Contents

- Problem
- Solution
  - From File | New to DevForce Entity Model
    - Create a DevForce model
  - Install MVVM Light using NuGet
  - Add the data service
  - Write the ViewModel
  - Modify the ViewModelLocator
  - Design the view
  - See it Blend
- Prerequisites

Laurent Bugnion's MVVM Light is one of the popular frameworks for building MVVM-style applications. It's lightweight, open-source, and you can find it here.

- **Platform:** Silverlight
- **Language:** C#, VB
- **Download:** MVVM Light with DevForce
  You'll need to unblock the MVVM light .dlls. If you don't, VS will give you a friendly reminder to do so.

Problem

A number of DevForce customers have asked for some guidance on bringing DevForce and MVVM together.

Solution

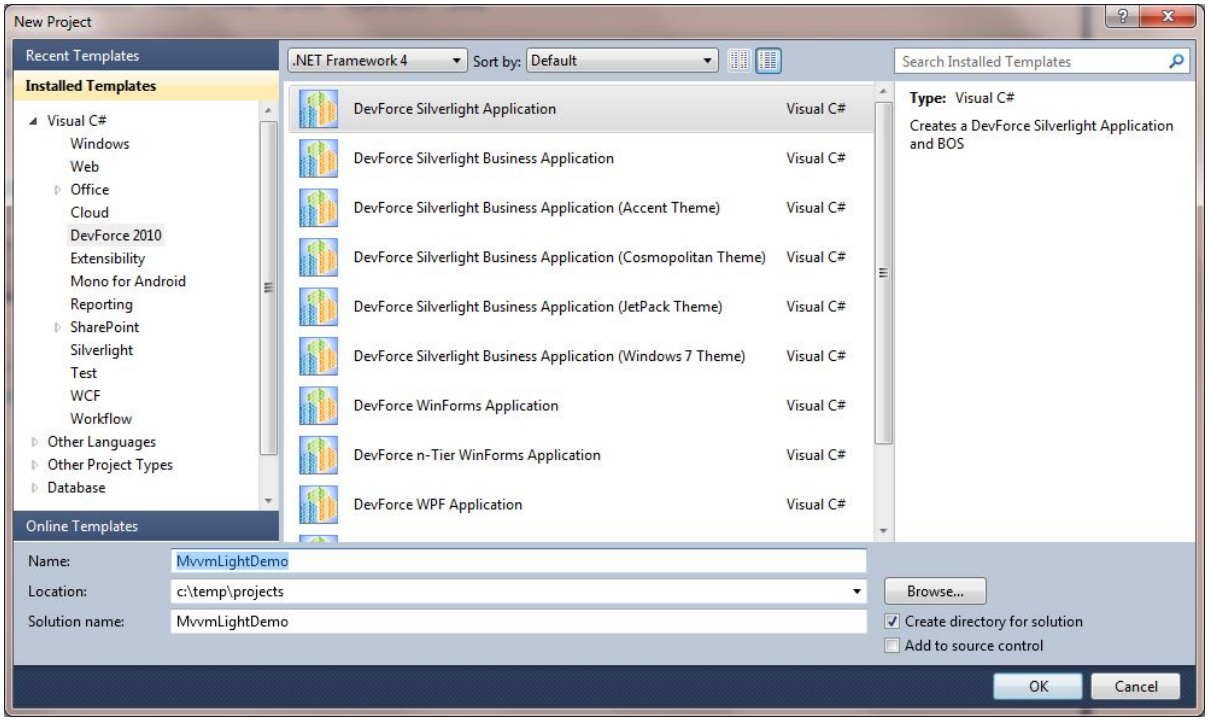
The code sample, and companion video, illustrates:

- DevForce and MVVM Light working together
- Repository/DataService pattern
- Design data repository to facilitate view development with design tools
- Building a view in VS "Cider" design tool

For the best learning experience, we recommend you watch the video and follow along with the sample code of the completed solution.

From File | New to DevForce Entity Model

Let's begin by creating a new project. First, select **File | New | Project** from the main menu in Visual Studio, and select the DevForce Silverlight Application template. Name the solution "MvvmLightDemo".



