Microsoft Visual Studio LightSwitch Extensions Cookbook – Beta

Table of Contents

Introduction

With the release of Microsoft Visual Studio LightSwitch, customers can easily build applications for the desktop or for the cloud.

While the built-in functionality provides a lot of capabilities for the developer, there are times where the developer may not have the time or the expertise to invest in additional functionality.

In this case, LightSwitch includes extensible capability as a part of its design. An extensibility developer has a number of extension options that they can choose from.

This cookbook is designed to educate you on what extensibility is available for the LightSwitch product, and to illustrate the various extensions types that can provide entry points into extending a LightSwitch application. The final section contains recipes for creating the various extension types, allowing you to get started on building the extensions and be aware of the different options for building an extension.

What are extensions?

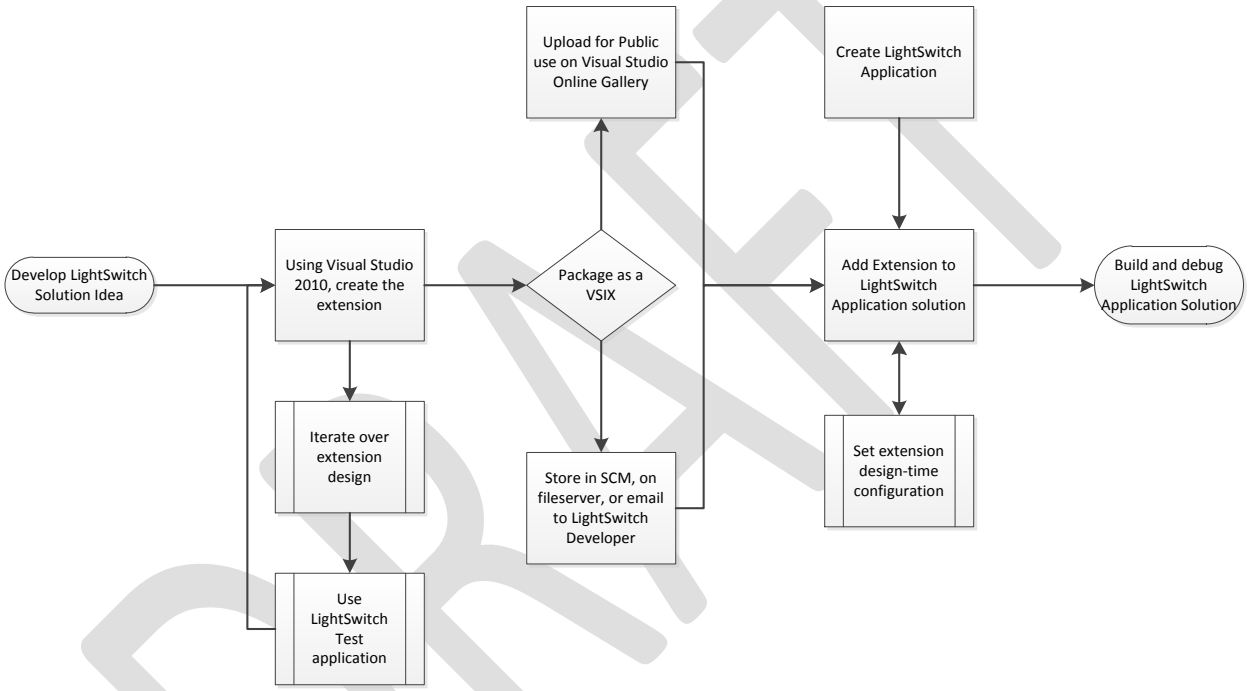
Extensions provide additional functionality that is not standard in the product. A way to think of them is like add-ons. The extensions can be used to directly interact with the user, or to do some work behind the user interface such as data access. In LightSwitch there are 6 extension points illustrated in the following diagram:

An important way of thinking about extensions is that you can combine multiple extensions to create a solution-based experience for the LightSwitch developer. An example of this would be where an extension provides a Money Market solution by using a Shell that has specific trading navigation, a theme that is specific to the trading company, and a number of screen templates and controls that provide visualizations for trading data. The data could also come from a variety of data sources and a custom data source could be the extension that aggregates the data to the application.

LightSwitch uses the Microsoft Extensibility Framework (MEF) which allows a developer to build a LightSwitch application using a model centric architecture approach. Extensions are a combination of .NET/Silverlight assemblies and metadata that are packaged together and consumed by a Visual Studio LightSwitch application.

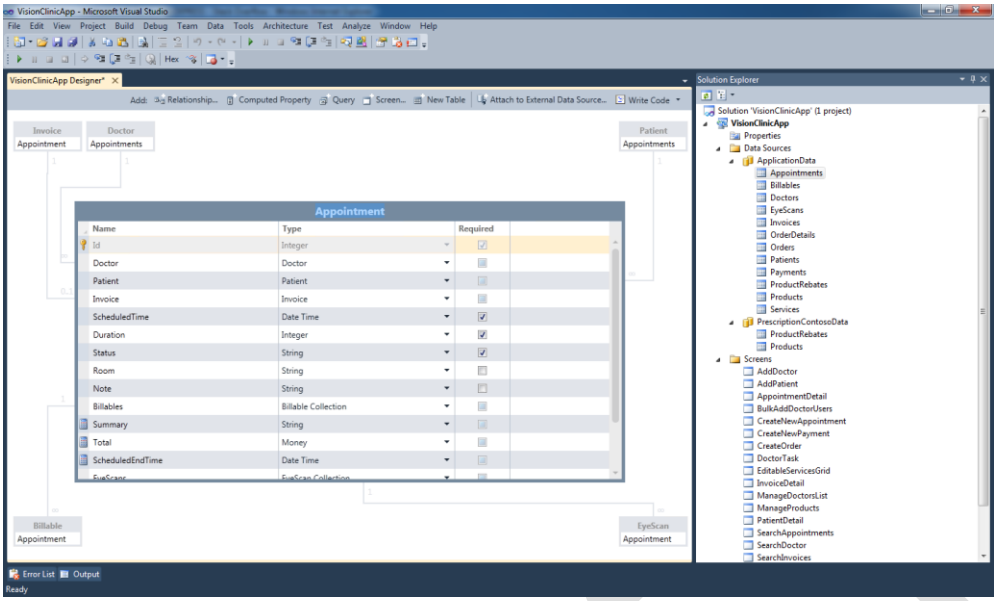
Extensions can provide a design or runtime experience, or both, depending on the extension type.

The following diagram shows the high-level steps in creating an extension and then having it consumed by a LightSwitch Application.

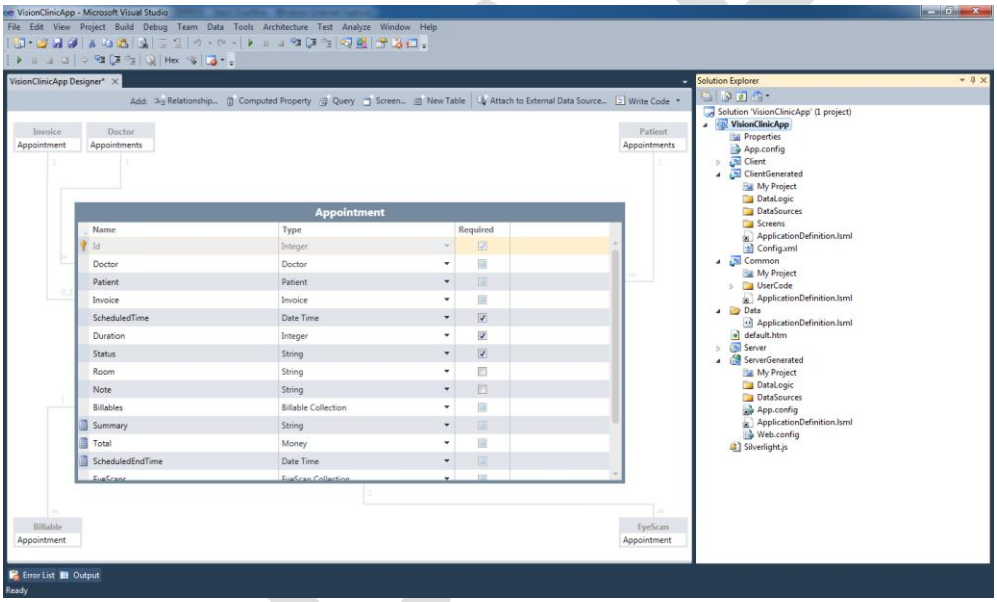


The diagram above shows that there is a possible iterative approach in designing, building and testing the extension. Even though a LightSwitch Test application is used, it is quite possible that the application is a smaller version of the complete solution that is being achieved.

The extension structure/layout is in support of the physical project layout of a Visual Studio LightSwitch application. When a developer is creating a LightSwitch application, they only see the one logical application project structure in the LightSwitch IDE as shown below:



If you select file view and show hidden projects you will see a very different project structure.

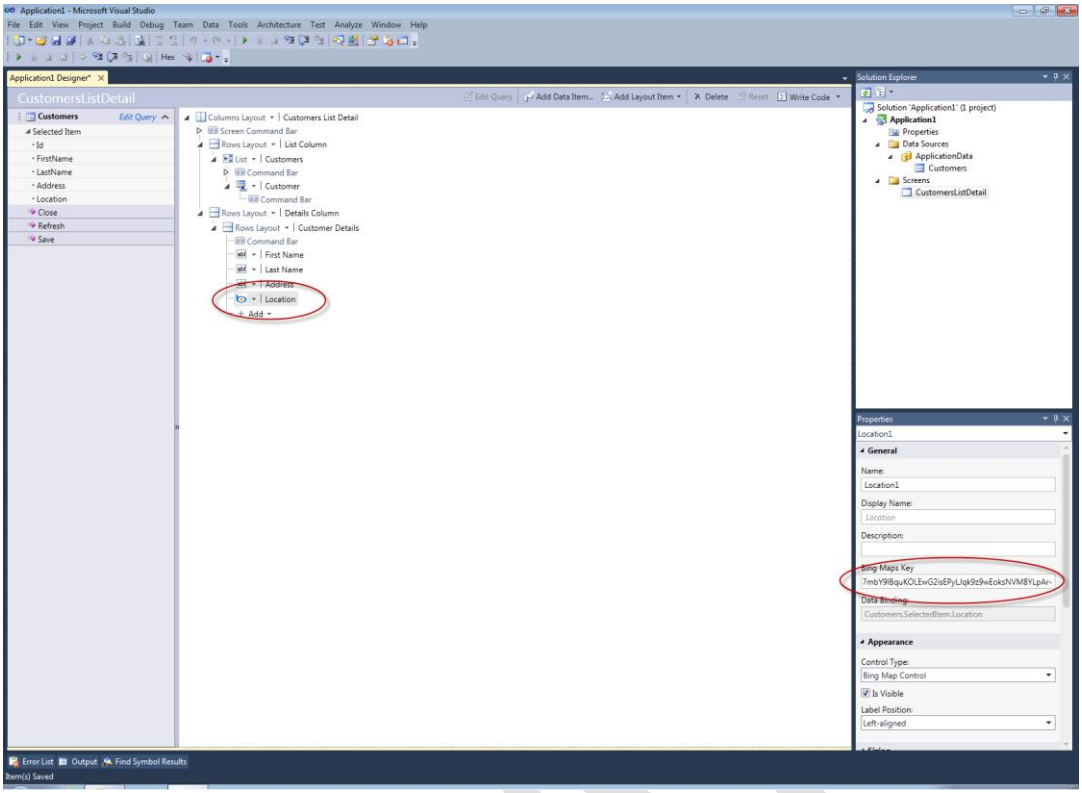


The project structure above shows how an extension should be structured and developed.

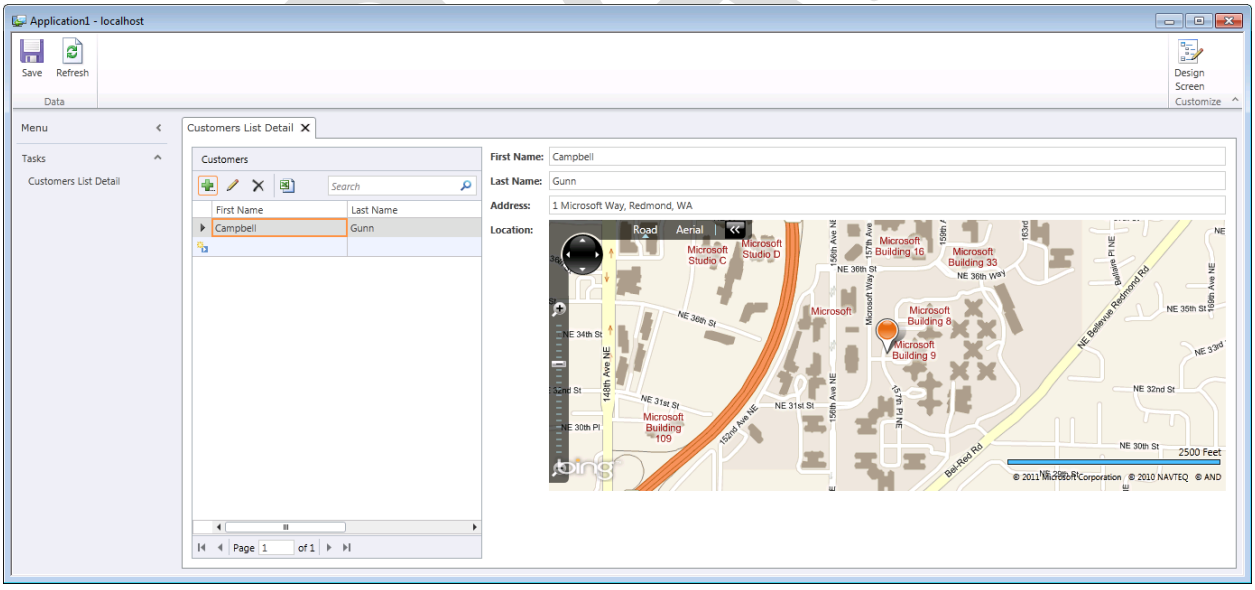
Extensions are added to an existing LightSwitch application via the design time environment

During design time development, a control extension is added in the same way a built-in control is added. This ensures that the experience in LightSwitch will be the same no matter what type of extension is being used.

