Contents

The **Windows Phone app tour** provides an introduction to developing a Windows Phone app with DevForce 2012.

What you'll learn:

- How to write n-tier LINQ queries in DevForce
- How to handle asynchronous queries and saves
- Simple navigation techniques in Windows Phone apps
- Abstracting data management into a DataService

This version of the tour uses a **Code First** model. See **here** for the Database First version.

- **Platform:** Windows Phone
- **Language:** C#
- **Download:** Windows Phone Dev Tour (Code First)
- **Prerequisites:** Windows Phone 8 SDK

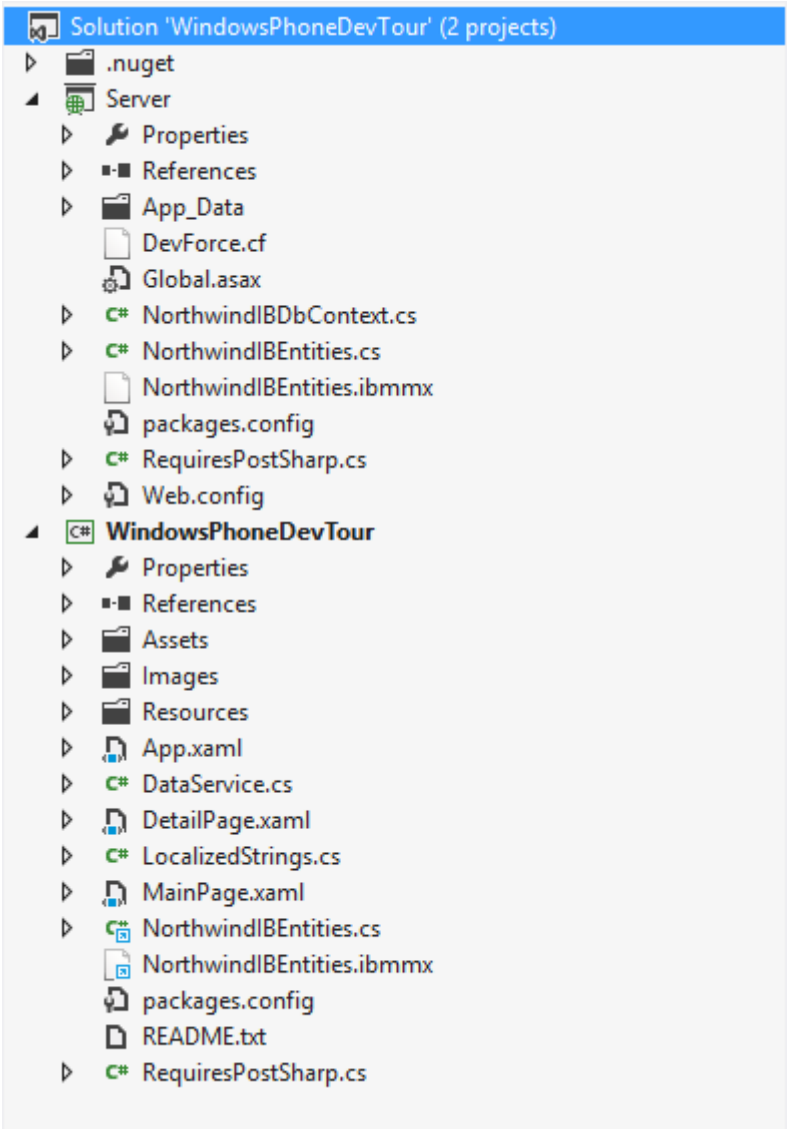
Getting started

The Windows Phone app tour demonstrates a simple two page master/detail phone application. The first page lists all customers in the NorthwindIB sample database and provides search capability. Tap a customer and it takes you to the detail page where you can edit the customer and save. Tapping the back button takes you back to the list.

You must enable NuGet package restore within Visual Studio in order to restore the required **DevForce NuGet packages** and other dependencies. The first time you build the application, the dependent NuGet packages are installed.

The Windows Phone app tour includes a SQL CE version of the *NorthwindIB* sample database.

Let's first take a look at the solution structure:



A DevForce Windows Phone application is an n-tier application, and uses an EntityServer to perform all backend data access. We see that reflected in the solution structure: *WindowsPhoneDevTour* is the client application project, and *Server* is the web application project hosting the *EntityServer*.

Next, let's look at the model.

We've added the DevForce Code First NuGet package to both projects. This package adds the necessary DevForce and PostSharp dependencies, and is always required in a model project, *Server* here; and in the "linked" client project, *WindowsPhoneDevTour* here.

