Contents

Part 1: From "File New" to a working DevForce application - In Part 1 of this series we'll see how quickly you can get started writing a Silverlight application using DevForce. In this tutorial we'll create a new application, add an entity model, and load query results into a grid.

This video was recorded using DevForce 2010. Download the sample code below for updated techniques in DevForce 2012.

- **Platform:** Silverlight
- **Language:** C#, VB
- **Download:** Tour of DevForce Silverlight

Getting started

Let's begin by creating a new project. We'll be using Visual Studio 2012 and targeting Silverlight 5.

If you haven't yet installed DevForce 2012 you can begin with the standard *Silverlight Application* template and then add the **DevForce NuGet packages**. You'll need to install the DevForce 2012 Client package to both projects, and the DevForce 2012 Server package to the web application project.

First, select File | New | Project from the main menu.

You'll notice that several project templates are available under DevForce 2012. Select DevForce Silverlight Application, name your project "SimpleSteps", and click OK.

If you select a targeted framework other than .NET Framework 4.5 at the top of the New Project dialog, the DevForce templates will not be visible.
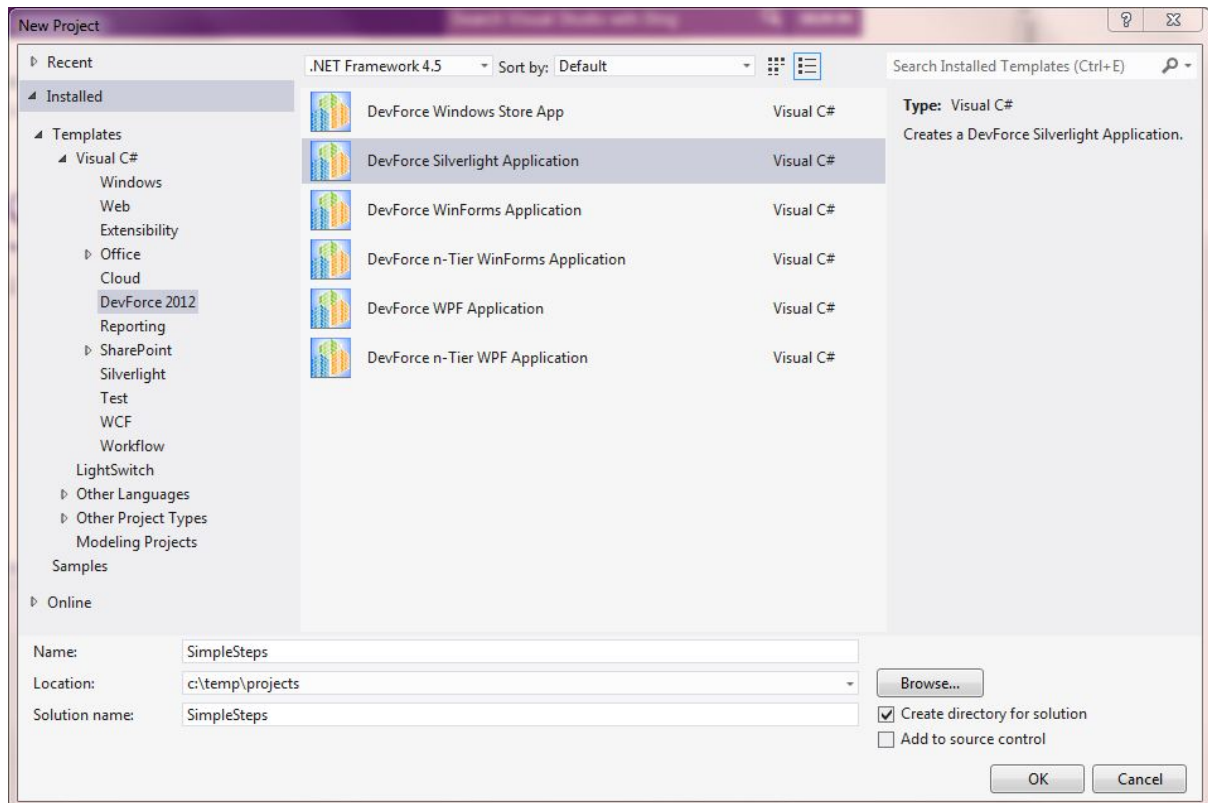
The template creates two projects for us.
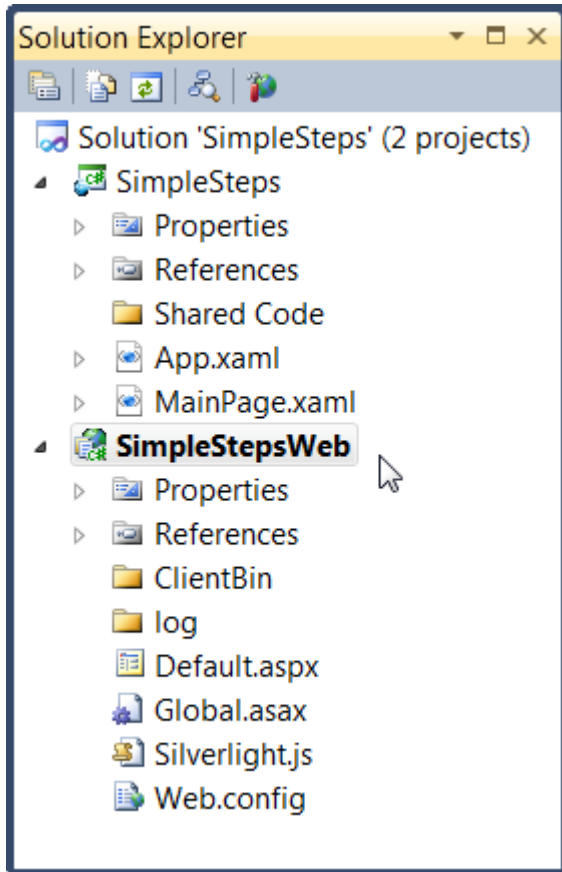
The first one is a Silverlight 5 project named SimpleSteps.