Custom branding can be shown beside the control and also in control properties that require additional information, in the case below a license key to allow the Bing Map to work with the Bing Map web service.

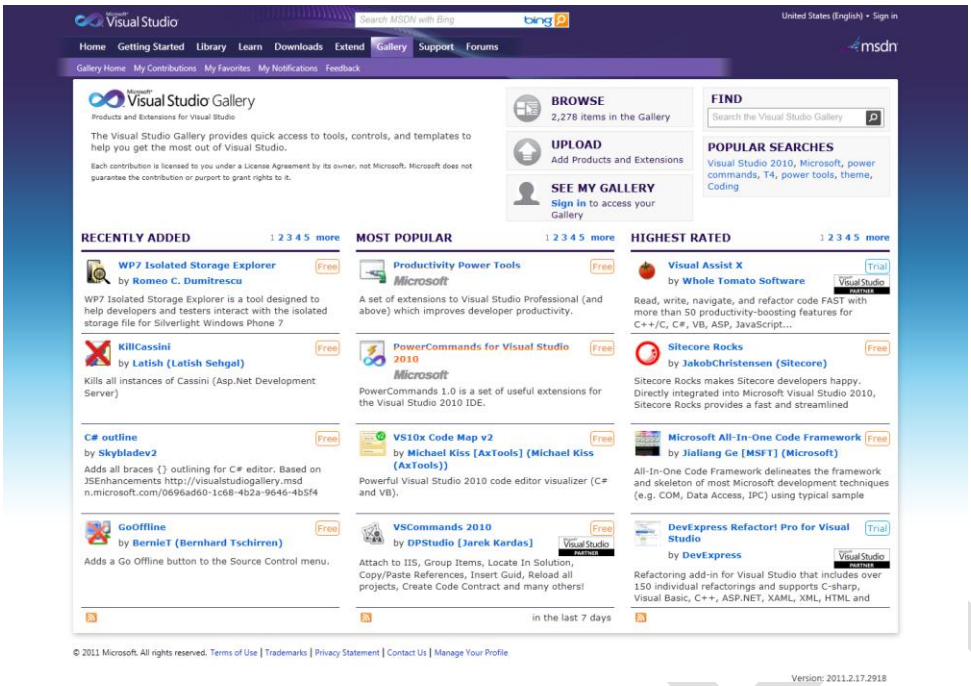


In the runtime the experience can be shown in the illustration below:

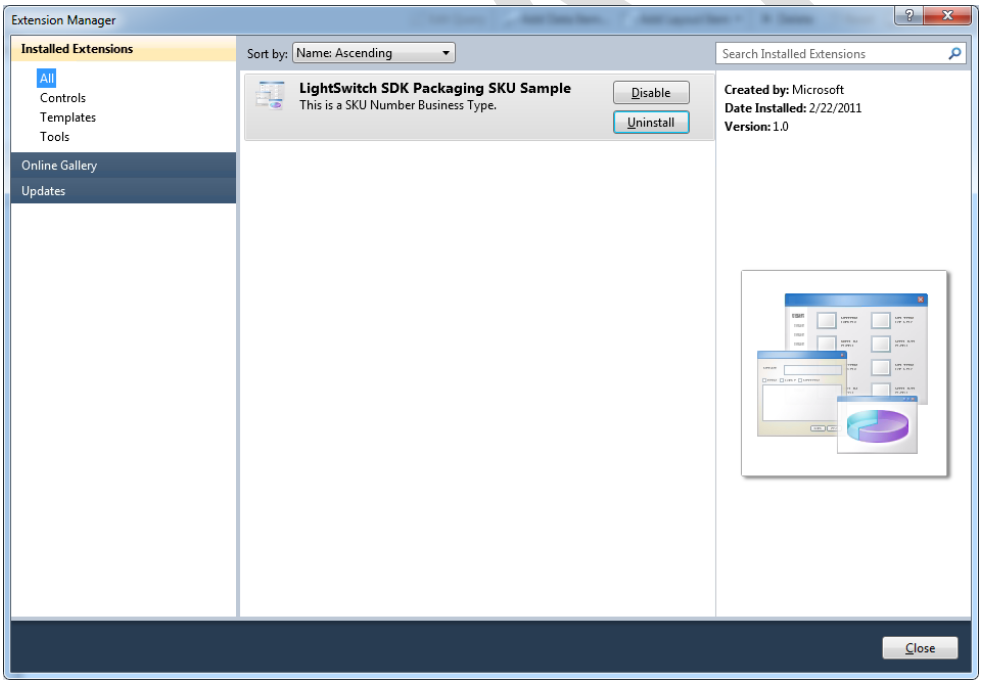


Extensions are found on the Visual Studio Gallery, where they can be downloaded via a web browser or from within the LightSwitch environment via the Extensions Manager.

Visual Studio Gallery (http://visualstudiogallery.msdn.microsoft.com/):



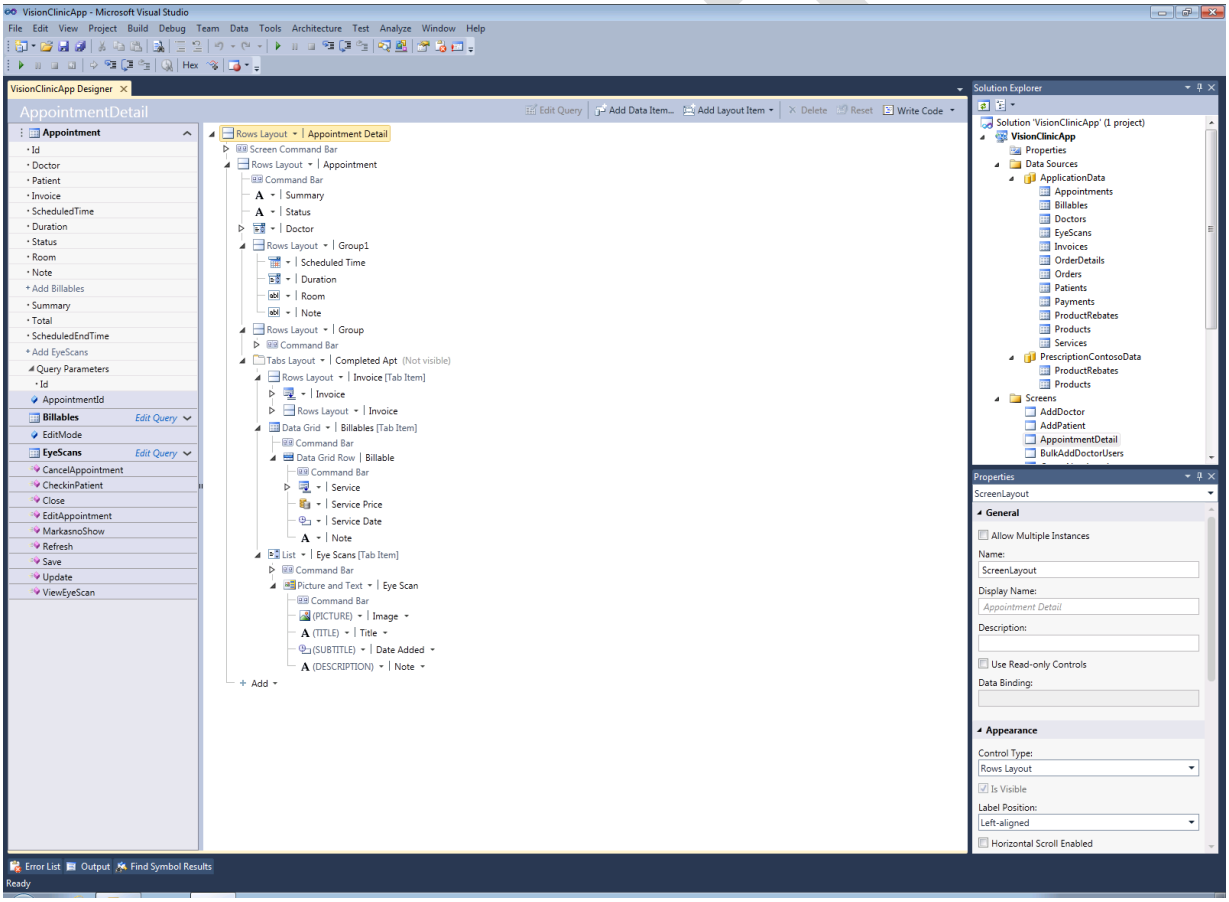
Extension Manager in LightSwitch after an extension has been installed:



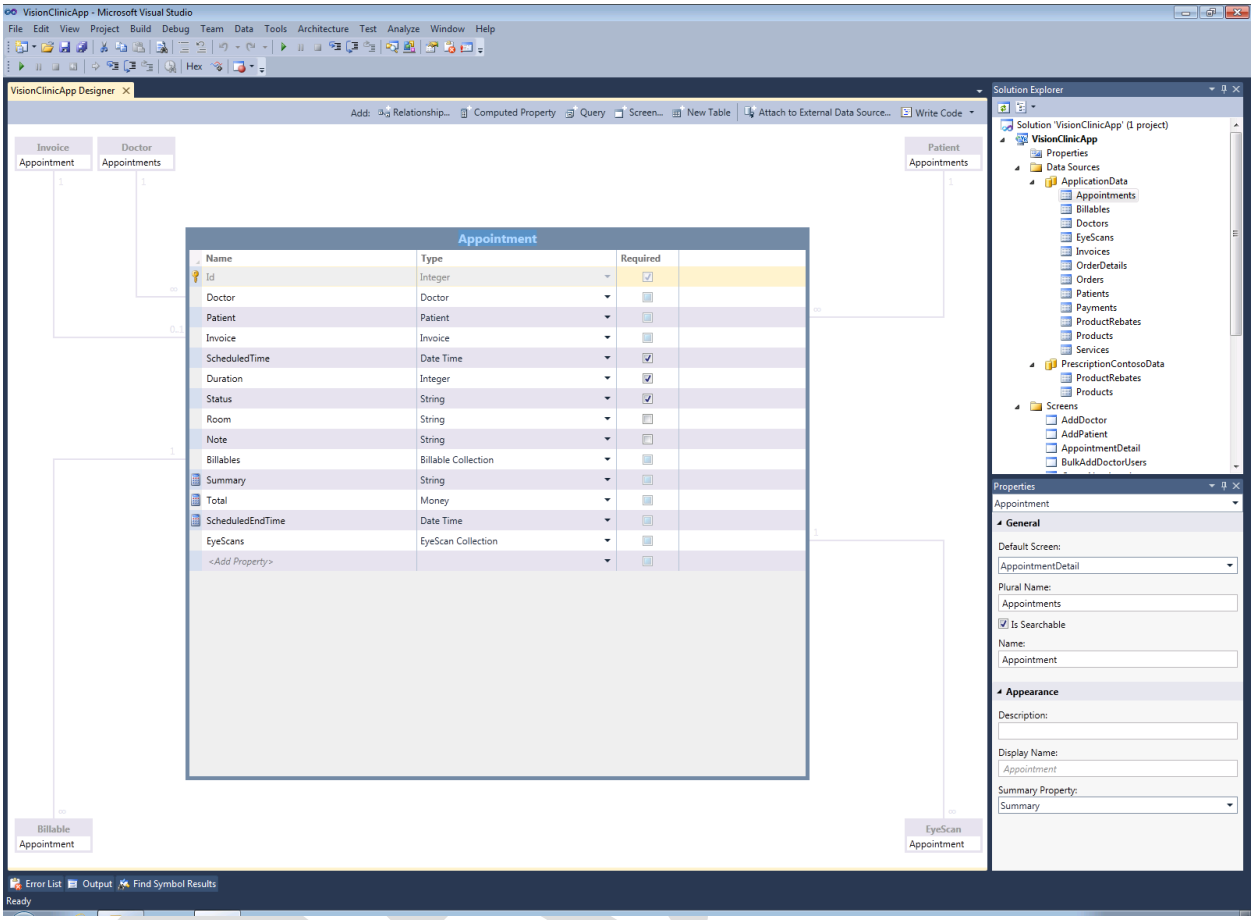
LightSwitch Extension Architecture

When designing a LightSwitch extension, you need to consider two separate things: how you interact with the extension during design time, and how the extension is implemented in the runtime. In fact, you can create a design time experience without having any runtime implementation first. You will see this later in this document, when you start creating extensions using recipes.