Create a DevForce model

We'll add the entity model directly to our web and Silverlight projects in this example. This is a simplistic approach, and for long term maintainability we recommend separate .NET and SL model projects to hold the model, but that's a lesson for another time!

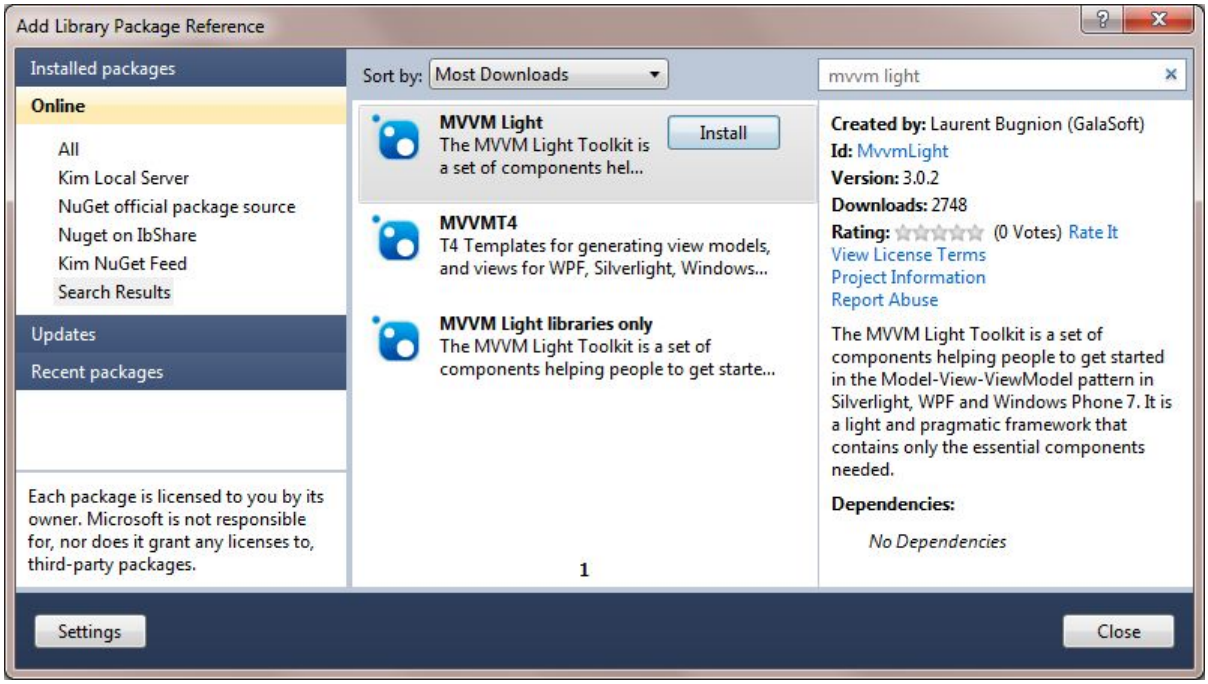
Right-click on the MvvmLightDemoWeb project and select **Add | New Item**. In the Add New Item dialog, you will find the ADO.NET Entity Data Model under Data. We'll name ours *Model.edmx* and click Add. (You can also locate the template quickly by simply typing "ADO" into the Search Installed Templates field in the upper right of the dialog.)

Generate from the NorthwindIB database that is provided as part of the DevForce installation and accept defaults. We'll include only the *Customer* table in this simple model. Once the wizard completes you'll see the DevForce-generated model in the web project and a link to it in the Silverlight project.

Install MVVM Light using NuGet

We use the NuGet package manager to install MVVM Light. If you don't have NuGet installed you can directly download and install *MVVM Light*, but we highly recommend using NuGet since it will simplify the process, and many other vendors are now using NuGet too.

Right click on the MvvmLightDemo project and select "Add Library Package Reference ..." from the menu. Search for "Mvvm Light", and click "Install" once found.