The second one is an ASP.NET project named SimpleStepsWeb. It will host our Silverlight application as well as the DevForce application server.

Create an entity model

In this tutorial, we are going to use the NorthwindIB database that is provided with DevForce samples. You can find the database here.

We'll use an Entity Framework Entity Data Model for accessing data. Right-click on the SimpleStepsWeb 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 NorthwindIB 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.
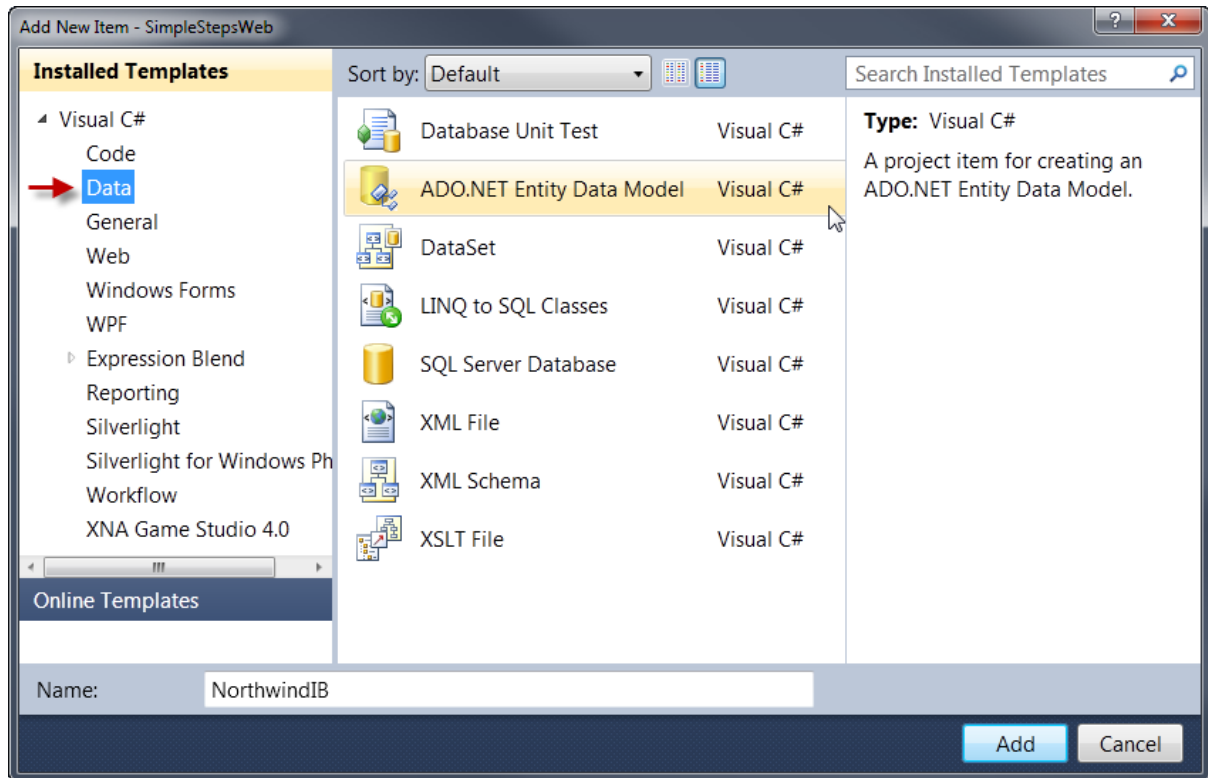
Next we'll select Generate from database and then click Next.

