The `DataContext`'s constructor parameter is the url connection to the database. It is similar to the `DB_PATH` parameter in the SQLite sample.

In the new `TaskDataContext` class is defined any needed table. For now, it has defined only the `Tasks` table. It is enough to show the basics of local database mechanism, as insert, delete and update. Analyze the code below with the comments.

```
public class TaskDataContext : DataContext
{
    public TaskDataContext()
        : base("Data Source=isostore:/TaskDB.sdf")
    {
    }

    public Table<Task> Tasks;
}

/// <summary>
/// Task class representing the Task table. Each attribute in the class become one
attribute in the database.
/// </summary>
[Table]
public class Task
{
    /// <summary>
```


(C) Copyright Nokia 2014. All rights reserved.

```
    /// You can create an integer primary key and let the SQLite control it.
    /// </summary>
    [Column(IsPrimaryKey = true, IsDbGenerated = true, DbType = "INT NOT NULL Identity",
CanBeNull = false, AutoSync = AutoSync.OnInsert)]
    public int Id { get; set; }

    [Column]
    public string Title { get; set; }

    [Column]
    public string Text { get; set; }

    [Column]
    public DateTime CreationDate { get; set; }

}
```

Insert data

Now that you have created the `DataContext` subclass, you are able to handle the database. In order to insert a new task record, the developer has to create a `Task` instance and insert it in the `Tasks` attribute of the `TaskDataContext` class. After that, the developer has to call the `SubmitChanges()` method of the `TaskDataContext` class to actually persist the new task in the database file.

```
    // Create a new task.
    Task newTask = new Task()
    {
        Title = TitleField.Text,
        Text = TextField.Text,
        CreationDate = DateTime.Now
    };
    /// Insert the new task in the Task table.
    taskDataContext.Tasks.InsertOnSubmit(newTask);
    taskDataContext.SubmitChanges();
```

Query data

As said before, to perform searches in the local database you have the powerful LINQ to SQL language. This language make much easy to implement queries in the database mainly because you don't have to learn a specific SQL database syntax. LINQ to SQL is nothing more than C# syntax with some improvements.

```
//A basic query. All stored tasks
IEnumerable<Task> retrievedTasks = from Task task in taskDataContext.Tasks select task;
//A more complex query. Tasks that have the 'title' field equals to 'filter_test'
IEnumerable<Task> retrievedTasks = from Task task in taskDataContext.Tasks where
task.Title == "filter_test" select task;
```

Tip: At this point, would be good to reserve some time to test more complex queries.

Working with relationships

When working with databases, a very common task is to manipulate relationship between tables. The local database mechanism provides a very easy way to do that and we are going to see it now.

At this point, the basic sample code should be running fine, but there is only one table at the example. Let's make it more complex

(C) Copyright Nokia 2014. All rights reserved.

before go through the next sections.

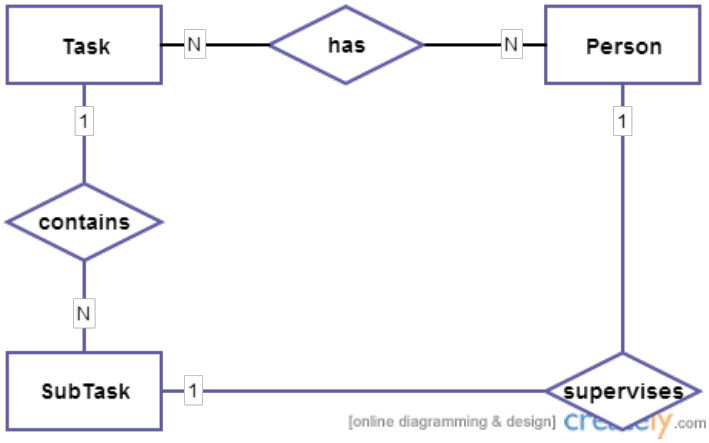
Updating the database structure

We have one main entity named **Task**. Let's create another entities and define that relationship between them.

- **SubTask** - A task can be split in sub tasks. The relationship between the entities **Task** and **SubTask** are of kind One-To-Many (One table's record can be related to many records of other table).
- **Person** - A task can be done by one or more persons. In the same way a person can does one or more tasks. This will create a Many-to-Many relationship between the tables **Task** and **Person** (Many records of one table can be related to many records of other table).

Each sub task can have only one supervisor. This supervisor is a **Person** instance. This relationship between the entities **SubTask** and **Person** is of kind One-To-One (One table's record can be related to one record of other table).

The figure below shows the updated model.



One to One

A 'One to One' relationship happens when one record in a table is related to one record from another table. In our example it happens in the relationship 'Supervises' between the entities 'Task' and 'Person'. A task can have only one supervisor and one supervisor can supervises only one task.

Code

One to one code here

One to Many

A 'One to Many' relationship happens when one record in a table is related to many other records from another table. In our example it happens in the relationship 'Task' -> 'SubTask'. A task can have many sub tasks, but one sub task is related to only one task.

This kind of relationship can be implemented using the `EntitySet` class.

Code

One to many code here

Many to Many

A 'Many to Many' relationship happens when many records in a table are related to many other records from another table. In our example it happens in the relationship 'Task' -> 'Person'. A task can have many workers, and one worker can work in multiple tasks at the same time.

Code

Many to many code here

(C) Copyright Nokia 2014. All rights reserved.

(C) Copyright Nokia 2014. All rights reserved.