File *NorthwindIBEntities.cs* contains our model: here a single entity *Customer*, and a custom *EntityManager*.

```
[ProvideEntityAspect]
[DataContract(IsReference = true)]
public abstract class BaseEntity {

...
}

/// <summary>
/// The domain-specific EntityManager for your domain model.
/// </summary>
public partial class NorthwindIBEntities : EntityManager {

  public EntityQuery<Customer> Customers { get; set; }
}

[DataContract(IsReference = true)]
public partial class Customer : BaseEntity {

...
}
```

See Apply DevForce AOP attributes, Add a custom EntityManager, and Add a custom DbContext for more information on writing a Code First model.

Our custom *EntityManager* and *Customer* classes in file *NorthwindIBEntities.cs* are also required in the Windows Phone project, so the file has been added as a link. When we build the solution, a metadata file, here *NorthwindIBEntities.ibmmx* is added to the *Server* project, and also added as a link to the *WindowsPhoneDevTour* project. In both cases, the file is given a build action of "Embedded Resource". See Generate metadata for more information on how to work with *ibmmx* files in DevForce.

Overview

1. In order to query data from the server, you need to specify the URL, port and service name of the *EntityServer*. You can do this either with an *app.config* file, or programmatically, as shown in the application constructor (in App.xaml.cs):

```
public App()
{
 ...
   IdeaBladeConfig.Instance.ObjectServer.RemoteBaseUrl = "http://xx.xx.xx.xx";
   IdeaBladeConfig.Instance.ObjectServer.ServerPort = 80;
   IdeaBladeConfig.Instance.ObjectServer.ServiceName = "WindowsPhoneDevTour/EntityService.svc";
}
```

Important: The phone emulator is a separate virtual machine, and as such a server address such as http://localhost will not work. You can instead use the IP address of the host PC. Also, by default IIS Express only allows connections to *localhost*. To work around this you can modify the applicationhost.config file for the IIS Express application, or you can instead use IIS as your web server. These options are explained in more detail **here**.

2. MainPage.xaml is the master/search page. It displays all customers and provides search capability by customer name.

Looking at the code, notice the Start method, called from the *OnNavigatedTo* handler:

```
public async void Start()
{
  try
   {
      IsBusy = true;
      var customers = await DataService.Instance.GetAllCustomersAsync();
      Customers = new ObservableCollection<Customer>(customers);
   }
  finally
   {
      IsBusy = false;
   }
}
```

A few things to note here: data retrieval is asynchronous using the *async/await* keywords, and data management activities are performed by a *DataService* class.

The *IsBusy* flag is used, via data binding, to display a progress indicator:

```
<shell:SystemTray.ProgressIndicator>
  <shell:ProgressIndicator IsIndeterminate="true" IsVisible="{Binding IsBusy}" />
</shell:SystemTray.ProgressIndicator>
```

Let's also look at the *search* button event handler:

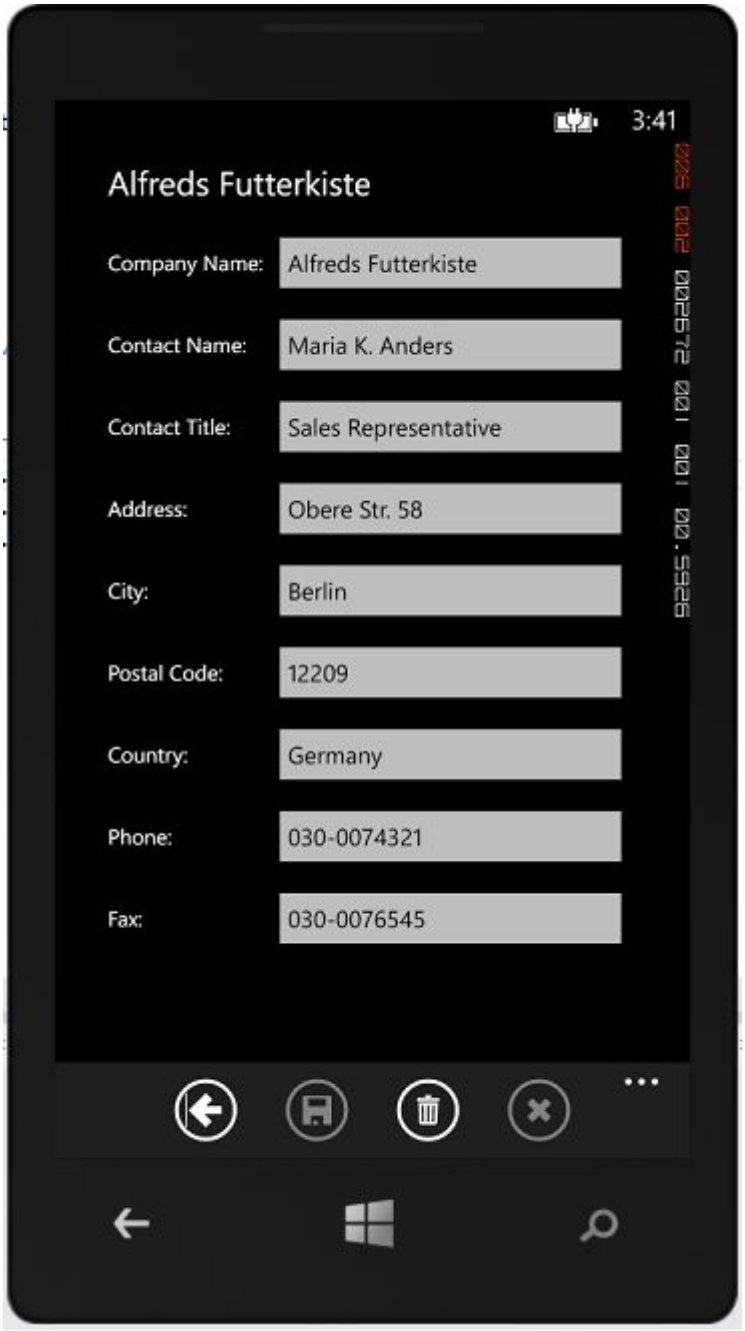
```
private async void Search(object sender, RoutedEventArgs e)
{
  try
   {
      IsBusy = true;
     if (string.IsNullOrEmpty(SearchText))
      {
         Start();
        return;
      }
      var customers = await DataService.Instance.FindCustomersAsync(SearchText);
      Customers = new ObservableCollection<Customer>(customers);
   }
  finally
   {
      IsBusy = false;
   }
}
```

Again, we see asynchronous data retrieval using the *DataService* which encapsulates the DevForce LINQ queries.

3. When a customer is selected, the *NavigateToDetailPage* method is called. This uses the NavigationService to navigate to the detail page. We're also passing an *id* parameter containing the ID of the selected customer.

```
private void NavigateToDetailPage() {
  NavigationService.Navigate(new Uri("/DetailPage.xaml?id=" + HttpUtility.UrlEncode(SelectedCustomer.CustomerID.ToString()),
UriKind.Relative));
}
```

4. *DetailPage.xaml* shows customer details, and allows editing with save, delete and undo capabilities.



Looking at the code, we see the *Start* method is called from the *OnNavigatedTo* handler, which receives the ID of the customer to be displayed in the *NavigationContext.QueryString*.

```
protected override void OnNavigatedTo(NavigationEventArgs e) {
...
  var id = NavigationContext.QueryString["id"];
  Start(new Guid(id));
...
}
```

```
public async void Start(Guid customerId)
{
    Customer = await DataService.Instance.GetCustomerAsync(customerId);
}
```

And here's the *Save* button handler, which asks the *DataService* to perform a save, and will display a message upon failure:

```
private async void Save(object sender, EventArgs e)
{
 try {
```

```
    IsBusy = true;
    await DataService.Instance.SaveAsync();
  } catch (EntityManagerSaveException err) {
    MessageBox.Show("Save failed: " + err.Message);
  } finally {
    IsBusy = false;
  }
}
```

One thing worth noting, the *ApplicationBar* does not support data binding, so we've used a bit of a hack to associate hidden *CheckBox* controls, which are data bound, to the buttons of the *ApplicationBar*, and toggle the *IsEnabled* status of each button when the status of its corresponding checkbox changes.

5. The *DataService.cs* contains a singleton data service used throughout the application. Both the *MainPage* and *DetailPage* work directly with the *DataService*, which encapsulates the DevForce EntityManager and performs all queries and saves.

Here's a sample method, this one to retrieve all customers:

```
public Task<IEnumerable<Customer>> GetAllCustomersAsync()
{
  return _entityManager.Customers.OrderBy(x => x.CompanyName).ExecuteAsync();
}
```

The *Task* returned is awaited upon by the caller, the *MainPage Start* method we saw above.

Additional resources

- Windows Phone Dev Center