You'll need to create a connection to NorthwindIB that is specific to your environment. Click New Connection to set one up.

Note that, by default, the wizard will save your entity connection settings under the name "NorthwindIBEntities". This will also become the name of the EntityManager you will use in your application to retrieve and cache entities. Accept this default name, and click Next to continue

The Entity Data Model Wizard wants to know which objects to include in the model. For this tutorial we'll just create entities backed by Tables. Expand the Tables node and select the following tables, leaving the others unchecked:

- Customer
- Employee
- Order
- OrderDetail
- Product
- Supplier

We'll leave the rest of the settings at their default values and click Finish.

You might receive a Security Warning at this point. Don't worry. DevForce is using Visual Studio's templating system to generate code based upon the Entity Data Model we just created. We recommend checking Do not show this message again and then clicking OK.
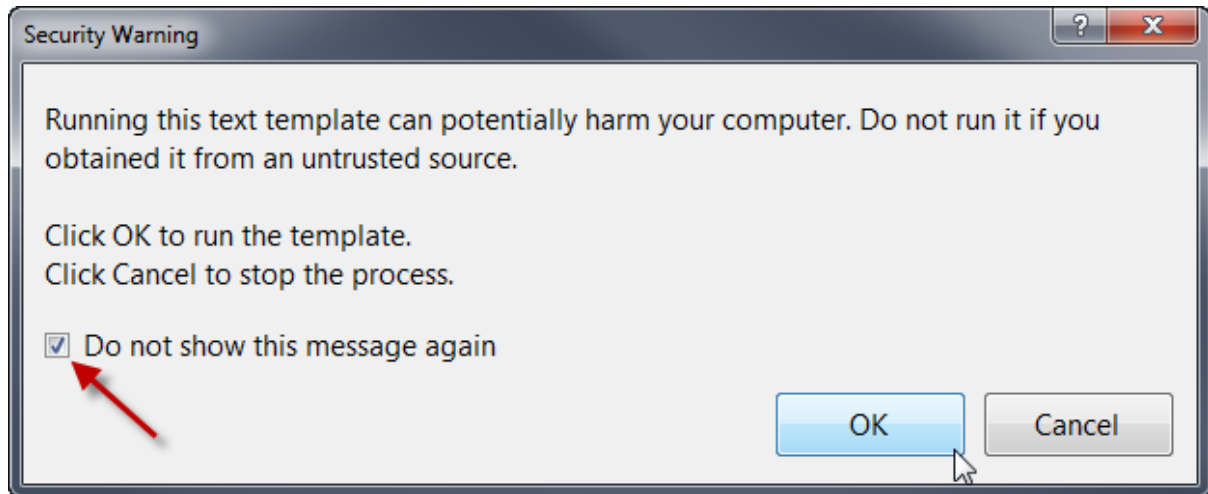
After the template completes the visual designer opens and we can explore the Entity Data Model or "EDM". We'll come back to it in a moment.