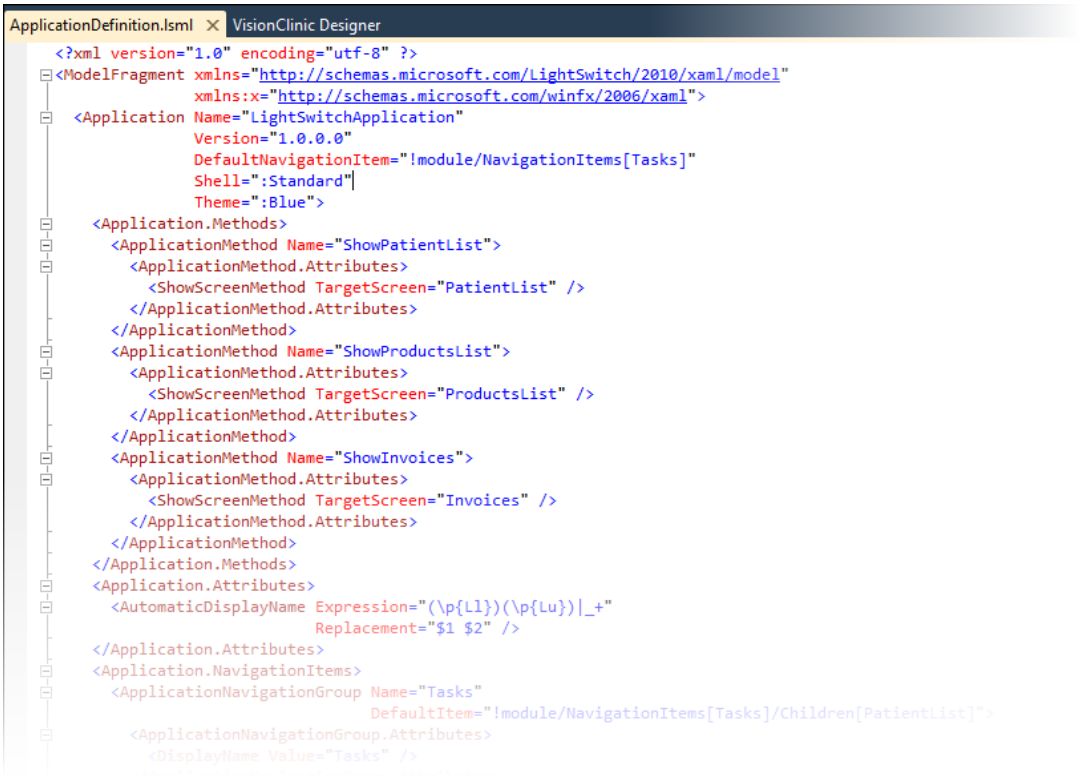
Most of the design time elements are contained within a LightSwitch Markup Language file or also known as lsml. The lsml file is like an xml file and is also a model fragment that gets added to your LightSwitch Application model, so that LightSwitch knows how it will show up in the IDE. An example of this is a control in the screen designer within the visual tree, shown below:



Or in the entity designer, shown below:



If you change to file view we can see the ApplicationDefinition.lsml, which is the model file for the LightSwitch Application. In this case it is the model for the sample VisionClinic application.



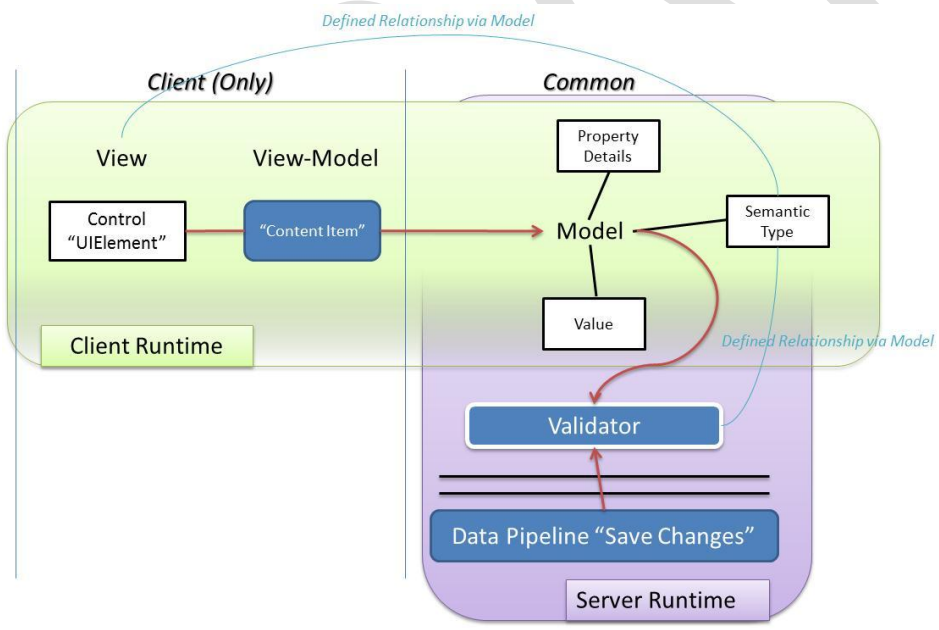
Within an extension there is a model fragment that provides metadata to the LightSwitch application and is integrated to the Application Definition above.

The following is an example of a presentation.lsml file for a control extension.

```
<?xml version="1.0" encoding="utf-8" ?>
<ModelFragment
  xmlns="http://schemas.microsoft.com/LightSwitch/2010/xaml/model"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <Control
    Name="BingMapControl"
    SupportedContentItemKind="Value"
    AttachedLabelSupport="DisplayedByContainer"
    DesignerImageResource="BingImages::BingMapControl" >
    <Control.Attributes>
      <DisplayName Value="$(BingMapControl_DisplayName)" />
    </Control.Attributes>
    <Control.SupportedDataTypes>
      <SupportedDataType DataType=":String"/>
    </Control.SupportedDataTypes>

    <!-- Property overrides -->
    <Control.PropertyOverrides>

    </Control.PropertyOverrides>

    <!-- BingMapControl Properties -->
```

```
    <Control.Properties>

      <!-- DefaultBingMapControl Property -->
      <ControlProperty Name="ApplicationKey"
        PropertyType=":String"
        IsReadOnly="False"
        CategoryName="Bing Map Properties"
        EditorVisibility="PropertySheet">
        <ControlProperty.Attributes>
          <DisplayName Value="$(BingMapControl_ApplicationKey_DisplayName)"/>
          <Description Value="$(BingMapControl_ApplicationKey_Description)"/>
        </ControlProperty.Attributes>
      </ControlProperty>
      <!-- End DefaultBingMapControl Property-->

    </Control.Properties>
    <!-- End BingMapControl Properties-->

  </Control>
</ModelFragment>
```

In order to piece these concepts together, you first need to understand a little about the LightSwitch Architecture. LightSwitch architecture is a model-driven based approach and uses the View – View Model – Model architecture.



The diagram above illustrates the concept of a business type (a specific type of extension). You can see the view, view-model, model, but you should also note here that you cannot access the model directly from the view (UI). This is an important concept to understand. How you can access the model is through the view-model using a Content Item, which builds a matrix of the view and model elements.

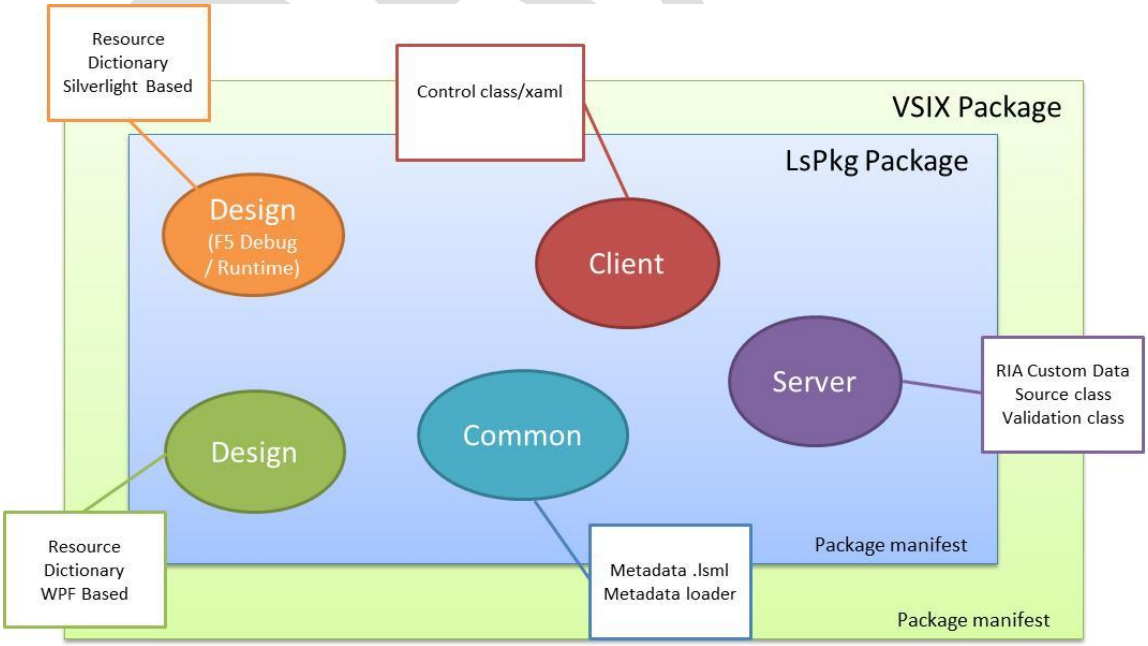
Where do various code elements live that you create? Any controls that are presented in the runtime (a UIElement) live in the View layer and also reside in the client project of the LightSwitch Application when an extension is added to a LightSwitch application. The Content Item is part of LightSwitch and is also the View-Model layer.

The next layer is the model, which in the diagram above is an intersection of common and server projects. Looking at the common portion of the diagram, this is where the lsml file is created and defined. What is unique about this location and file is that it provides model definitions for the client and server projects, in design time and runtime. Think of it as a way to share common code and model information for your extension in the LightSwitch Application.

What is interesting here is that we talk about client, common, server projects that make up a LightSwitch Extension, this is also the same for a LightSwitch application, except the extension assemblies go into ClientGenerated, Common, and ServerGenerated. In order for an extension to place the right assemblies in the correct locations, in LightSwitch extension development, we have the notion of designations. Think of designations as pointers when an extension is being unpacked that bits of the extension are placed in the right location for a LightSwitch Application.

There are additional designations that work across the different projects, which depending on the extension allow them to appear only in certain places of the design time and runtime experience. You will learn more about designations later in this document.

Let's now move on to how an extension is packaged. Below is a high-level diagram that shows what extension projects are needed for extension functionality. These projects are then wrapped in a zip like file called an LsPkg package. In fact, you can rename the file extension to zip and then you will be able to open up the file and look inside.



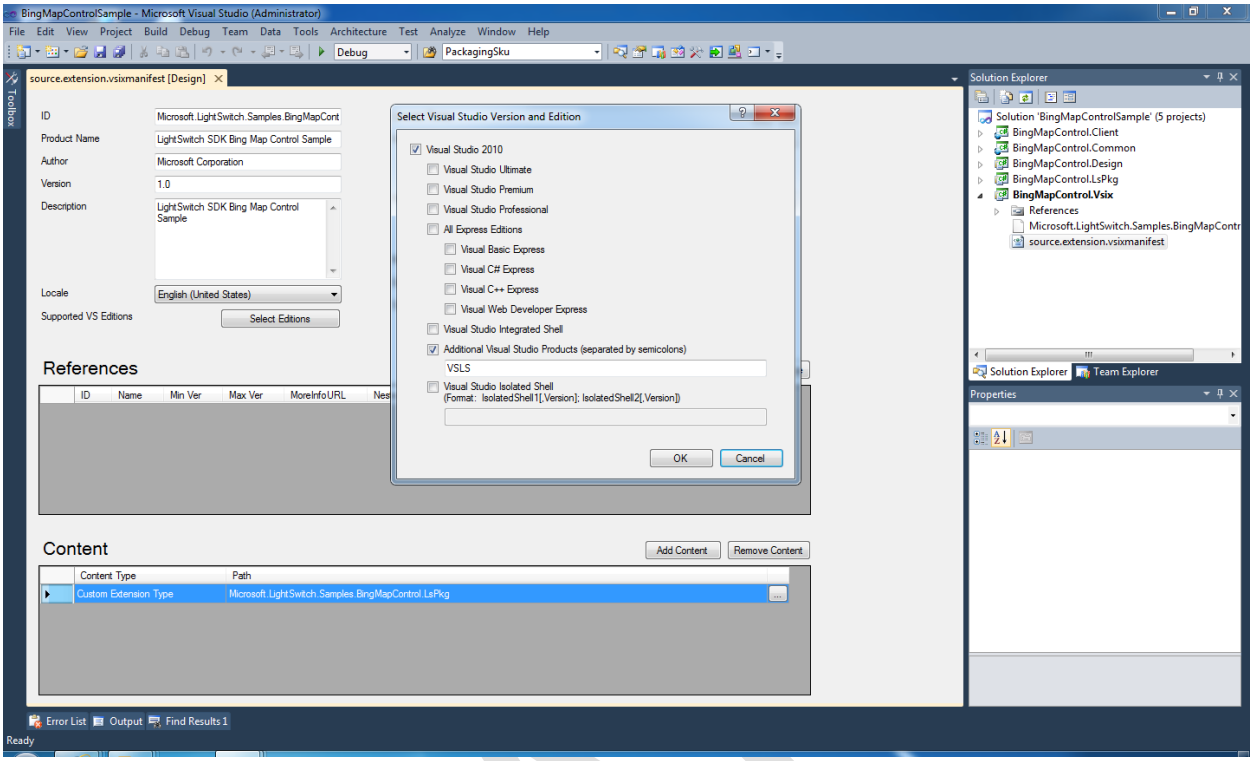
The LsPkg file contains the compiled assemblies of the projects within the extension solution as well as some metadata, and manifests that tell where the assemblies need to go when unpacked (designations).

The projects are specific project types based on the functionality they are to provide within the extension. The following table is a breakdown of the projects, types, and typically what resides in them.

