Portable Local Data Storage with Lex.DB for Windows Phone 8 and Windows Store Apps

This article describes the technique to build .NET apps targeting several platforms, maximizing code reuse and minimizing number of published packages to maintain.

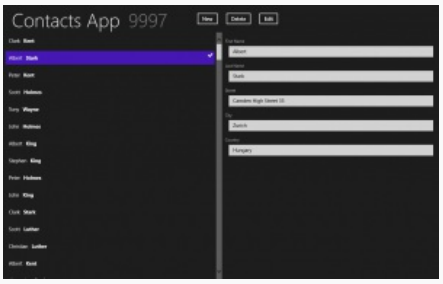
Preface

As an example we'll write a simple, but very responsive Contacts application. For sake of simplicity this example focuses on Windows Phone 8 application, but in similar manner other target platforms like Windows Store and Silverlight 5 could be implemented as well.

To store and retrieve the data my Lex.DB database engine will be used. Lex.DB is superfast, lightweight, serverless, POCO database engine, completely written in C#. Licensed used LGPL with source code available on GitHub, compiled as AnyCPU libraries are distributed via NuGet (package name lex.db). Lex.DB supports Windows Phone 8, Windows RT, Silverlight 5 and .NET 4-4.5 Frameworks. Additional info about Lex.DB.

Contacts App Main Page for Windows Phone 8

Contacts App Details Page for Windows Phone 8



Contacts App for Windows Store

Portable Part

(c) Copyright Nokia 2013. All rights reserved.

We use well known MVVM pattern for this application. Data models, View models and most of application logic will be shared between different versions of the Contacts application. To share this common part of the application, Portable Class Library (PCL) project targeting Windows Phone 8, Windows Store app will be used. Lex.DB supports asynchronous data access, PCL supports async/await using http://nuget.org/packages/Microsoft.Bcl.Async package available on NuGet.

Contact class

So let us define our simple Data Model like this:

```
namespace Contacts.Shared
{
  public class Contact
  {
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string FullName { get { return FirstName + " " + LastName; } }
    public string Email { get; set; }
    public string Phone { get; set; }
    public string Country { get; set; }
    public string City { get; set; }
    public string Street { get; set; }
    public override string ToString()
    {
      return string.Format("{0}, {1}", LastName, FirstName);
    }
  }
}
```

IDataAccess interface

To avoid platform specific references to Lex.DB, we need to abstract our data access logic with IDataAccess interface:

```
namespace Contacts.Shared
{
  public interface IDataAccess
  {
    Task<List<Contact>> GetContacts();
    Task<Contact> GetContact(int contactId);
    Task UpdateContact(Contact contact);
    Task<bool> DeleteContact(Contact contact);
  }
}
```

All methods return Task instances to provide asynchronous execution.

Platform class

Now we need to provide common service location class to use inside portable part of the application. The simplest way is to define a static class with references to service implementation.

```
namespace Contacts.Shared
{
  public static class Platform
  {
    public static IDataAccess DataAccess;
  }
}
```

We will initialize this class in our platform specific version of the application during startup with actual implementation of IDataAccess service. Next step is to define our View Models. However, before we will get into plumbing spree, I recommend you to download KindOfMagic VS.NET 2012 extension from CodePlex or from NuGet and enable it for our portable assembly.

PropertyChangedBase class

Now define base class PropertyChangedBase for all our view models to derive like this:

```
using System;
using System.ComponentModel;
using System.Runtime.CompilerServices;
namespace Contacts.Shared
{
  /// <summary>
  /// Enable automatic RaisePropertyChanged notification generation
  /// </summary>
  public class MagicAttribute : Attribute { }

  /// <summary>
  /// Disable automatic RaisePropertyChanged notification generation
  /// </summary>
  public class NoMagicAttribute : Attribute { }

  [Magic]
  public class PropertyChangedBase : INotifyPropertyChanged
  {
    protected virtual void RaisePropertyChanged([CallerMemberName] string property = "")
    {
      var e = PropertyChanged;
      if (e != null)
        e(this, new PropertyChangedEventArgs(property));
    }

    public event PropertyChangedEventHandler PropertyChanged;
  }
```


(c) Copyright Nokia 2013. All rights reserved.

```
  }
```

KindOfMagic will notice the MagicAttribute applied to the class and during compile-time automatically transform setters of our properties to call RaisePropertyChanged in case underlying value is actually changed. All derived classes will be transformed automatically. To disable transformation, we just need to apply NoMagic attribute to a class or property.

RootViewModel class

The RootViewModel contains just a list of contacts to show in Items property, busy status and details View Model for selected contact.

```
public bool IsLoading { get; private set; }
public ObservableCollection<ContactItemViewModel> Items { get; private set; }
```

Asynchronous LoadItems() method allows us to load the data in background without stalling the UI thread. We also set IsLoading property during load operation, so we can bind it to ProgressRing/BusyIndicator in our View to show user that something happens in background.

```
    async Task<ObservableCollection<ContactItemViewModel>> LoadItemsCore()
    {
      return await TaskEx.Run(async () =>
      {
        var contacts = await Platform.DataAccess.GetContacts();
        var models = from i in contacts select new ContactItemViewModel(i);
        return new ObservableCollection<ContactItemViewModel>(models);
      });
    }

    public async Task LoadItems()
    {
      IsLoading = true;
      try
      {
        Items = await LoadItemsCore();
      }
      finally
      {
        IsLoading = false;
      }
    }
```