You must now "enable" DevForce for this model in order for DevForce to generate the entity model code. See Enable DevForce for more information.
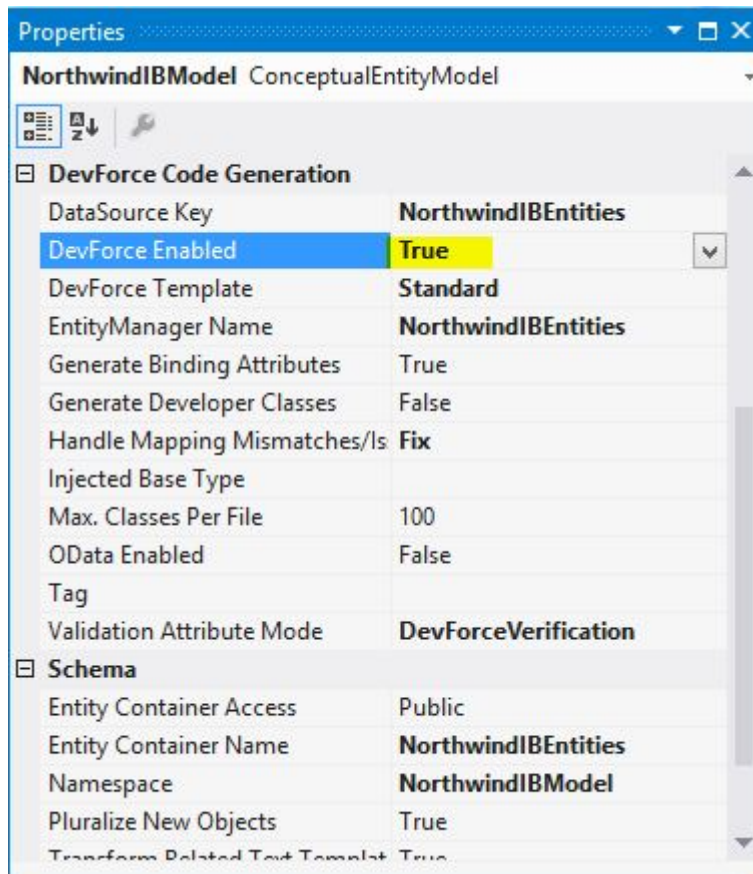
With DevForce enabled, the DevForce T4 template that guides the generation of our business model is added to SimpleStepsWeb. Notice that its name matches the name of our EDM and has the file extension "tt". If you expand the template node, you'll see a file named NorthwindIB.IB.Designer.cs. This is the code that was generated by DevForce using the template.
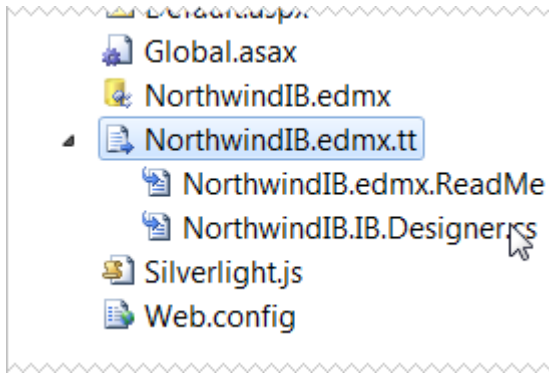
NorthwindIB.IB.Designer.cs is also linked to our Silverlight project. You can see it in the Shared Code folder. Note the *shortcut* icon overlay indicating that the file is linked rather than being copied.
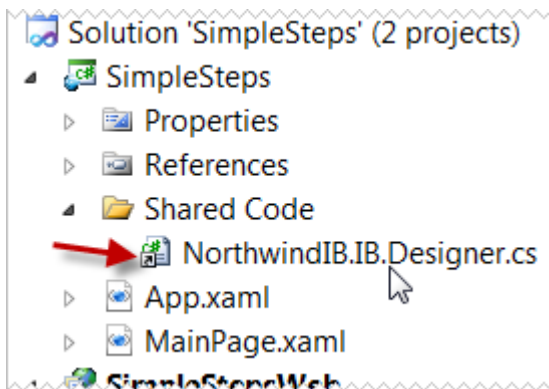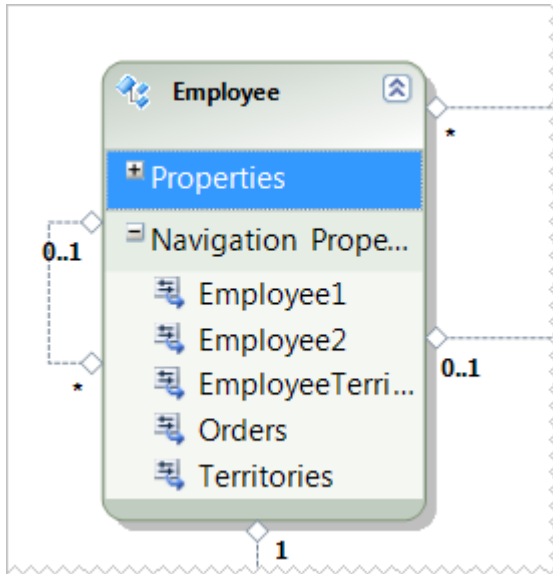

You might be wondering why the file is linked instead of having the Silverlight project simply reference the Web project. There are several reasons:

1. The Web project targets the .NET Framework and Silverlight projects cannot reference .NET Framework projects.
2. We don't want to copy the file, because it is likely to change and the copies would very quickly get out of sync.
3. As we develop our app, we are likely to include code or configuration settings in the web project that are only for server-side use, and which we don't want exposed on the client machine.

But sharing the generated code file by means of a link allows the class definitions to be shared between both projects, without creating undesirable side-effects. With our entities defined in a shared file that will be compiled and included in both our server- and client-side assemblies, we will be able to access and use our entities on both the client (Silverlight) and the application server.

Let's go back to the visual designer for our EDM. The diagram represents a conceptual view of the model entities and their associations. This view was generated based on the schema of the NorthwindIB database. DevForce uses this conceptual model to generate the corresponding .NET entity classes.