This will install the *MVVM Light* assemblies into the Silverlight project, add a "ViewModel" folder with *MainvViewModel* and *ViewModelLocator* classes, and modify the application resources in the *App.xaml* to add the *ViewModelLocator* as a resource. We'll discuss these more below.

Add the data service

Next we'll add a simple repository/data service to access our entity model. Add a code or class file to the MvvmLightDemo Silverlight project and name it *DemoDataService*.

Add an *IDemoDataService* interface. Here only a *LoadCustomers* method is needed.

```
using System;
using System.Collections.Generic;
using IdeaBlade.EntityModel;
namespace MvvmLightDemo {
 public interface IDemoDataService {
  void LoadCustomers(
      Action<IEnumerable<Customer>> success = null,
      Action<Exception> fail = null);
 }
}
```

```
Imports System
Imports System.Collections.Generic
Imports IdeaBlade.EntityModel
Namespace MvvmLightDemo
Public Interface IDemoDataService
 Sub LoadCustomers(Optional ByVal success As Action(Of IEnumerable(Of Customer)) = Nothing, _
  Optional ByVal fail As Action(Of Exception) = Nothing)
End Interface
End Namespace
```

Next add a *DemoDataService* class. This class will serve as the run time data service for this sample. It will be responsible for querying all Customers using the EntityManager.

```
public class DemoDataService : IDemoDataService {
 public DemoDataService() {
   Manager = new NorthwindIBEntities();
 }
 protected NorthwindIBEntities Manager { get; set; }
 public void LoadCustomers(
     Action<IEnumerable<Customer>> success = null,
     Action<Exception> fail = null) {
   Manager.Customers
```

```
    .OrderBy(c => c.CompanyName)
    .ExecuteAsync(op => {
     if (op.CompletedSuccessfully) {
       if (null != success) {
         success(op.Results);
       }
     } else {
      if (null != fail) {
        op.MarkErrorAsHandled();
        fail(op.Error);
      }
     }
    });
  }
}
```

```
Public Class DemoDataService
Implements IDemoDataService
Public Sub New()
 Manager = New NorthwindIBEntities()
End Sub
Protected Property Manager() As NorthwindIBEntities
Public Sub LoadCustomers(Optional ByVal success As Action(Of IEnumerable(Of Customer)) = Nothing, _
  Optional ByVal fail As Action(Of Exception) = Nothing)
 Manager.Customers.OrderBy(Function(c) c.CompanyName).ExecuteAsync(Sub(op)
 If op.CompletedSuccessfully Then
  If Nothing IsNot success Then
    success(op.Results)
  End If
 Else
  If Nothing IsNot fail Then
    op.MarkErrorAsHandled()
    fail(op.Error)
  End If
 End If
End Sub)
End Sub
End Class
```

Finally let's add a *DesignDemoDataService* class for design-time support. This service will be responsible for supplying design-time data.

```
public class DesignDemoDataService : IDemoDataService {
 public void LoadCustomers(
       Action<IEnumerable<Customer>> success = null,
       Action<Exception> fail = null) {
  if (null != success) success(CreateDesignCustomers());
 }
 private static IEnumerable<Customer> CreateDesignCustomers() {
   var custs = new List<Customer>();
  // First customer is for full detail customer design
  custs.Add(
     new Customer {
       CustomerID = Guid.NewGuid(),
       CompanyName = "The Design Width Company Name",
       Address = "123 Main Street",
       City = "Anytown",
       ContactName = "Harry Fidurcci",
       Phone = "(510) 555 1212 x10",
       Country = "France",
     }
   );
  // 10 dummy customers for listboxes
  var custCounter = 1;
  while (custCounter <= 10) {
    var cust = new Customer {
      CustomerID = Guid.NewGuid(),
      CompanyName = "Customer " + custCounter++,
      Country = "USA"
    };
    custs.Add(cust);
```

```
    }
   return custs;
  }
}
```

```
Public Class DesignDemoDataService
Implements IDemoDataService
Public Sub LoadCustomers(Optional ByVal success As Action(Of IEnumerable(Of Customer)) = Nothing, _
  Optional ByVal fail As Action(Of Exception) = Nothing)
  If Nothing IsNot success Then
    success(CreateDesignCustomers())
  End If
End Sub
Private Shared Function CreateDesignCustomers() As IEnumerable(Of Customer)
Dim custs = New List(Of Customer)()
  ' First customer is for full detail customer design
  custs.Add(New Customer With {.CustomerID = Guid.NewGuid(), _
     .CompanyName = "The Design Width Company Name", .Address = "123 Main Street", _
     .City = "Anytown", .ContactName = "Harry Fidurcci", _
     .Phone = "(510) 555 1212 x10", .Country = "France"})
  ' 10 dummy customers for listboxes
  Dim custCounter = 1
  Do While custCounter <= 10
  Dim cust = New Customer With {.CustomerID = Guid.NewGuid(), _
          .CompanyName = "Customer " & custCounter, .Country = "USA"}
   custCounter += 1
   custs.Add(cust)
  Loop
  Return custs
End Function
End Class
```