Root view model logic is simple, when user selects a contact, DetailItem property is set to ContactDetailViewModel instance. Because UpdateDetails() is asynchronous method, which could potentially take time to load a contact, we also need to set IsLoading flag during loading operation.

```
    public ContactDetailViewModel DetailItem { get; private set; }

    ContactItemViewModel _selectedItem;
    public ContactItemViewModel SelectedItem
    {
      get
      {
        return _selectedItem;
      }
      set
      {
        _selectedItem = value;
        UpdateDetails();
      }
    }

    async void UpdateDetails()
    {
      if (_selectedItem == null)
        DetailItem = null;
      else if (DetailItem == null || DetailItem.Model.Id != _selectedItem.Id)
      {
        IsLoading = true;
        try
        {
          var item = await LoadDetails(_selectedItem.Id);
          DetailItem = new ContactDetailViewModel(item);
        }
        finally
        {
          IsLoading = false;
        }
      }
    }
```

ContactItemViewModel class

ContactItemViewModel is the view model for presentation inside the ListBox/ListView control. It exposes just needed to show properties of the Contact class.

```
  namespace Contacts.Shared
  {
    public class ContactItemViewModel : PropertyChangedBase
    {
      public ContactItemViewModel(Contact contact)
      {
        LoadDataModel(contact);
      }

      public void LoadDataModel(Contact model)
      {
        Id = model.Id;
        FullName = model.FullName;
        FirstName = model.FirstName;
        LastName = model.LastName;
      }

      public int Id { get; set; }
```


(c) Copyright Nokia 2013. All rights reserved.

```
    public string FullName { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
  }
}
```

ContactDetailViewModel class

ContactDetailViewModel looks almost like ContactItemViewModel, but contains Contact instance and provides additional properties to edit, along with IsReadOnly, IsNew and IsDirty properties. IsReadOnly property controls current edit mode for the model. IsNew property indicates whether contact instance is a new one, so when user deletes it, we need just to discard the contact. IsDirty property will be set as soon, as user modifies any of the data properties. This logic implemented inside overridden RaisePropertyChanged method:

```
protected override void RaisePropertyChanged(string property)
 {
   base.RaisePropertyChanged(property);
   switch (property)
   {
     case "IsDirty":
     case "IsReadOnly":
       return;

     default:
       IsDirty = true;
       break;
   }
 }
```

In addition to presentation logic, we pack several methods to allow user to save, delete and cancel changes to contact. LoadDataModel() method reloads original contact information into view model, resets IsDirty flag and restores edit mode:

```
public void LoadDataModel()
{
  FirstName = _model.FirstName;
  LastName = _model.LastName;
  Country = _model.Country;
  Street = _model.Street;
  City = _model.City;

  IsReadOnly = !_isNew;
  IsDirty = false;
}
```

Asynchronous Save and Delete methods call IDataAccess service methods to persist and remove contact information:

```
public async Task Save()
{
  _model.City = City;
  _model.FirstName = FirstName;
  _model.LastName = LastName;
  _model.Country = Country;
  _model.Street = Street;

  await Platform.DataAccess.UpdateContact(_model);

  _isNew = false;
}
```

```
public async Task Delete()
{
  if (!_isNew)
    await Platform.DataAccess.DeleteContact(_model);
}
```

Now when we see the whole picture, let us provide additional methods to our RootViewModel to bind it all together:

RootViewModel class revisited

We provide following public methods to call from our View:

```
public void EditContact()
{
  if (DetailItem == null)
    throw new InvalidOperationException();

  DetailItem.IsReadOnly = false;
}
```

```
public void NewContact()
{
  SelectedItem = null;
  DetailItem = new ContactDetailViewModel(new Contact(), true);
}
```

```
public void RevertContact()
{
  if (DetailItem.IsNew)
    DetailItem = null;
  else
    DetailItem.LoadDataModel();
}
```

```
public async Task SaveContact()
```


(c) Copyright Nokia 2013. All rights reserved.

```
    {
      var selectedItem = _selectedItem;
      var detailItem = DetailItem;
      var isNew = detailItem.IsNew;

      await detailItem.Save();

      if (isNew)
        Items.Add(SelectedItem = new ContactItemViewModel(detailItem.Model));
      else if (selectedItem != null)
        selectedItem.LoadDataModel(detailItem.Model);
    }
```

```
    public async Task DeleteContact()
    {
      var detail = DetailItem;
      if (detail == null)
        throw new InvalidOperationException();

      var selectedItem = _selectedItem;

      await detail.Delete();

      if (selectedItem != null)
        Items.Remove(selectedItem);

      SelectedItem = null;
      DetailItem = null;
    }
```

Conclusion

Windows Phone 8, Windows Store, WPF and Silverlight 5 Contacts applications are available to download File:ContactsApp.zip. They look and behave the same way. 10000 records are generated on start in split second. All used third party tools are open source and free.

I sincerely hope that this example will be a good start for newcomers. Windows Phone 8 platform waits for your amazing apps.

(c) Copyright Nokia 2013. All rights reserved.

(c) Copyright Nokia 2013. All rights reserved.