Entity Framework does a fairly good job of naming these items to meet our expectations. If we examine the Navigation Properties[1] on the Employee class, we see that it created a property, Orders, that returns a collection of Order instances.

However, in some cases the EDM designer really didn't know what to do. Notice the properties on Employee that are named Employee1 and Employee2. One of the properties represents the manager for a given employee and the other represents a collection of the employee's direct reports. If we right-click on Employee1 and select Properties, we can examine the details for the Employee1 property.

(Note that we've also selected the Categorized icon on the toolbar in the Properties panel so that you can see how the properties are grouped.)

Notice that the Multiplicity for Employee1 is "* (Many)" and that the Return Type is "Collection of Employee". This lets us know that this property returns the collection of direct reports rather than the single Employee representing the current Employee's manager. Since we want our model to be both clear and rich in meaning, let's rename this property.

We'll change the value of Name to "DirectReports". When we do so, the value of Display Name will change as well.

Next, we select the property Employee2 on the Employee class. By examining its details, we confirm that it represents the employee's manager. We'll change its Name to "Manager".

Display Name is not a property native to Entity Framework. It's actually an attribute that DevForce adds to the EDM. DevForce extends both the Entity Framework EDM and the Entity Framework Visual Designer with its own XML attributes and properties. All of the DevForce extensions are grouped under the "DevForce Code Generation" category.

We now have a functional business model, and we can turn our attention to the user interface.

Run a query

Let's say that we want a form in our application for viewing the details of employees.

We'll begin by creating a new class that will be a logical representation of our form. This class will be our view model. We'll use the powerful data binding system in Silverlight to bind an instance of the view model to the currently existing MainPage.xaml. The XAML file and its associated code-behind are known as the *view*. This approach is very popular in Silverlight development and is part of the design pattern known as Model-View-ViewModel.

Right-click on the Silverlight project, SimpleSteps, and select Add | Class ….

Name the new class MainPageViewModel and then click Add. (We're choosing this name to remind us that it is an abstraction of the view MainPage.xaml.)

We'll need a collection of all the employees. So let's add

```
public ObservableCollection<Employee> Employees { get; private set; }
```

```
Public Property Employees() As ObservableCollection(Of Employee)
 Get
   Return m_Employees
 End Get
 Private Set(ByVal value As ObservableCollection(Of Employee))
   m_Employees = value
 End Set
End Property
```

and we'll need to add a using or Imports statement also:

```
using System.Collections.ObjectModel;
```

```
Imports System,Collections.ObjectModel
```

ObservableCollection<T> is a special type that raises an event whenever items are added or removed. This is known as *change notification* and is an important part of Silverlight's data binding system.

Employee is one of the entity classes that was generated by DevForce.

If you created the solution from the standard Silverlight Application template, you'll also now need to add a using/Imports statement for the namespace in which the model was generated. In this example, that namespace will be *SimpleSteps.Web*.

Now, we want to populate our new collection with data. We gain access to the data by means of a class known as the EntityManager. DevForce generates a special EntityManager for us that is extended to support our specific database. Let's jump back to the visual designer for NorthwindIB.edmx for a moment. If we click somewhere on the design surface, in empty space and not on an entity, then we'll see the details for the EDM as a whole in the Properties panel.
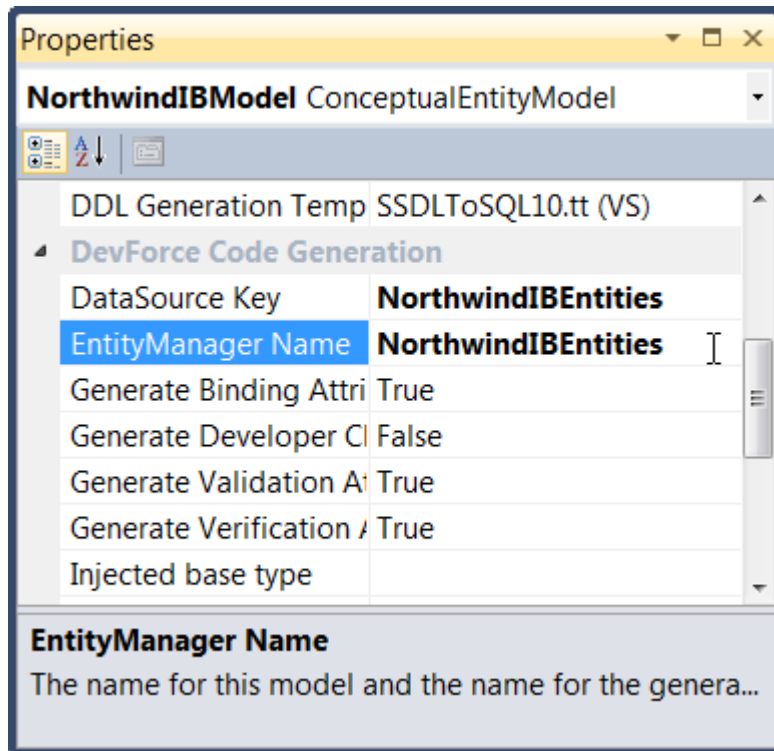
We're interested in the property EntityManager Name. This is the name of our model-specific class.