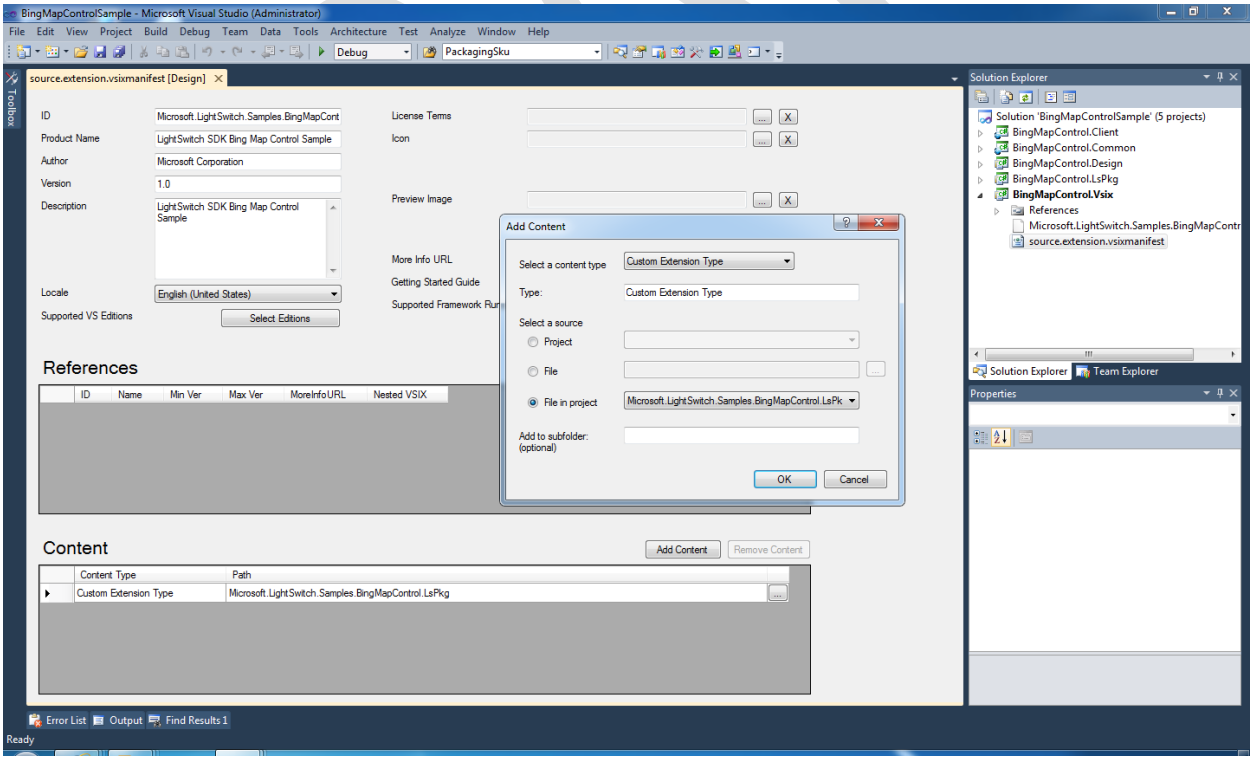
Project	Type	Contents
Client	Silverlight 4 class library project	This contains the UI elements of an extension, such as a control.
Design (known as ClientDesign)	Sliverlight 4 class library project	This contains image and resource files used when customizing the screen in debugging mode in the runtime.
Design	.NET class library project	Contains image and resource files that are used in the IDE during design-time.
Common	Silverlight 4 class library project (marked as portable) given that data is used from client and server.	Contains the metadata markup from the lsml files.
Server	.Net 4 class library project	Contains server side code, such as validation code used in business types and for custom data sources.
LsPkg	LightSwitch Package type project	Project type specific to packaging up assemblies to be unpacked into a LightSwitch application based on the manifest with defined designations.
Vsix	Visual Studio SDK package project	Uses the standard Vsix package project, with two custom properties defined, one is the SKU and the other is the LightSwitch Package.

The LsPkg package is then wrapped up in a VSIX package file. This is a standard VSIX Package, except for the following:

The first illustration shows the specific SKU type that needs to be provided. This ensures that the extension is only used for LightSwitch.



The second illustration shows the LsPkg being included in the VSIX package as a custom extension type.



When using the LightSwitch Extensibility project templates, these details are taken care for you, but it is important to know what is happening behind the scenes.

The designations are set within the LsPkg project properties.

The important aspect of the designations in the LsPkg project properties is that some project types need the appropriate designations set. The following table shows a matrix of what projects are needed and the necessary designations for a given extension type.

Extension Type / Designation	Shell	Theme	Control	Custom Data Source	Business Type	Screen Template
ClientReference			X[1]			
ClientGeneratedReference	X	X	X		X	
ClientReferenceResources			X[1]		X	
ClientGeneratedReferenceResources	X	X	X		X	
CommonReference						
CommonReferenceResources						
ServerReference						
ServerReferenceResources					X	
ServerGeneratedReference	X	X	X	X	X	
ServerGeneratedReferenceResources	X	X		X	X	
IDEReference	X	X	X	X	X	X
IDEReferenceResource	X	X	X	X	X	X
ClientDebugOnlyReference			X			
Project Types						
Client	X[2]	X	X		X	
Design.Client			X		X	
Common	X	X	X		X	X
Design			X		X	X
Server				X		
LsPkg	X	X	X	X	X	X
VSIX	X	X	X	X	X	X
Footnotes:						
[1] Having a Client Reference provides the assembly with public types that Tim can use.						
[2] Ensure that Shell Client Project references SdkProxy Assembly.						

To look back on what has been covered, you have learned about the architecture of a LightSwitch application and how the extension is overlaid/merged into an extension which is made up of multiple projects and gets designated to the appropriate locations when added and unpacked into a LightSwitch application. Finally, you learned about the packaging format for an extension.

Adding an Extension to a LightSwitch Application

So the big question remains - after you have built an extension, how does it get into a LightSwitch application? That depends on how you want to deliver your extension. The extension final file is a Vsix file, which can be put on a file server, e-mailed, and uploaded to the Visual Studio Online Gallery.

The first two methods require simply clicking on the Vsix file. For the online gallery, you can download the Vsix file from your browser, or if you want a more integrated feeling, then this can be done via Visual Studio Extension Manager Interface.

A Vsix package gets onto a developer's computer via a number of ways. One way is via the Visual Studio Gallery as already shown. The developer can go to the web site and download the package manually or they can obtain it directly through Visual Studio LightSwitch itself using the extension manager. If the extension is being developed in-house then the vsix package can be stored in a Source Code Management tool, on a file share, and even e-mailed to other developers.

Once the extension vsix package has been installed on the developer's computer, it is available for use in a LightSwitch project.

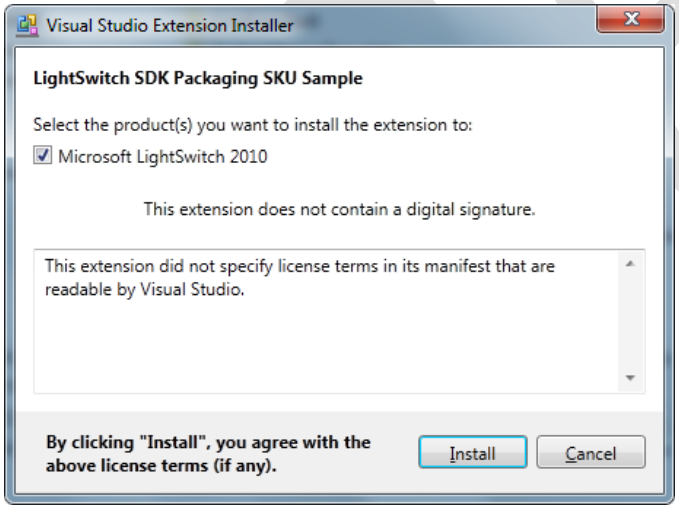


Figure 1. Installing a vsix package on a computer using the vsix standalone installer.

To use the extension you need to have an existing LightSwitch project open, or you can create a new one. At that point, you can select the Properties page for the application and open the Extensions tab to see the list of available extensions.

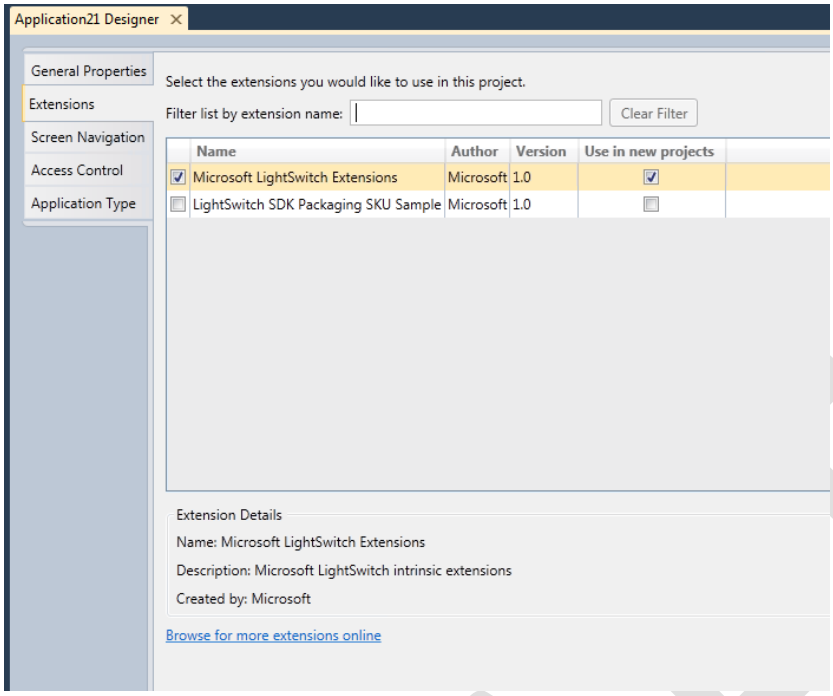


Figure 2. Extensions page in the LightSwitch application properties.

It is at this point that the LightSwitch developer can select one or many extensions to be used in the current project, or also choose to make an extension available for any future projects.

The extension that is shown here in this example is a business type, called the PackagingSKU. To use it in LightSwitch, first you will create an application and enable the extension. From there you will create an entity called “Products” that has two properties, ProductName and SKU, shown in figure 3. Since you have installed the Business Type extension, you can select the Packaging SKU type as shown in figure 2.

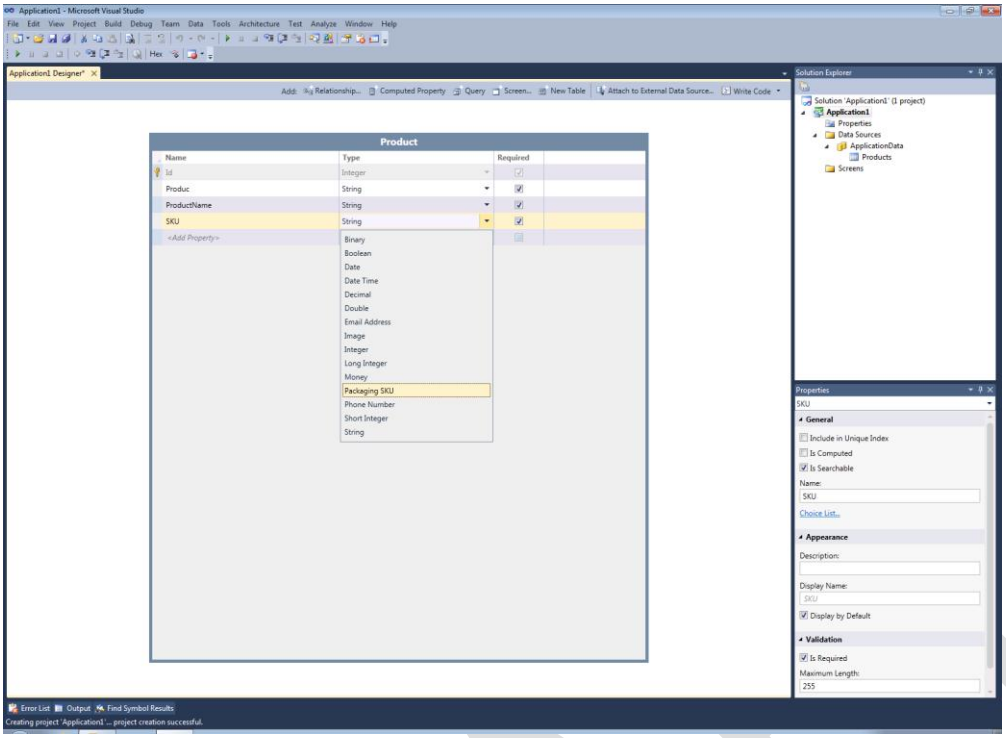


Figure 3. Entity Designer on the added business type

The next step is to create the screen and select the controls that will work with the Products entity as shown in the figure 4 below.

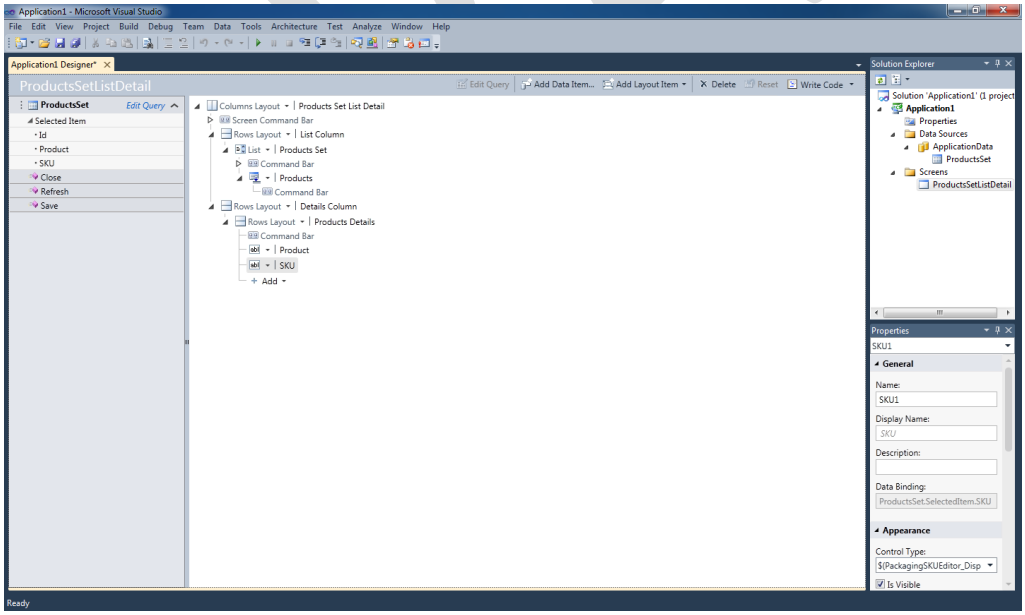


Figure 4. Screen Designer and selecting the editor control

Here the SKU control is selected and the editor version on the control is used, allowing data to be entered. Next you can run this application and look at the behavior of the PackagingSKU business type.