We've now got a simple data service providing *LoadCustomers* support at both design and run time. Now let's write something to use it.

Write the ViewModel

A class called *MainViewModel* was created for us when we added *MVVM Light*. We'll add our own properties and logic to it.

First, let's make sure we've got the right *usings* (or *Imports*) statements:

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using GalaSoft.MvvmLight;
```

```
Imports System
Imports System.Collections.Generic
Imports System.Collections.ObjectModel
Imports System.Linq
Imports GalaSoft.MvvmLight
```

Our view will display Customers, a status message, and detail on each selected Customer. We'll add these properties to the ViewModel so the view can bind to them:

```
public ObservableCollection<Customer> Customers { get; private set; }
private string _message;
public string Message
{
  get { return _message; }
  set { _message = value; RaisePropertyChanged("Message"); }
}
private Customer _currentCustomer;
public Customer CurrentCustomer
{
  get { return _currentCustomer; }
  set { _currentCustomer = value; RaisePropertyChanged("CurrentCustomer"); }
}
```

```
Private privateCustomers As ObservableCollection(Of Customer)
Public Property Customers() As ObservableCollection(Of Customer)
Get
```

```
 Return privateCustomers
End Get
Private Set(ByVal value As ObservableCollection(Of Customer))
  privateCustomers = value
End Set
End Property
Private _message As String
Public Property Message() As String
  Get
   Return _message
  End Get
  Set(ByVal value As String)
    _message = value
    RaisePropertyChanged("Message")
  End Set
End Property
Private _currentCustomer As Customer
Public Property CurrentCustomer() As Customer
  Get
   Return _currentCustomer
  End Get
  Set(ByVal value As Customer)
    _currentCustomer = value
    RaisePropertyChanged("CurrentCustomer")
  End Set
End Property
```

We also want to inject a data service into the ViewModel when constructed, so let's remove the default constructor generated for us and add a constructor which accepts an *IDemoDataService*.

```
private IDemoDataService _dataService;
public MainViewModel(IDemoDataService dataService)
{
    _dataService = dataService;
}
```

```
Public Sub MainViewModel(ByVal dataService As IDemoDataService)
 _dataService = dataService
End Sub
```

We also want to initialize and load our data when the constructor is called, so let's also add a call to the *LoadCustomers* method on our data service, along with "success" and "failure" callbacks. These will be called based on whether the load succeeds or fails.