Let's add a constructor to MainPageViewModel and from that call the method we'll use to initialize the view model asynchronously. In that method we'll create an instance of our EntityManager. We'll also initialize the Employees collection.

```
async void AsyncMainPageViewModel() {
  Employees = new ObservableCollection<Employee>();
  var mgr = new NorthwindIBEntities();
}
```

```
Async Sub AsyncMainPageViewModel()
  Employees = New ObservableCollection(Of Employee)()
  Dim mgr As New NorthwindIBEntities()
End Sub
```

If you examine the properties of the NorthwindIBEntities manager, you will discover that many of them correspond to the entities in our EDM. In fact, we see a property named Employees that returns the type EntityQuery<Employee>. These properties are our "hooks" for querying the data in our database. For example, let's say that we want all of the employees whose last name begins with 'B'. We could use LINQ to create the query:

```
var query = from employee in mgr.Employees
  where employee.LastName.StartsWith("B")
  select employee;
```

```
Dim query = From employee In mgr.Employees _
  Where (employee.LastName.StartsWith("B")) _
  Select employee
```

In our case, we want the all of the employees in the database. We don't need to filter the set, so we can simply say:

```
var query = mgr.Employees;
```

```
Dim query = mgr.Employees
```

This abbreviated form is equivalent to:

```
var query = from employee in mgr.Employees
  select employee;
```

```
Dim query = From employee In mgr.Employees _
  Select employee
```

We want to emphasize that the Employees property produces a query and not a collection of objects. In order to get results of the query, we need to execute it. Just defining the query does not actually cause the database to be hit. Additionally, note that

our query lives on the Silverlight client, and that Silverlight prohibits remote synchronous operations such as querying a server for data. We'll have to fetch the Employees asynchronously.

First, we'll need two more using statements:

```
using IdeaBlade.Core;
using IdeaBlade.EntityModel;
```

```
Imports IdeaBlade.EntityModel
Imports IdeaBlade.Core
```

and then we'll execute our query like this:

```
var results = await query.ExecuteAsync();
results.ForEach(Employees.Add);
```

```
Dim results = Await query.ExecuteAsync()
results.ForEach(Sub(emp) Employees.Add(emp))
```

Here we *await* the *ExecuteAsync* method, and then add the result, an *IEnumerable<Employee>, to the* ObservableCollection. *See the topic on asynchronous queries for more information.*

We are using the extension method *ForEach()* to iterate over the results and add them to the ObservableCollection<Employee> (Employees) in MainPageViewModel. The ForEach() "extension method" is a part of *IdeaBlade.Core*. It executes a given method once for each member in Results, and so we use it to call the Add() method on Employees.

The complete code for our view model now looks like this:

```
using IdeaBlade.Core;
using IdeaBlade.EntityModel;
using System.Collections.ObjectModel;
namespace SimpleSteps {
 public class MainPageViewModel {
  public MainPageViewModel() {
    AsyncMainPageViewModel();
  }
  async void AsyncMainPageViewModel() {
    Employees = new ObservableCollection<Employee>();
    var mgr = new NorthwindIBEntities();
    var query = mgr.Employees;
    var results = await query.ExecuteAsync();
    results.ForEach(Employees.Add);
  }
  public ObservableCollection<Employee> Employees { get; private set; }
 }
}
```

```
Imports System.Collections.ObjectModel
Imports IdeaBlade.EntityModel
Imports IdeaBlade.Core
Namespace SimpleSteps
  Public Class MainPageViewModel
    Public Sub New()
        AsyncMainPageViewModel()
    End Sub
    Async Sub AsyncMainPageViewModel()
      Employees = New ObservableCollection(Of Employee)()
      Dim mgr = New NorthwindIBEntities()
      Dim query = mgr.Employees
      Dim results = Await query.ExecuteAsync()
      results.ForEach(Sub(emp) Employees.Add(emp))
    End Sub
    Public Property Employees() As ObservableCollection(Of Employee)
      Get
        Return m_Employees
      End Get
      Private Set(ByVal value As ObservableCollection(Of Employee))
        m_Employees = value
      End Set
    End Property
    Private m_Employees As ObservableCollection(Of Employee)
  End Class
```

End Namespace

Display the data

For this next step, we're going to need the Silverlight Toolkit from Microsoft. It provides a number of additional controls that are not a part of the basic Silverlight SDK. If you don't have already have the Toolkit installed, you can download the latest release from:

http://silverlight.codeplex.com/

Be sure to download the **Silverlight 5** release.