If the SKU is entered correctly on the screen the validation is not triggered. However, if the data is entered incorrectly, meaning the format of the SKU isn't valid, and then the validation message appears as shown in the figure 5 below.

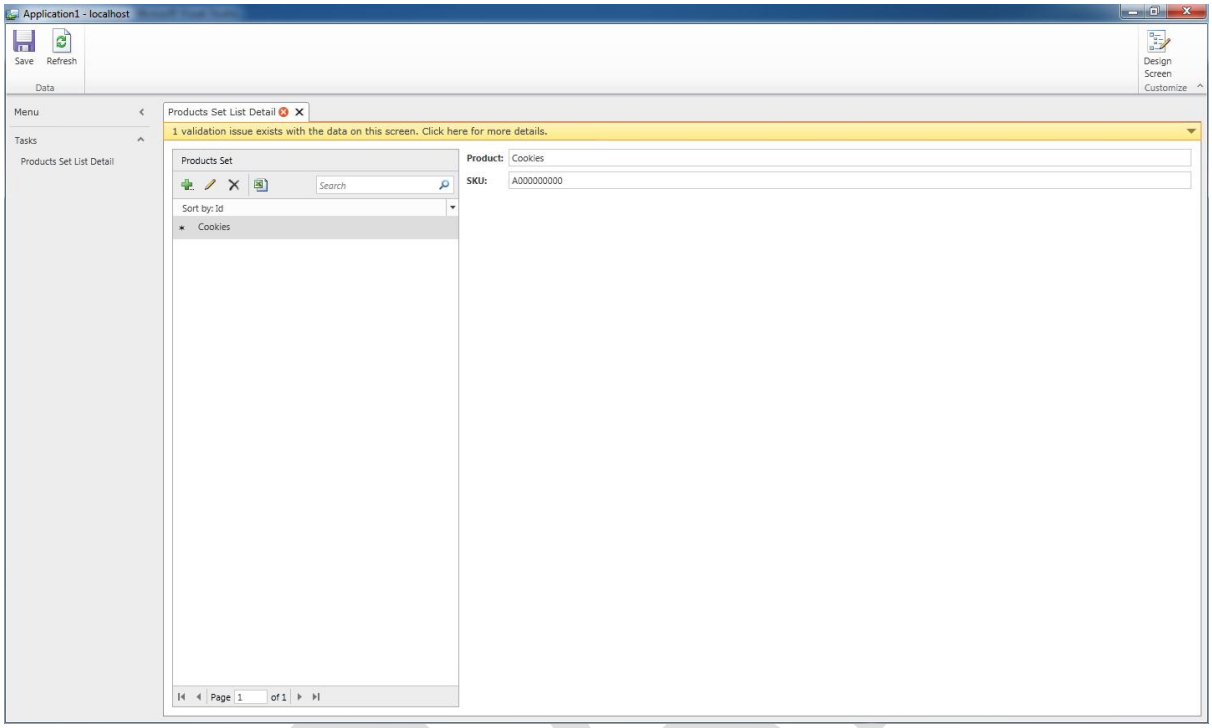


Figure 5.Using a business type extension on a screen will trigger validation when data is entered

LightSwitch Extension Types

Extension types are specific points of extensibility within the LightSwitch application environment. There are six different types of extensions. The following is a list of the extension types and descriptions of their capabilities.

Controls

This extension type is a building block to extension development. The reason is that the user interacts with a control. Controls appear in the screen designer, they have a design-time representation in the visual tree, which can include a custom image icon to provide a branding experience for the control. The control can also have properties to provide user-defined information to make the control behave in a particular way. The way to let screen designer interpret the control and type of behavior the control will implement in design-time is via metadata in the lsml file. Every extension requires some metadata contained in a model fragment in the lsml file. A control extension makes extensive use of it. Controls need the following basic information in metadata. If you use the extensibility project templates then this is automatically provided for you.

Here is the default information that should be provided:

```
<Control
    Name="BingMapControl"
    SupportedContentItemKind="Value"
    DesignerImageResource="BingImages::BingMapControl" >
    <Control.Attributes>
      <DisplayName Value="$(BingMapControl_DisplayName)" />
    </Control.Attributes>
    <Control.SupportedDataTypes>
      <SupportedDataType DataType=":String"/>
    </Control.SupportedDataTypes>
<control/>
```

There are obvious value pairs here, such as the Name, Display Name. Let's focus on the values that do have some significant impact to the way you design your control.

Controls can be different kinds, which is referenced in the SupportedContentItemKind. The following is a list of controls supported:

Value	Description
Unset	Not set.
Value	Represents a style for a specific scalar simple data type. This is visualized using a single control.
Details	Represents a content item for an entity or complex type. This may be visualized using a single control or as individual fields using the node's children (if there are no children in metadata, they will be automatically created by the system if needed).
Command	Represents a command content item. This may use a style like a button, link, etc.

Collection	Represents a collection content item. (Using styles like: Grid, ListBox, etc.)
Group	Represents a group content item. (Using styles like: TableLayout)
Screen	Represents a screen
ScreenContent	Represent a node which always binds to the screen. Useful for custom controls.

Depending on the kind value that is selected above, this will have an effect on how you implement your code to represent the control in the runtime. An example of this is, when accessing data from an entity, the binding for your control can differ based on the value kind. A Value can access a single scalar value by {Bindings=Details.Value}, while accessing a Collection requires that collection is created using IEnumerable and is populated by getting a collection of values of an entity via IContentItem. This is because direct access to the model via a control is not permitted and must go via the viewmodel through the IContentItem. You will learn more about how to use SupportedContentItemKind in the control recipe explained later in this cookbook.

The Supported Data Type allows the developer to tightly control what type of data can be used with this control. The following table shows the different options:

```
<SupportedDataType DataType=":String"/>
<SupportedDataType DataType=":Decimal"/>
<SupportedDataType DataType=":Double"/>
<SupportedDataType DataType=":Guid"/>
<SupportedDataType DataType=":Int16"/>
<SupportedDataType DataType=":Int32"/>
<SupportedDataType DataType=":Int64"/>
<SupportedDataType DataType=":DateTime"/>
```

After the model is represented it now requires the necessary implementation code. This is the same approach as creating a Silverlight control. The only difference is if you want to provide additional design-time functionality that can affect the runtime behavior of the control. A prime example of this is properties. There needs to be some metadata defined about the property for a given control as well as some implementation code such as getters and setters, dependency properties and callback functions., Again, more of this is explained in the recipe section of this document.

Another important aspect to think about is that if you have multiple controls that have dependencies on each other. How would you show this experience in the screen designer?

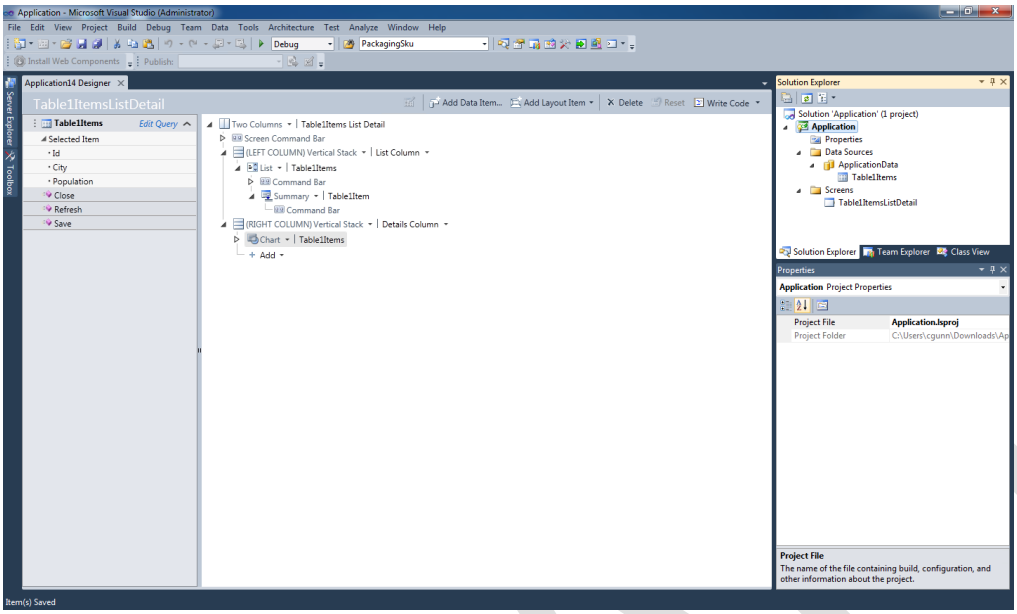
An example of this is a chart control. How would this be represented in the screen designer?

One way could be that you have a single control and have a separate properties dialog that the user can access through the properties window to set the necessary values for a given axis, the type of chart, and other properties such as chart name, description and legend. While this could be fine in a regular Silverlight control development experience, this is not necessary ideal for LightSwitch. It is important to consider the power of the visual tree, look at how a data grid or list control is used; there are in fact child controls that allow the user to do further manipulation without having to go to the properties

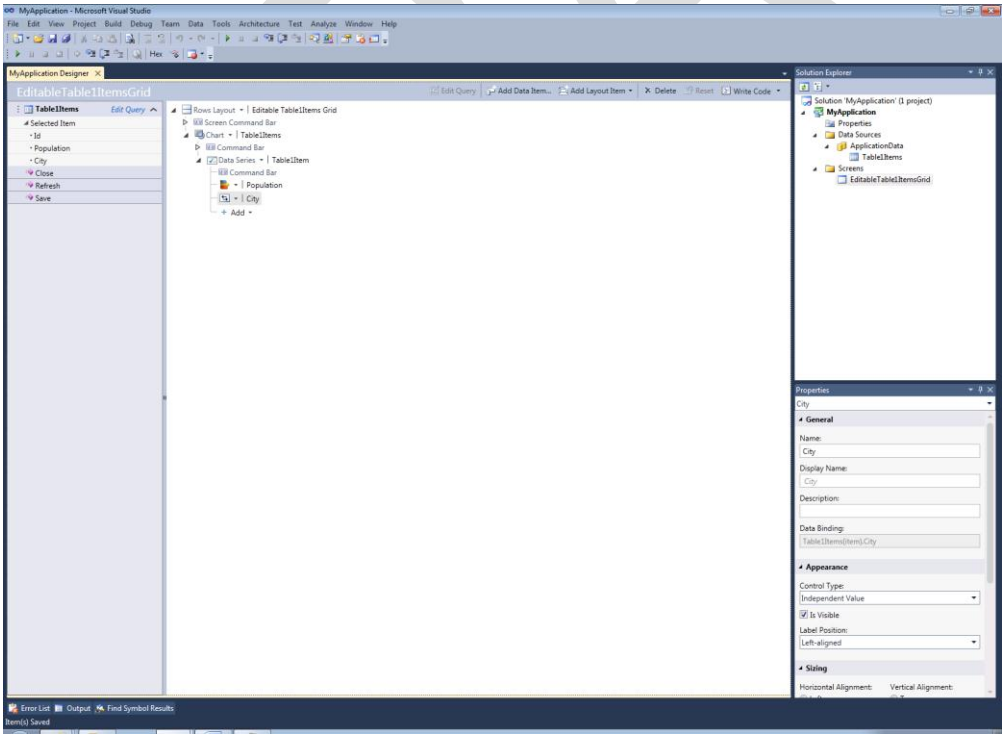
pane. The same could be applied to the chart control where you could have child controls that allow the selection of different chart types, as well as the different values applied to the selected chart.

The following is an illustration to show you the differences in the visual tree.

Single control with properties:



Multiple controls with a set hierarchy:

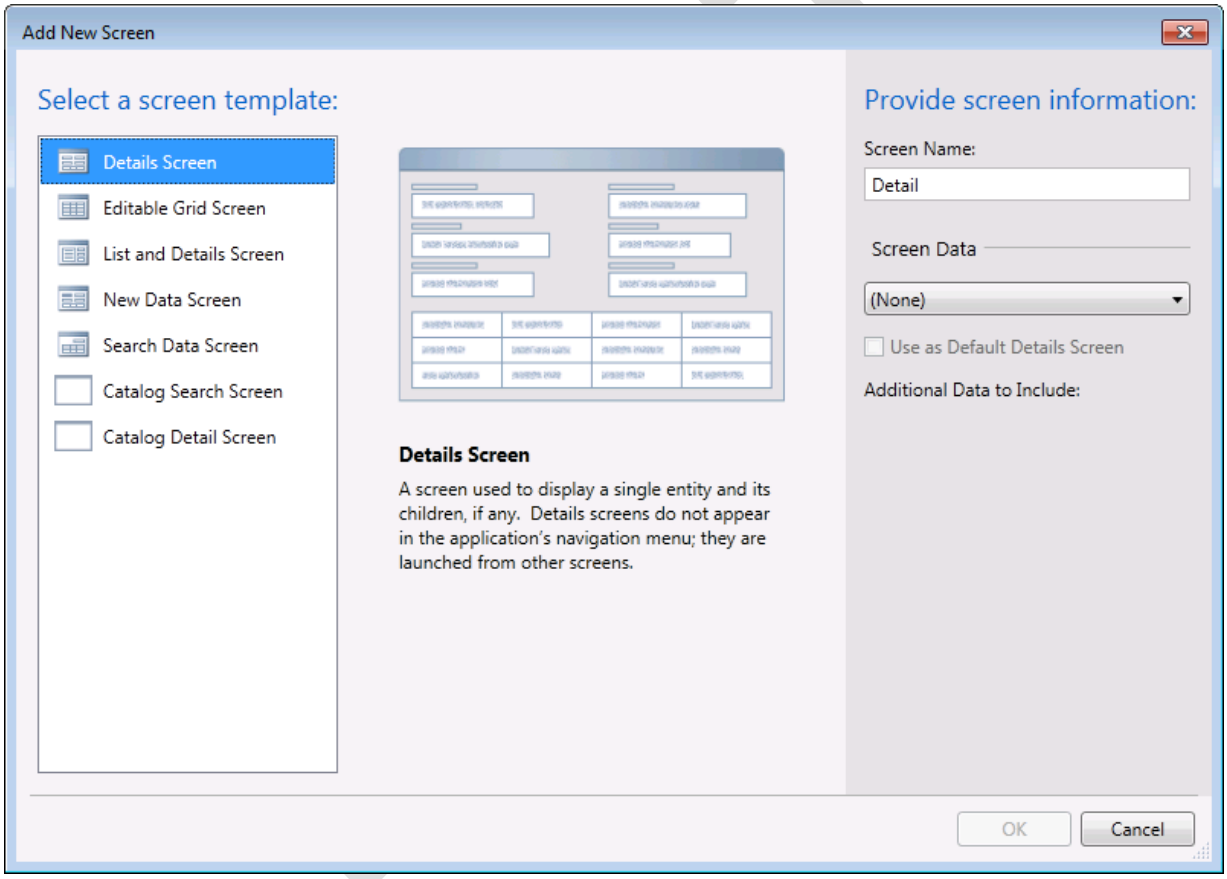


You can see in the screen shot above that there are multiple child controls under the Data Series control. This is an important factor to consider when developing collection of controls to use the visual tree in the screen designer to provide a similar experience to that of the built-in controls in the product, as the developer already has experience in using the built-in controls.