```
public MainViewModel(IDemoDataService dataService)
{
    _dataService = dataService;
    Customers = new ObservableCollection<Customer>();
    Message = "... Loading ...";
    _dataService.LoadCustomers(CustomersLoaded, CustomerLoadFailed);
}
private void CustomersLoaded(IEnumerable<Customer> customers)
{
   foreach (var customer in customers)
    {
       Customers.Add(customer);
    }
    CurrentCustomer = customers.FirstOrDefault();
    Message = "Customers loaded.";
}
private void CustomerLoadFailed(Exception except)
{
    Message = "Customer Load Failed: " + except.Message;
}
```

```
Public Sub New(ByVal dataService As IDemoDataService)
_dataService = dataService
Customers = New ObservableCollection(Of Customer)()
Message = "... Loading ..."
_dataService.LoadCustomers(AddressOf CustomersLoaded, AddressOf CustomerLoadFailed)
End Sub
Private Sub CustomersLoaded(ByVal customers As IEnumerable(Of Customer))
```

```
  For Each customer In customers
Customers.Add(customer)
  Next customer
CurrentCustomer = customers.FirstOrDefault()
Message = "Customers loaded."
End Sub
Private Sub CustomerLoadFailed(ByVal except As Exception)
Message = "Customer Load Failed: " & except.Message
End Sub
```

Modify the ViewModelLocator

The *ViewModelLocator* is another class auto-generated when you added the *MVVM Light* package. We've made a few changes to make the "_main" variable non-static, and added a variable for our "_dataService". Here in the locator's constructor is where we've moved the check for *DesignMode*, and we've added logic to construct either our *DesignDemoDataService* or *DemoDataService* as appropriate. We pass this service along when we construct the *MainViewModel*.

```
private MainViewModel _main;
private IDemoDataService _dataService;
public ViewModelLocator()
{
  if (ViewModelBase.IsInDesignModeStatic)
   {
     _dataService = new DesignDemoDataService();
   }
  else
   {
     _dataService = new DemoDataService();
   }
   _main = new MainViewModel(_dataService);
}
```

```
Private _main As MainViewModel
Private _dataService As IDemoDataService
Public Sub New()
  If ViewModelBase.IsInDesignModeStatic Then
  _dataService = New DesignDemoDataService()
  Else
_dataService = New DemoDataService()
  End If
_main = New MainViewModel(_dataService)
End Sub
```

The *MVVM Light* installation also added the locator as an application resource to our App.xaml.

```
<Application.Resources>
  <vm:ViewModelLocator x:Key="Locator" d:IsDataSource="True" />
</Application.Resources>
```

Design the view

First, to tie our view to our view model, in the *MainPage.xaml* we set the *DataContext* for the view to the *Main* property of the locator.

```
DataContext="{Binding Source={StaticResource Locator}, Path=Main}"
```

Next we create a simple grid layout with three rows. We'll add a title, and a *TextBlock* bound to the *Message* property we created on our *MainViewModel*:

```
<TextBlock Name="textBlock1" Text="Mvvm Light Demo" Grid.ColumnSpan="2" FontSize="40" TextAlignment="Center" />
<TextBlock Grid.Row="2" Name="textBlock2" Text="{Binding Path=Message}" Margin="8,0,0,0" />
```

The second row will contain another grid, holding a *ListBox* of customers in the left column, and a few simple customer properties on the right.

Here's the *ListBox*. Note its *ItemsSource* is bound to the *Customers* property of our *MainViewModel*, while the *SelectedItem* is bound to the *CurrentCustomer*.

```
<ListBox x:Name="listBox1"
        ItemsSource="{Binding Customers}"
        SelectedItem="{Binding Path=CurrentCustomer, Mode=TwoWay}"
```

```
        ItemTemplate="{StaticResource customerListBoxItemsTemplate}"
        Margin="0,0,2,0"  />
```

The *ItemTemplate* determines which customer properties are displayed in the *ListBox*.