Please note that the Silverlight Toolkit is not the same thing as Silverlight Tools. The Toolkit is mostly a set of controls whereas Silverlight Tools contains the primary SDK and Visual Studio integration bits.

Let's open MainPage.xaml.

If you look in the Document Outline panel (or even in the XAML pane), you will see that the root element is a UserControl.
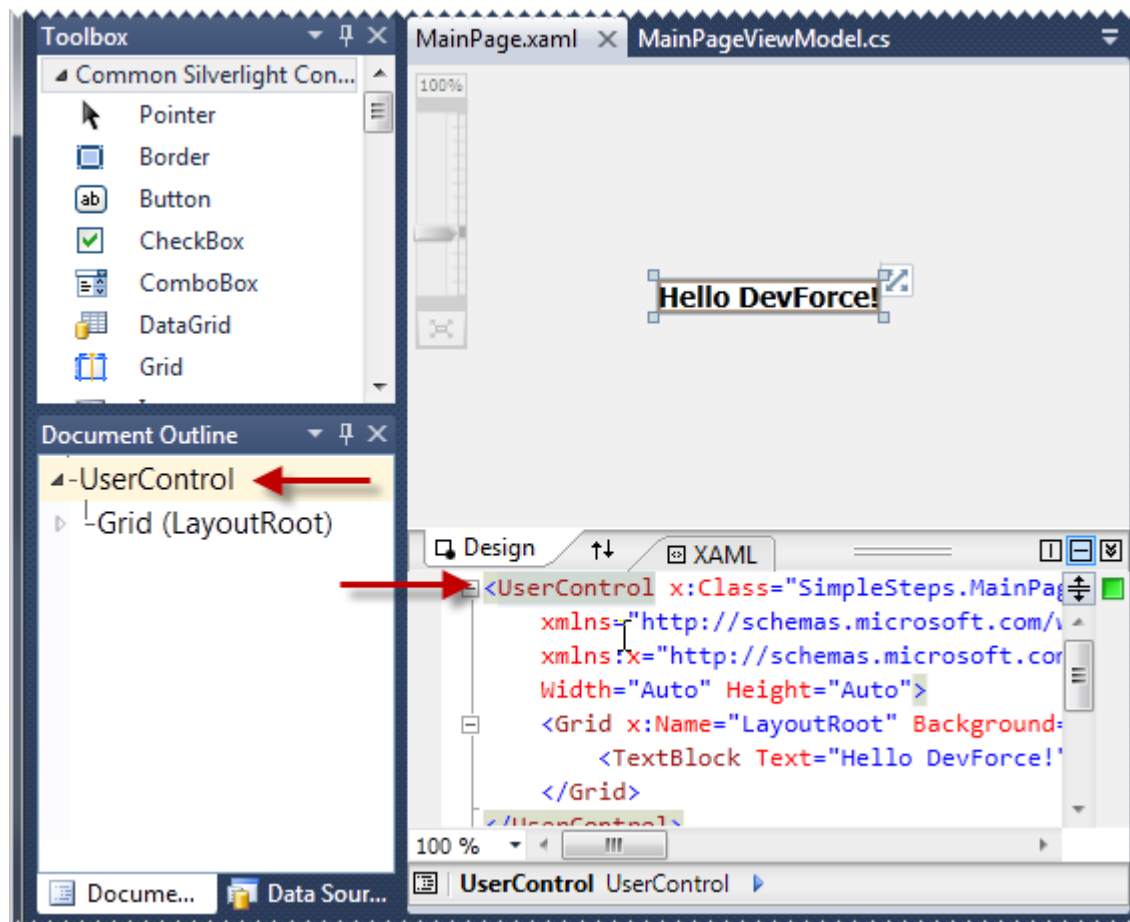
Select the UserControl either by clicking on it in the Document Outline or by clicking on the blank space in the design surface.