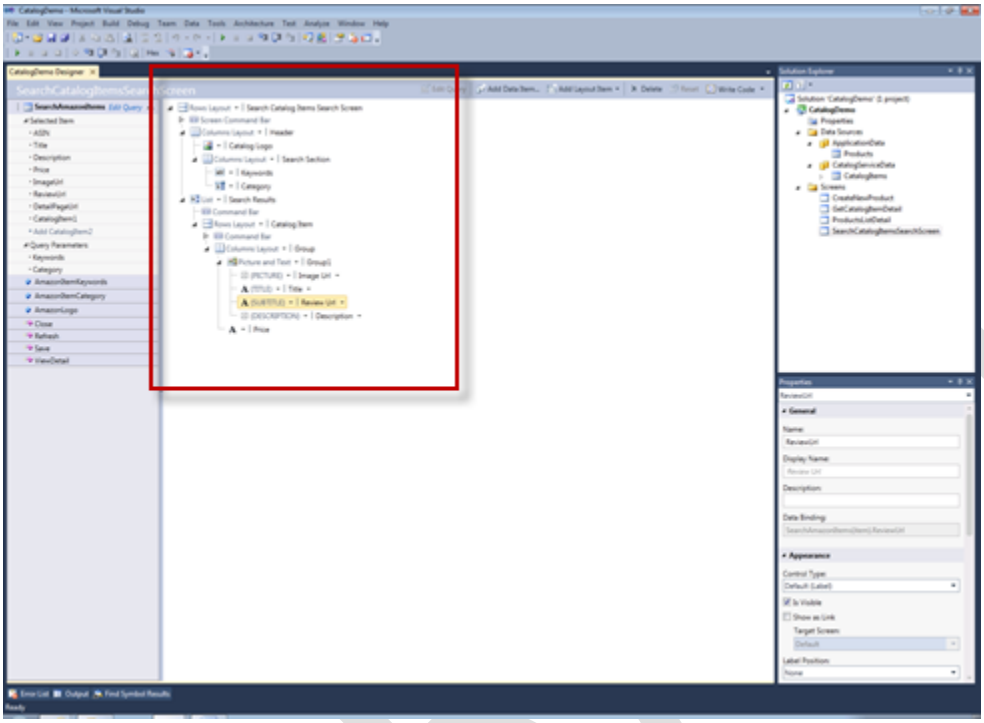
Screen Template

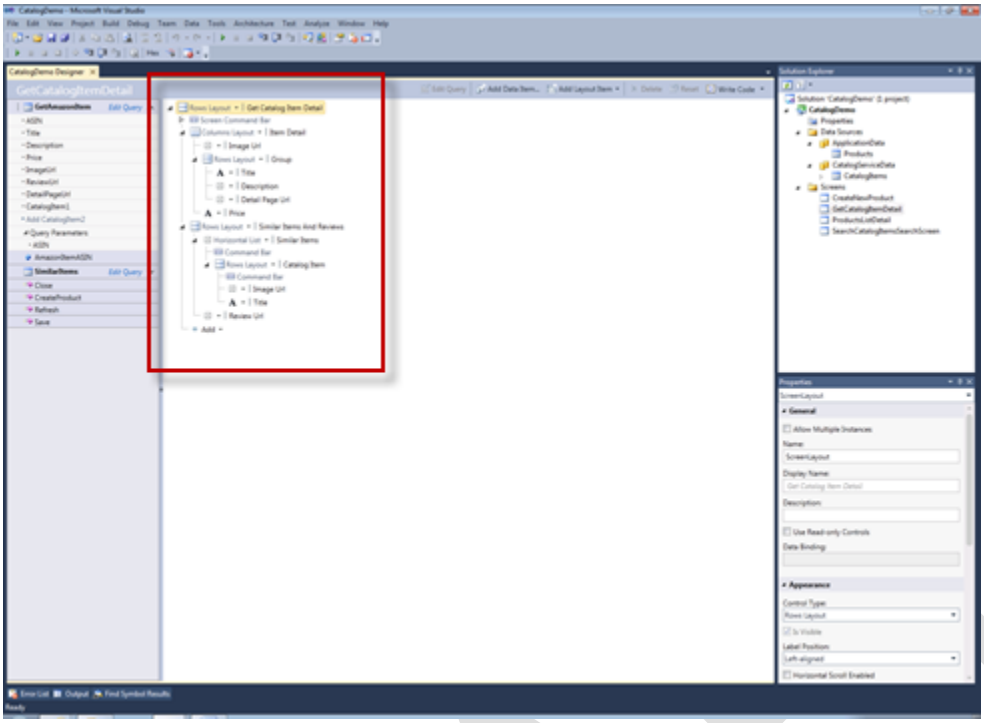
A screen template extension provides a stencil for the layout of controls in a screen designer and for what will appear in a screen in the application during run-time.

A screen template can be used in an application and then removed without affecting existing screens that have been created using the same template. After adding the screen template extension, the screen templates will appear in add new screen dialog, along with several built-in screen templates.



The following is an illustration of a couple of screen templates (Catalog Search Screen and Catalog Detail Screen) and what they provide, in the layout of the various controls and the order in the visual tree hierarchy:





Business Type

Business types provide a way to visualize, format, validate and store information/data in a LightSwitch application based on the unique requirements you may have for your data.

The diagram above is a logical flow of how a business type works, starting from the left at the control and moves through to the right where the data is stored in the storage type and where attributes are adorned on the custom business type.

With a business type extension, it has a way of presenting the data in the UI, based on some formatting rules, validating data when added to application, as well as ensuring the data is validated once the data is saved.

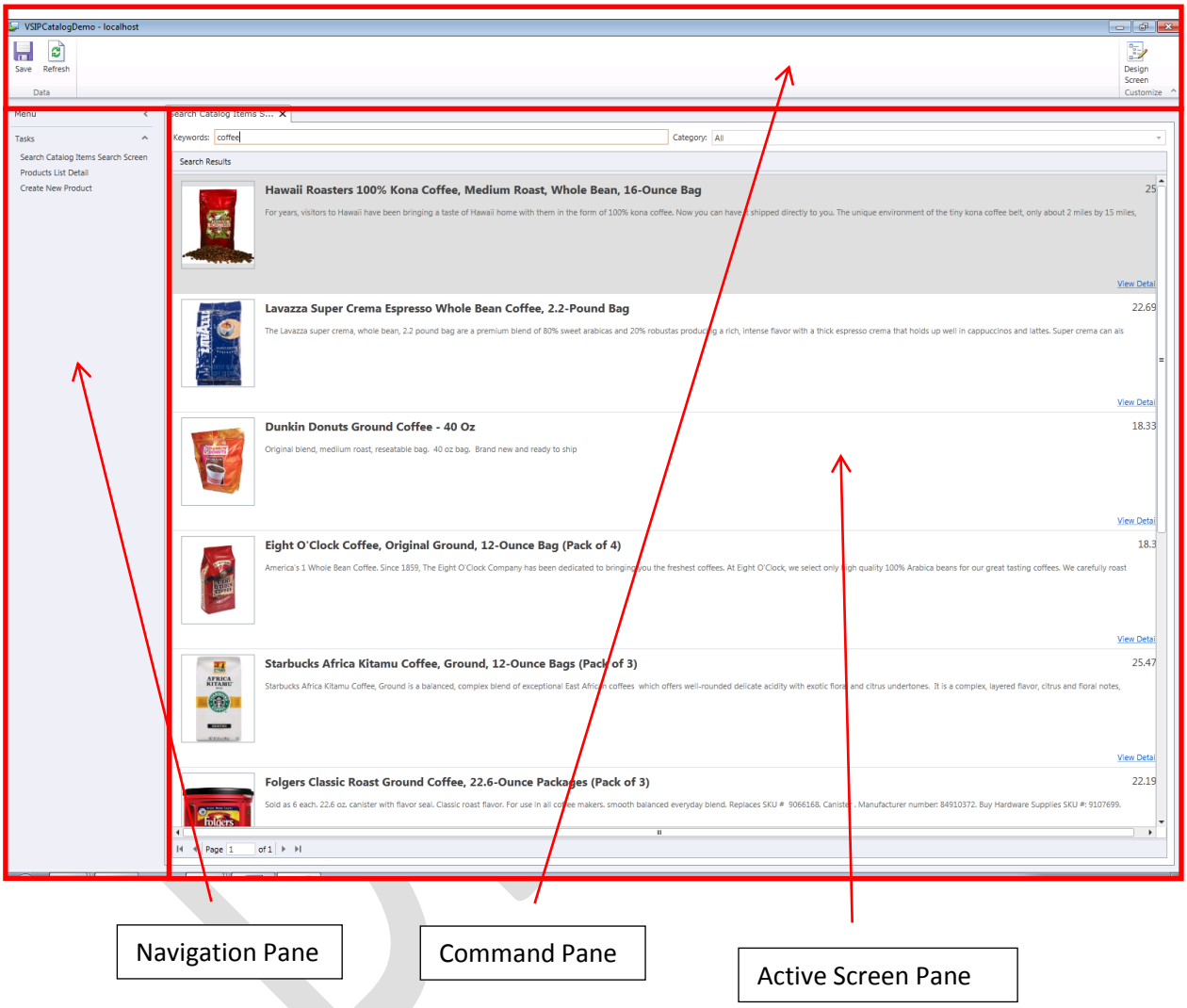
The business type diagram shows two stages of formatting and validation. This is unique to business types as the first series of actions is based on adding data that is dirty (has not been saved), but is still formatted and validated, and then when the save action is done, the data is formatted and validated appropriately again. The reason it is done on the client first is so that you can get immediate feedback to

the user and conserve middle-tier resources. When the user saves the screen the data needs to be formatted and validated at the server; the data will traverse across the network only once.

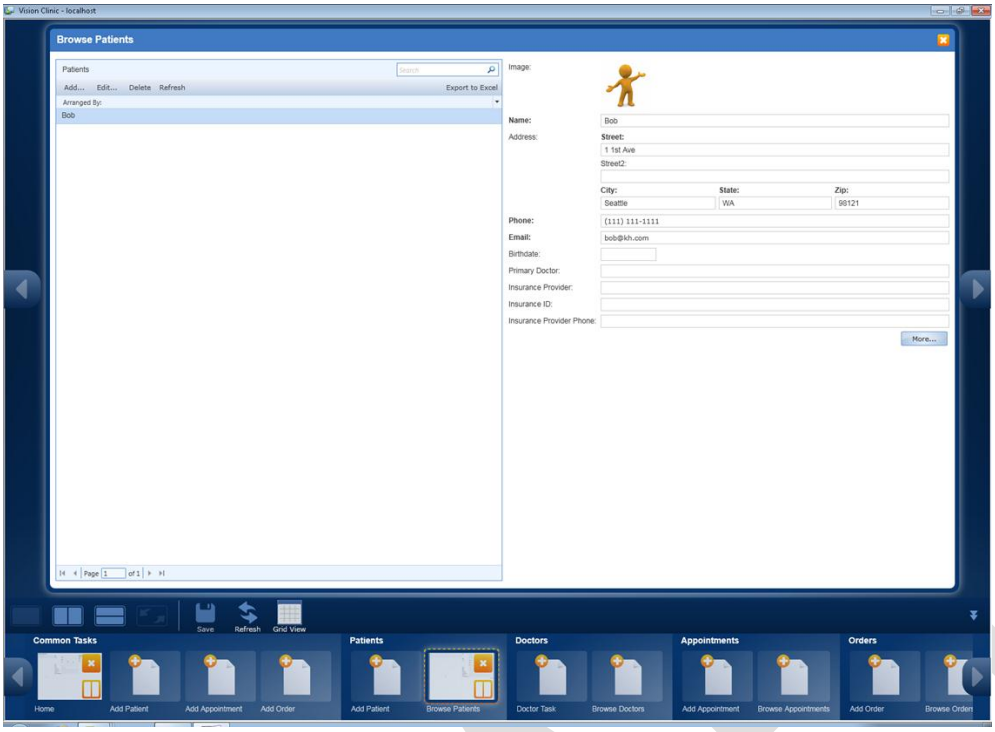
A unique aspect of working with a business type is that you can give a storage type an alias. An example is that the PackagingSKU storage type is string, but for readability purposes the alias is PackagingSKU. The built-in business types such as the PhoneNumber and EmailAddress use this approach, as they are both stored as strings in the database.