```
<UserControl.Resources>
   <ResourceDictionary>
      <DataTemplate x:Key="customerListBoxItemsTemplate">
         <Grid>
            <TextBlock Text="{Binding CompanyName}"
            TextTrimming="WordEllipsis" />
         </Grid>
      </DataTemplate>
   </ResourceDictionary>
</UserControl.Resources>
```

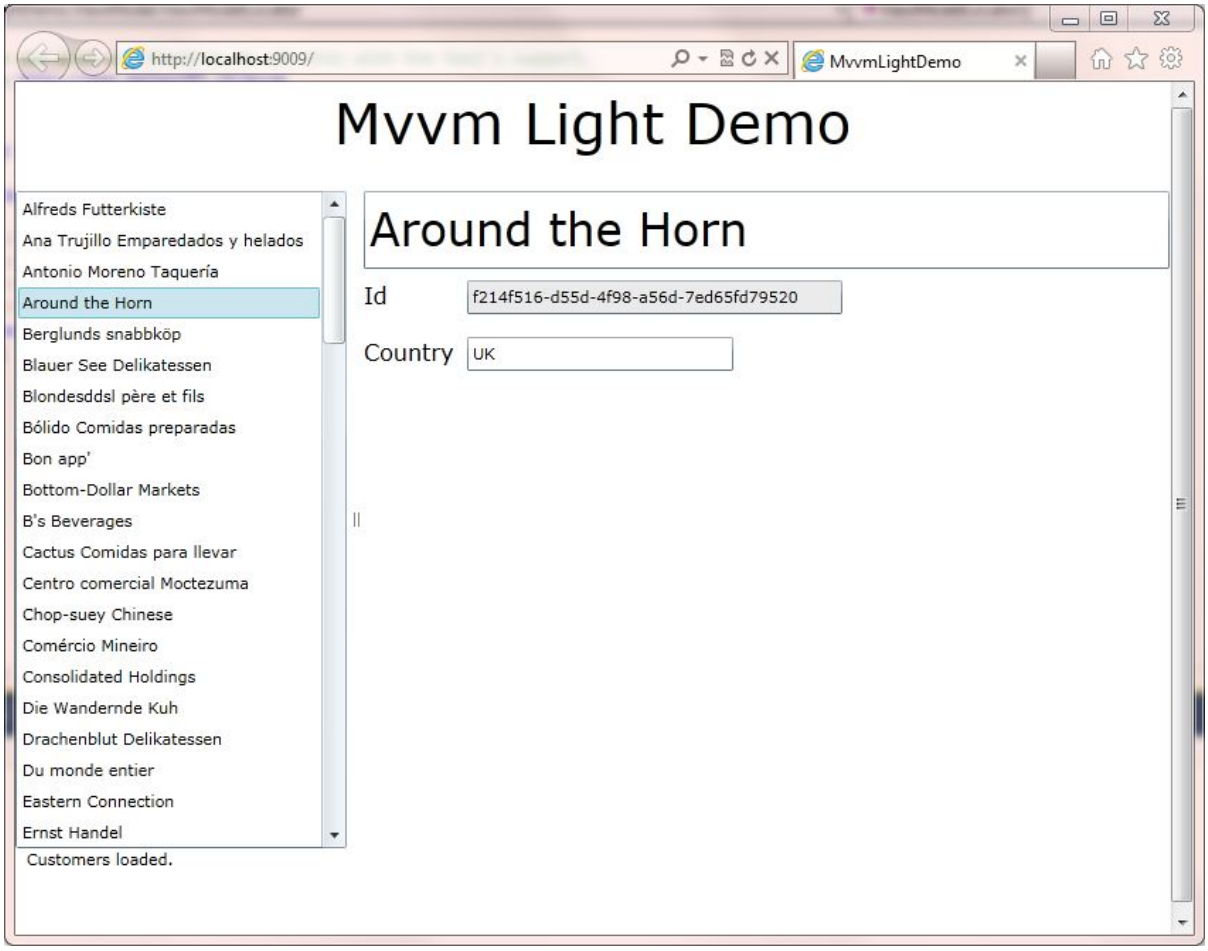
Finally, another grid bound to *CurrentCustomer* will display a few simple customer properties.