Next, click on the small square on the lower right corner of the bounding box. Drag this down and to the right to increase the design-time size of the user control. Resize it so that it fills the majority of the design surface. This won't affect the size of the form when it's running: it only changes the size inside the designer.
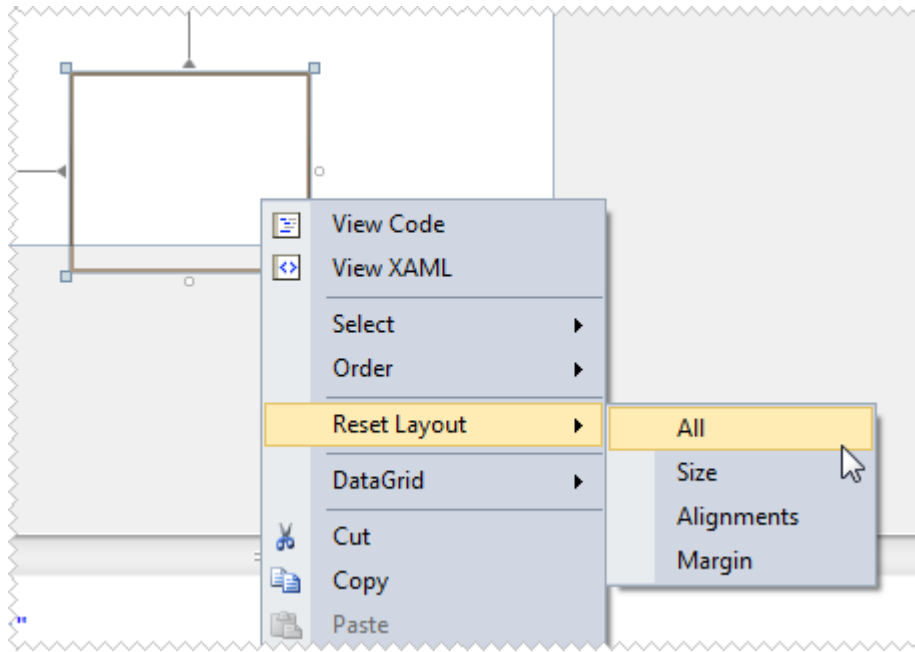
Next, locate the DataGrid control in the Toolbox and drag it anywhere on to the design surface. Dragging the control to the design surface does several things for us. First, it adds a reference in the project to the necessary assemblies from the Silverlight Toolkit. Second, it adds a namespace alias to the XAML file so that we can access the control in the markup.

The initial placement, alignment, and margins of the data grid are not very useful. Right-click on the data grid and select Reset Layout | All.

If you examine the XAML, you'll see that the data grid is represented by the markup:

```
<sdk:DataGrid AutoGenerateColumns="False" Name="dataGrid1" />
```

We want to use this control to view our employees. We have to make two changes to the XAML for the DataGrid:

1. Change the value of its AutoGenerateColumns property to "True"; and
2. Bind its ItemsSource to the Employees collection on our view model.

```
<sdk:DataGrid AutoGenerateColumns="True"
   Name="dataGrid1"
   ItemsSource="{Binding Employees}"/>
```

Ok, there's one more thing we need to do before we can run the application. We need to tell the view, MainPage.xaml, about our view model, MainPageViewModel.

Right-click on MainPage.xaml and select View Code. We're going to create an instance of our view model and set it as the data context for MainPage.xaml whenever it loads. The data context is what Silverlight uses to resolve data bindings. We've told Silverlight to bind the data grid to Employees, but Silverlight doesn't yet know what "Employees" means. We have to give it a context so that it can correctly interpret the binding.

Modify the constructor for MainPage.xaml.cs to this:

```
public MainPage() {
  InitializeComponent();
  Loaded += delegate { DataContext = new MainPageViewModel(); };
}
```

```
Public Sub New()
  InitializeComponent()
  DataContext = New MainPageViewModel()
End Sub
```

Now we are ready to run the application!

Notice that it takes just a second before the data is displayed in the data grid. This is because the Silverlight client is making an asynchronous call to the application server to retrieve the data.

This concludes Part 1 of the tour of DevForce Silverlight.

Learn More

We've briefly covered a lot of topics here. Here are some links to what we've covered, and additional information we hope you find useful.

- Getting started with DevForce
- Creating a new DevForce application
- Understanding the Entity Data Model
- Code generation
- EntityManager Overview
- Writing a LINQ query
- More on related technologies

Prerequisites

The user interface for the application built during this tour uses a DataForm component supplied by the Silverlight 5 Toolkit (different from the Silverlight 5 Tools!). You can download the Toolkit here:

http://silverlight.codeplex.com/

To work with this application, as all DevForce 2012 Silverlight applications, you will need Visual Studio 2012 and Silverlight 5.

1. ^ Navigation Properties are properties that allow you to "navigate" to other entities in the model. For example, if you have an instance of an Employee you can navigate to any Order instances that are associated with that employee.