Shell

The shell provides the common look and feel of an application. There are three main parts of a shell, which are fundamental to working with LightSwitch Applications. These parts are illustrated in the screenshot below:

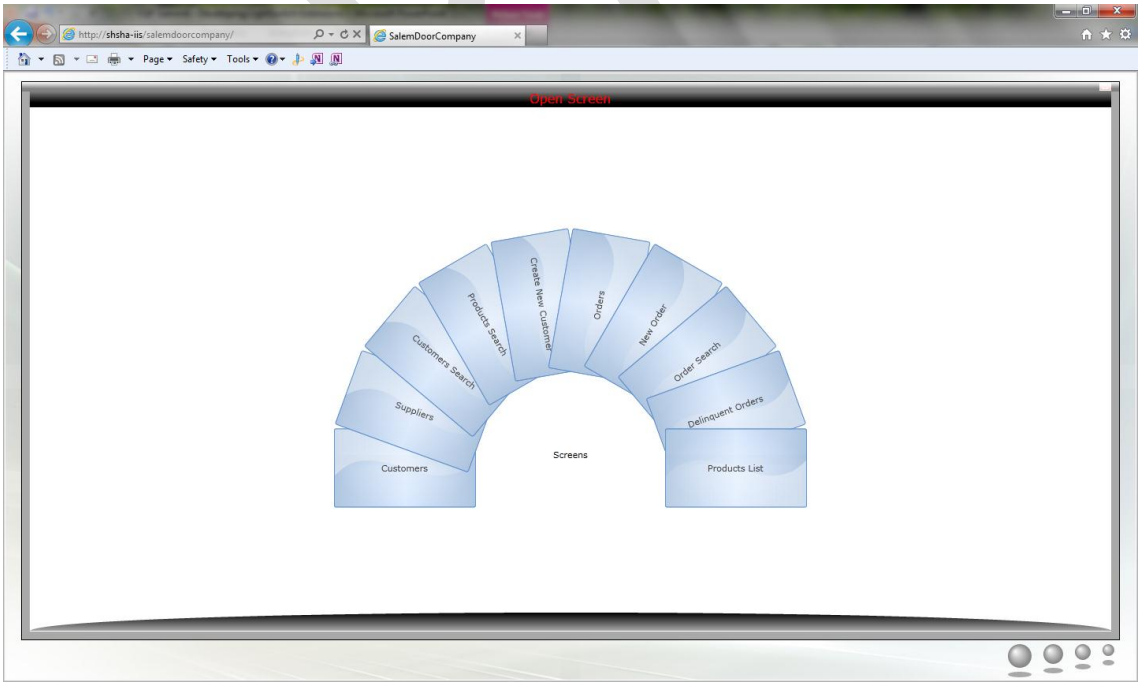


As you can see there are three pane parts; you are free to provide your own layout. You could create your own shell that is more useable touch screens, as shown in the next image.



As you can see in this shell the navigation is at the bottom of the screen and the screen navigation uses large buttons as opposed to links in the default shell.

Below is another shell that leverages specific custom controls to show off the screens in a fanned-out card deck manner.

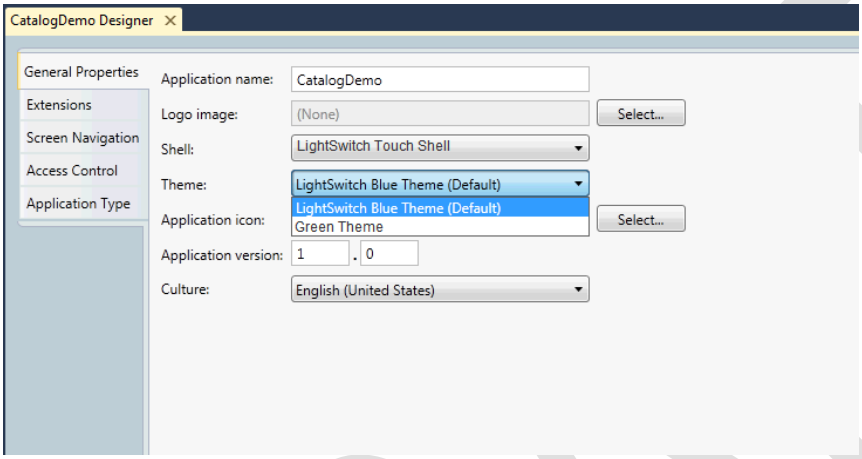


You will learn more about the mechanics of developing a shell in the shell extension recipe.

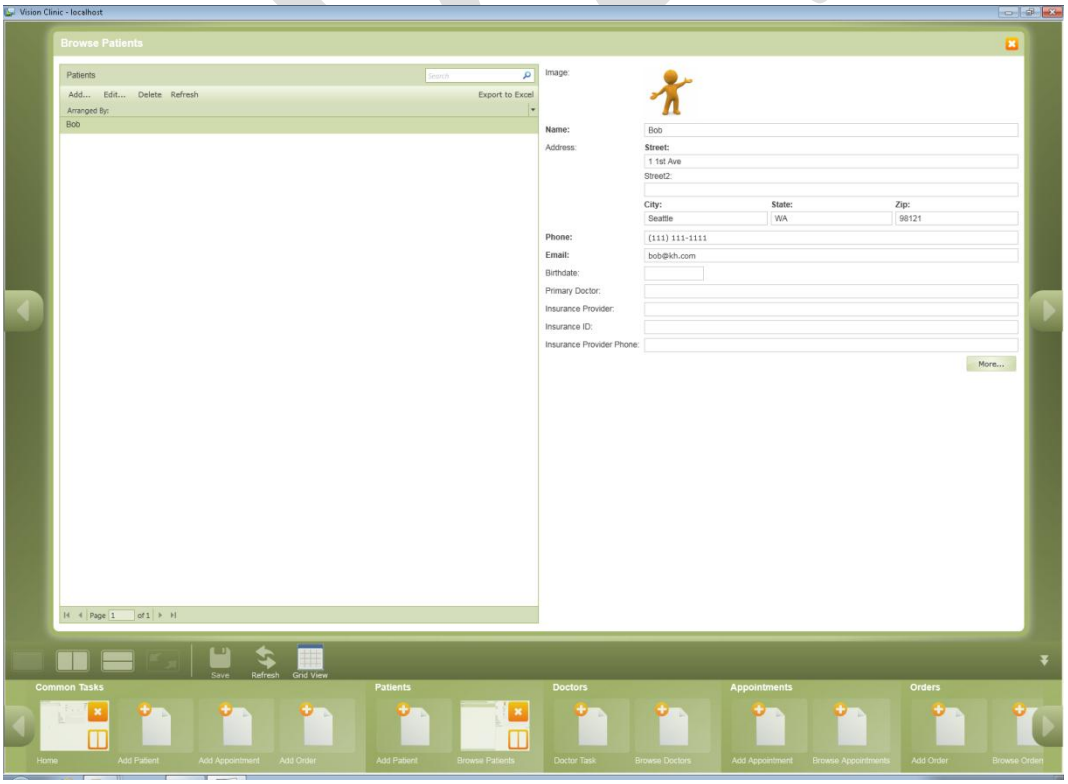
Theme

A theme is a color and font palette; it is an essentially a large resource dictionary with keys. The keys that are use in the recipe are the standard theme keys for the default built-in shell and theme. You can modify these keys to change the colors and fonts in the built-in shell to meet your specific requirements. Themes can be tightly coupled to a custom shell, in that the resource keys may be specific to the custom shell. It may be wise to provide the built-in keys as well just in case the developer wants to switch from one shell to another.

A theme is set within the properities of the LightSwitch application project, as shown below:

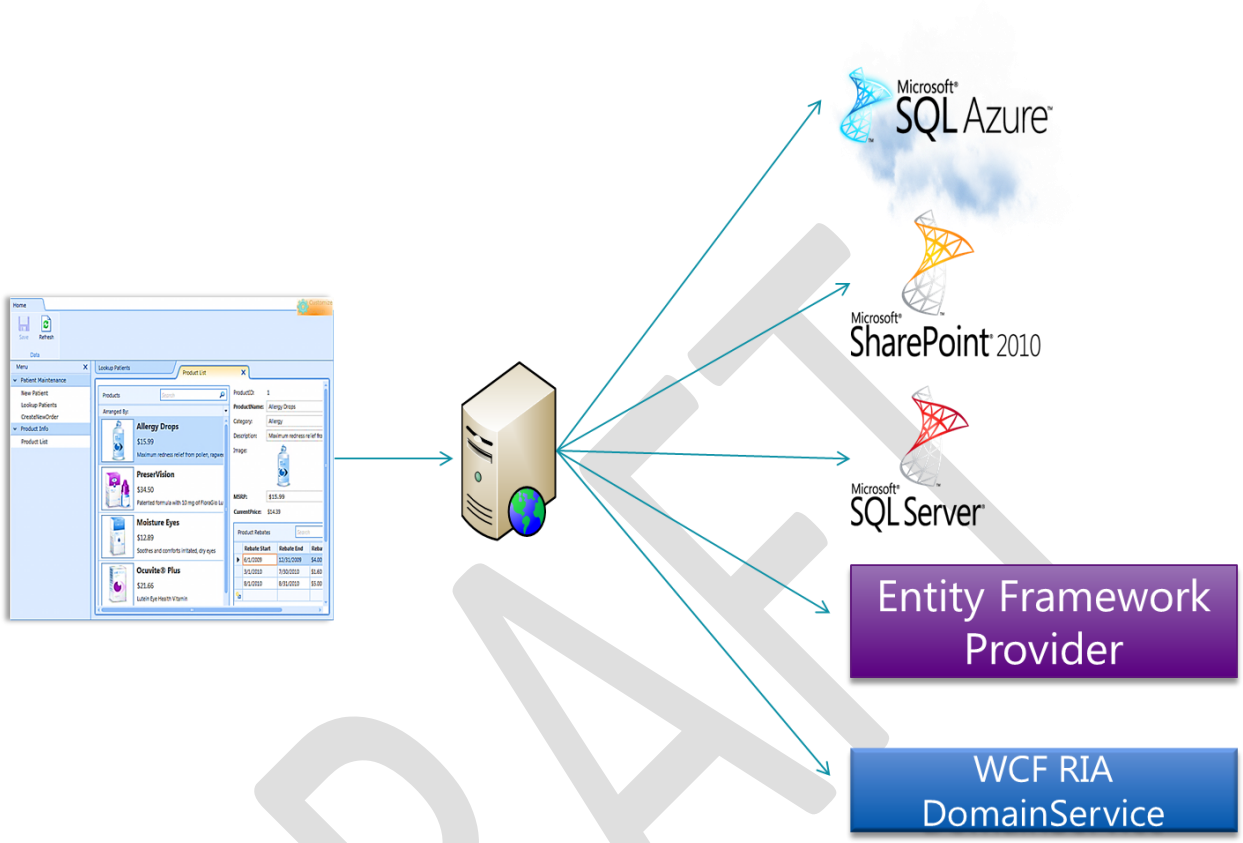


Selecting the Green Theme has the following effect on the touch-friendly shell:



Custom Data Source

A custom data source provides connectivity to other data sources via a WCF RIA Service.



Extension Recipes

This section of the cookbook provides recipes for creating your own extensions. In order, to use these recipes, you will need to have Visual Studio 2010 LightSwitch, Visual Studio 2010 SP1 Professional or higher, the Visual Studio SDK, and the LightSwitch Extensibility SDK which contain the extensibility project templates. The project templates can also be installed via Extension Manager in Visual Studio

> **NOTE:** As this document and Visual Studio LightSwitch are in Beta, the extensibility project templates are not yet available. Instead, a Blank Solution template is being used. It contains all of the projects needed to create any of the extension types. When Visual Studio LightSwitch is released, the extensibility project templates will be released as along with an update to this cookbook.

Control Extension

The control extension recipe enables you to add a Silverlight control or set of controls to a LightSwitch application.

The ingredients that you will to need to create a control extension are the following:

1. Visual Studio 2010 Professional or higher
2. Visual Studio 2010 Service Pack 1
3. Visual Studio 2010 SDK
4. Visual Studio LightSwitch
5. LightSwitch Blank Extension Solution

Once you have all of the ingredients installed on your machine you are ready to create a control extension.

The first step is to get you set up with the right environment in order to add the specific code that you want your control extension to display and use. This is done by opening the Blank Extension solution sample, which is called BlankExtension.sln. (You can rename the solution to any name you want to use - at the moment we will keep using the name of Blank.

The next step when creating a control extension is to focus on the metadata in the common project.

In the common project you will create control that stores a scalar value based on string. In the BlankPresentation.lsml file you need to add the control metadata definition. Add the following code:

```
<?xml version="1.0" encoding="utf-8" ?>
<ModelFragment
  xmlns="http://schemas.microsoft.com/LightSwitch/2010/xaml/model"
```

```
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <Control
     Name="MyControl"
     SupportedContentItemKind="Value"
     DesignerImageResource="Microsoft.LightSwitch.DesignerImages::TextBox">
    <Control.Attributes>
      <DisplayName Value="$(MyControl_DisplayName)" />
    </Control.Attributes>
    <Control.SupportedDataTypes>
      <SupportedDataType DataType=":String"/>
    </Control.SupportedDataTypes>
  </Control>

</ModelFragment>
```

The above code contains the control name and the SupportedContentItemKind, which in this case is Value. The DesignerImageResource is using the image from a built in control. Later you will learn how to add your own icon image.

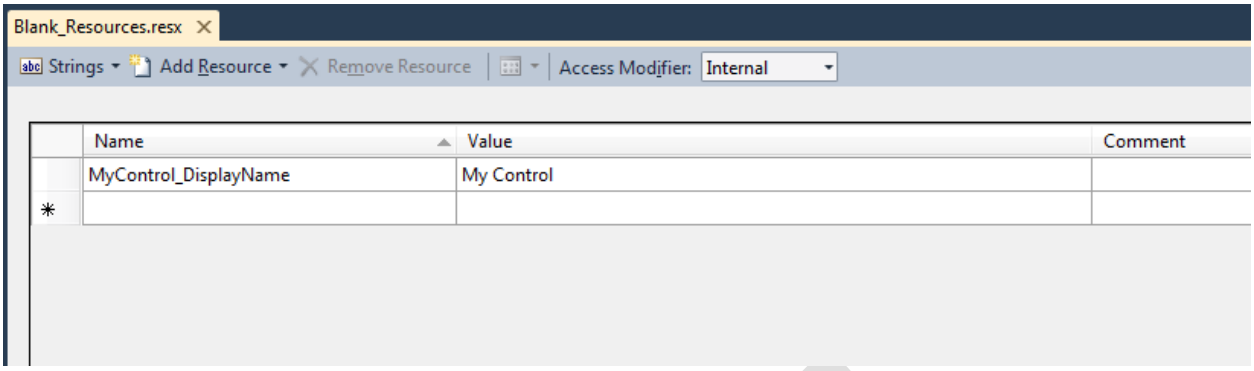
The DisplayName is what will appear in the screen designer and is set from a resource file.