```
<TextBox Grid.ColumnSpan="2" Text="{Binding CompanyName, Mode=TwoWay}" FontSize="32" TextWrapping="Wrap" />
<TextBox Grid.Column="1" Grid.Row="1" Text="{Binding Path=CustomerID, Mode=TwoWay}" TextWrapping="NoWrap"
  VerticalAlignment="Center" IsReadOnly="True" Width="265" HorizontalAlignment="Left" />
<TextBox Grid.Column="1" Grid.Row="2" Text="{Binding Country, Mode=TwoWay}" TextWrapping="Wrap"
  HorizontalAlignment="Left" Width="188" VerticalAlignment="Center" Margin="0" />
```

See the sample solution for the full XAML markup or follow the video for complete directions.

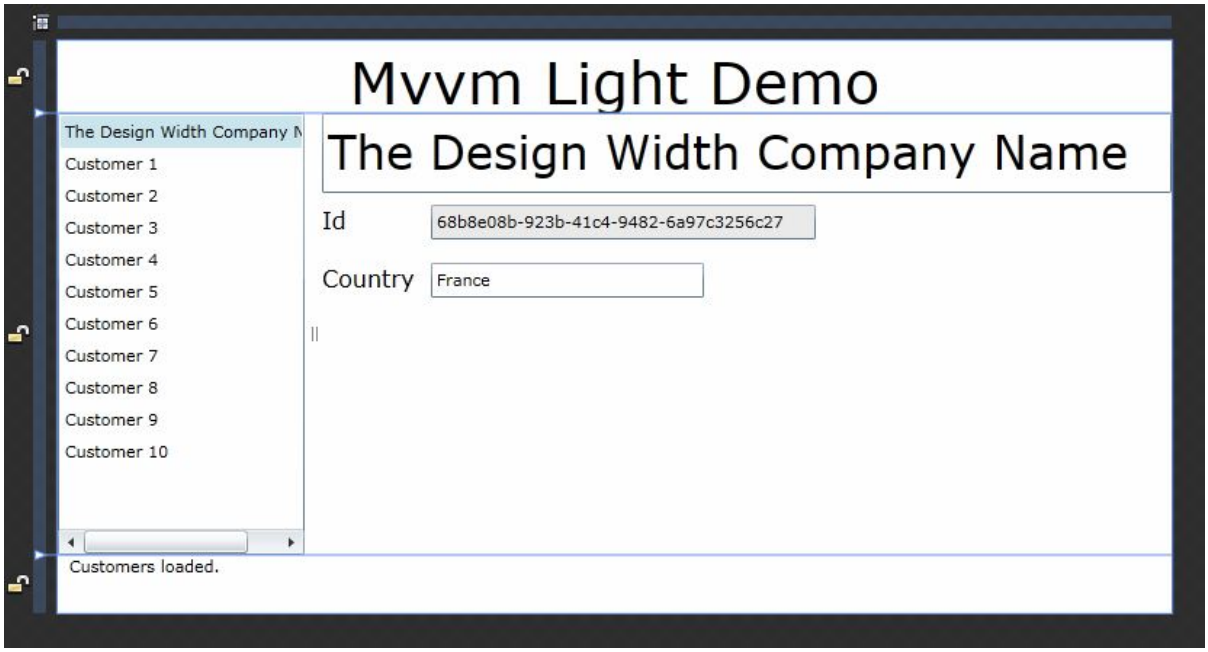
As you've been making these changes to *MainPage*, you've probably noticed that the design time data was displayed in the design surface for bound properties. Wahoo! The design time data is coming from our *DesignDemoDataService*, instantiated by the *ViewModelLocator* when in design mode and passed to the *MainViewModel*. Remember the view's *DataContext* is set to the *MainViewModel* via the locator.
Let's be brave and run the application now too to see our run time data. Real customers should have been loaded, and we can select customers in the *ListBox* to see additional information. Since it's run time, the *ViewModelLocator* created our *DemoDataService*, which is querying the database to load data.

Here's our running application:



See it Blend

With the separation of the view and view model, and the inclusion of design time data, we can easily use the Visual Studio designer (Cider) or design tools such as Expression Blend to continue improving the design and usability of the view.



Prerequisites

The MVVM Light Toolkit emphasizes the "blendability" of your application, including the creation of design-time data and separation of your view from your model. The toolkit contains much more than shown here. Check it out!

We've used NuGet to install *MVVM Light*, although it's not required and you can manually install *MVVM Light* if desired. NuGet will save you several steps, and you'll quickly become a fan.