The SupportedDataType specifies the type of data that the control can accept, which is mentioned earlier in this document.

Next you need to specify a constant for the new control. Update the BlankModule.cs class file with the following code:

```
namespace Blank
{
    internal static class BlankModule
    {
        public const string Name = "Blank";
        public const string ModulePrefix = Name + ":";

        internal static class MyControl
        {
            private const string Name = "MyControl";
            public const string GlobalName = ModulePrefix + Name;
        }
    }
}
```

Update the Blank_Resources.resx file to have the following name/value pair:



All you have done so far is describe the control for use in design-time of LightSwitch. Next let's look at the implementation of the control.

In the Client project, create a Controls folder.

In the controls folder, add a Silverlight User Control and name it MyControl.xaml.

In the MyControl.xaml file, add the Silverlight TextBox control, as shown in the code below:

```
<UserControl x:Class="Blank.Controls.MyControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400">

    <TextBox x:Name="MyCtl" Text="{Binding Details.Value, Mode=TwoWay}"
TextWrapping="NoWrap"/>

</UserControl>
```

What is important here is the binding path to Details.Value, which is provided via the IContentItem interface from the View through the ViewModel to the Model. The reason for this is to provide a way to interact with an entity and allow within the screen designer that a entity type can be bound (set) to a control based on the developers preference and what is defined in the supported data type.

In the code-behind file MyControl.xaml.cs you need to specify the control factory, which allows LightSwitch through the model lsml to get the visual of the control shown in the code above.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
```

```
using Microsoft.LightSwitch.Presentation;
using System.ComponentModel.Composition;

namespace Blank.Controls
{
    public partial class MyControl : UserControl
    {
        public MyControl()
        {
            InitializeComponent();
        }
    }

    [Export(typeof(IControlFactory))]
    [ControlFactory(BlankModule.ControlName)]
    internal sealed class MyControlFactory : BaseControlFactory
    {
        #region Constructor

        public MyControlFactory() : base("MyControlTemplate") { }

        #endregion
    }

}
```

You now need to update the BlankModule.cs file to include a constant of ControlName. This is done inside the BlankModule class. Add the following code:

```
public const string ControlName = ModulePrefix + "MyControl";
```

The control factory needs some additional files which you will create next.

The next file you need to create is the BaseControlFactory file. This file contains boilerplate code that allows the data template file that contains a reference to the control to be loaded.

The following is the BaseControlFactory.cs class file that needs to be created, though it could be placed in the Common project and linked in the Controls folder in the Client project:

```
using System;
using System.Diagnostics;
using System.IO;
using System.Reflection;
using System.Windows;
using System.Windows.Markup;
using System.Windows.Resources;

using Microsoft.LightSwitch.Presentation;

namespace Blank
{
    [DebuggerNonUserCodeAttribute]
    internal abstract class BaseControlFactory : IControlFactory
    {
```

```
        #region Constructors

        static BaseControlFactory()
        {
            // Ger resource
            Uri resourceUri = GetResourceUri();
            StreamResourceInfo streamInfo =
System.Windows.Application.GetResourceStream(resourceUri);
            Debug.Assert(null != streamInfo, "ResourceDictionary load failed.", "Stream
not found: {0}", resourceUri.ToString());

            if (null != streamInfo)
            {
                // Read resource contents
                string contents = string.Empty;
                using (var reader = new StreamReader(streamInfo.Stream))
                {
                    contents = reader.ReadToEnd();
                }

                // Ensure contents is not empty
                Debug.Assert(!String.IsNullOrEmpty(contents), "ResourceDictionary load
failed.", "Resource was empty: {0}", resourceUri.ToString());

                // Load the contents as resource dictionary
                BaseControlFactory.Dictionary = XamlReader.Load(contents) as
ResourceDictionary;
                Debug.Assert(null != BaseControlFactory.Dictionary, "ResourceDictionary
load failed.", "Resource wasn't ResourceDictionary: {0}", resourceUri.ToString());
            }
        }

        public BaseControlFactory(string templateKey)
        {
            _dataTemplate = BaseControlFactory.GetDataTemplate(templateKey);
        }

        #endregion

        #region IControlFactory Members

        DataTemplate IControlFactory.DataTemplate
        {
            get
            {
                return _dataTemplate;
            }
        }

        DataTemplate IControlFactory.GetDisplayModeDataTemplate(IContentItem contentItem)
        {
            return null;
        }

        #endregion

        #region Private Methods
```

```
        private static DataTemplate GetDataTemplate(string templateKey)
        {
            Debug.Assert(BaseControlFactory.Dictionary.Contains(templateKey),
"DataTemplate not found.", "Looking for resource key '{0}'", templateKey);

            return BaseControlFactory.Dictionary[templateKey] as DataTemplate;
        }

        private static Uri GetResourceUri()
        {
            Assembly assembly = Assembly.GetExecutingAssembly();
            AssemblyName thisAssemblyName = new AssemblyName(assembly.FullName);
            return new Uri(thisAssemblyName.Name +
";component/Controls/DataTemplates.xaml", UriKind.Relative);
        }

        #endregion

        #region Private Fields

        private DataTemplate _dataTemplate;

        private static ResourceDictionary Dictionary;

        #endregion
    }
}
```

The next file you will create in the same location is the DataTemplates.xaml file. The following is the code required in this file based on the MyControl.

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:c="clr-namespace:Blank.Controls;assembly=Blank.Client">

    <DataTemplate x:Key="MyControlTemplate">
        <c:MyControl />
    </DataTemplate>

</ResourceDictionary>
```

Make sure that the build action on this file is set to Page, which is the default.

At this point using the Blank Solution, you have a default project which is basic. In order to use the extension, you need to go to the VSIX project folder. In the bin\debug or bin\release folder is the blank.vsix.VSIX file; clicking on it installs the extension, making it available for your next LightSwitch application. You will likely want to test the extension before using it in an actual LightSwitch application. To provide a debugging experience with Visual Studio 2010 and LightSwitch, you can configure the VSIX properties to start LightSwitch and set a test LightSwitch application test your extension.

In order for your project to include the test LightSwitch application, you need to tell the extension what solution file to use. This is only necessary for debugging.

The following steps need to be done in the debug page of the project properties of the VSIX project within the extension solution:

Section	Property	Value
Start Action	Start external program:	Select radio button and set the path location to the devenv.exe executable. The default is: C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\devenv.exe
Start Options	Command line arguments:	At the beginning specify the /rootSuffix Exp and then the path to the LightSwitch application and solution file (.sln).

After setting these properties, ensure that the VSIX project is the startup project. Setting these properties in the extension Visx project will load the extension in Visual Studio's experimental hive, which means that the Visual Studio build task will load and unload the extension into the LightSwitch project each time you debug.

So what options are available next? The following toppings can be added to this recipe:

1. Custom inline properties
2. Different presentation items for the control
3. Different control styles, such as collection, group controls.

Let's look at adding these toppings one by one.

Custom Inline Properties:

To create additional properties for a control extension, there are two areas that require modifying. These are the presentation metadata, which are the control.lsml and the code behind the control itself.

Let's look at modifying the control presentation lsml file.

To add a custom property you can add the following:

```
<Control.Properties>

      <!-- My Property -->
      <ControlProperty Name="MyProperty"
        PropertyType=":String"
        IsReadOnly="False"
        CategoryName="My Properties"
        EditorVisibility="PropertySheet">
        <ControlProperty.Attributes>
          <DisplayName Value="$(MyProperty_DisplayName)"/>
          <Description Value="$(MyProperty_Description)"/>
        </ControlProperty.Attributes>
      </ControlProperty>
      <!-- End Control My Property-->

</Control.Properties>
```

The important thing about this XML code is that the presentation part is taken care for you. When the Property type is set to :String, a textbox is added to the property sheet.

The code above also specifies the category as well. If this item is omitted then it will be placed under the Appearance category.

Properties are either opt-in or opt-out:

Opt-out properties are enabled in the property sheet by default and also have behavior that is provided by default by LightSwitch for the control.

Opt Out:

AttachedLabelPosition

- Enabled by default, you don’t need to do anything to support standard attached labels. See also IPresentationDefinition.AttachedLabelSupport
- To display your own attached label, set the control's AttachedLabelSupport to "DisplayedByControl" (defaults to DisplayedByContainer)
- If you don't support attached labels at all, then you should also override the property to EditorVisibility="NotDisplayed"

HeightSizingMode

WidthSizingMode

MinHeight

MaxHeight

MinWidth

MaxWidth

Height

Width

- These are all displayed by a single editor. In general we recommend leaving these on. If for some reason you need to turn these off, you can override HeightSizingMode's EditorVisibility to "NotDisplayed".

NOTE: It also displays Characters/Lines from TextBox/Label. In a future build we hope to have a way for third-party controls to have their own characters/line properties displayed by this editor as well.

HorizontalAlignment

VerticalAlignment

- These are all displayed by a single editor (along with WeightedRowHeight, WeightedRowWidth, which we also hope to have a mechanism for third party properties).
- If you need to turn these off, you can override VerticalAlignment's EditorVisibility to "NotDisplayed"

Opt-in controls are hidden in the property sheet by default. They can be displayed in the property sheet, but may require the control to do work for the property to have any effect.

Opt In:

ShowAsLink

- Override EditorVisibility="PropertySheet"
- Then you can use the built-in Microsoft.LightSwitch.Presentation.Framework.LinkableLabel to get the built-in label-or-hyperlink behavior, or use your own UI and use the Microsoft.LightSwitch.Presentation.Framework.Helpers.ShowAsLinkPropertyHelper class

FontStyle

- To support FontStyle, you just need to set EditorVisibility="PropertySheet", and it will be handled automatically by IContentItem

TextAlignment

- You'll need to override EditorVisibility and also provide your own implementation

BrowseOnly

- Just override EditorVisibility="PropertySheet"

IsReadOnly

IsEnabled

- These are for internal uses and should not be displayed in the property sheet

ContainerState

- This should never be displayed in the property sheet. It is used to allow controls to change their UI or behavior based upon context in the tree. The only standard values currently defined are "None" and "Cell". "Cell" indicates that the control is inside of a control that displays items that look like cells, e.g., like the DataGrid.
- If you are creating a control like the DataGrid and want to have controls change their appearance in a similar manner, set the value of this property to "Cell" on your children, e.g.:

```
<Control.ChildItemPropertySources>
```

```
<!-- Set ContainerState for descendants to "Cell", so they can draw themselves differently
inside a datagrid -->

<ControlPropertySource Property="RootControl/Properties[ContainerState]">

 <ControlPropertySource.Source>

  <ScreenExpressionTree>

   <ConstantExpression ResultType="String" Value="Cell" />

  </ScreenExpressionTree>

 </ControlPropertySource.Source>

</ControlPropertySource>


</Control.ChildItemPropertySources>
```

- If you are creating a control that wants to change it behavior in response to its ContainerState, you can check IContentItem.ContainerState (or find it in IContentItem.Properties), and respond in code, or perhaps via a visual state.

Image

You'll need to override EditorVisibility and also provide your own implementation. You should use ImagePropertyHelper in Microsoft.LightSwitch.Presentation.Framework.Helpers to interpret the property settings in IContentItem and retrieve the image to display.

The control property attributes provide a display name and description which are also displayed on the property sheet. The description appears as a tooltip when mouse over the property.

The next step is to bind this property to the control. Open the control class file Mycontrol.xaml.cs

In this file you need to create a dependency and getter and setter properties as well as the callback to the property.

The following code you can add to MyProperty:

```
public static readonly DependencyProperty MyProperty =
            DependencyProperty.Register(
                Control1.MyProperty,
                typeof(string),
                typeof(MyControl),
                new PropertyMetadata(OnMyPropertyChanged));
```

```
public string MyProperty
        {
            get { return (string)base.GetValue(Control1.MyProperty); }
            set { base.SetValue(Control1.MyProperty, value); }
        }


private static void OnMyPropertyChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
        {
            ((Control1)d).OnMyPropertyChanged((string)e.OldValue, (string)e.NewValue);
        }

        private void OnMyPropertyChanged(string oldMyProperty, string newMyProperty)
        {
              MyControl.Background = newMyProperty;
        }
```

In the property topping above, there is not a lot going on except setting the background color of textbox control to whatever was set in the MyProperty property. Feel free to add you own property topping here to add more spice to your control.

Try experimenting with different properties for your control.

Different Presentation items to the Control:

The default control is using the default added presentation items, which is shown in this code snippet.

```
<f:AttachedLabelItemPresenter LabelStyleId="AttachedLabelStyle">
        <f:ContentSizingControl
            ContentSizingMode="{Binding
Properties[Microsoft.LightSwitch:RootControl/ContentSizingMode]}"
            ContentSize="{Binding
Properties[Microsoft.LightSwitch:RootControl/ContentSize]}">
            <Textbox x:Name="Control1" />
        </f:ContentSizingControl>
    </f:AttachedLabelItemPresenter>
```

The two properties shown above provide control over the sizing mode and content size. You can remove or add additional properties to your control if needed.

Try experimenting with the different properties for your control.

Control Styles (Group and Collection based):

The control that you have been creating so far is a value based control. This is fine if you have a single data point to display in a control.

This topping allows you to group controls together as well as provide a collection of data into a control such as a datagrid control.

Let's add a new group control in the current solution package.

You will modify the BlankPresentation.lsml file to support more control styles in the screen designer when interacting with the visual tree. Modify the default model code to the code below or delete the default code and add the following code:

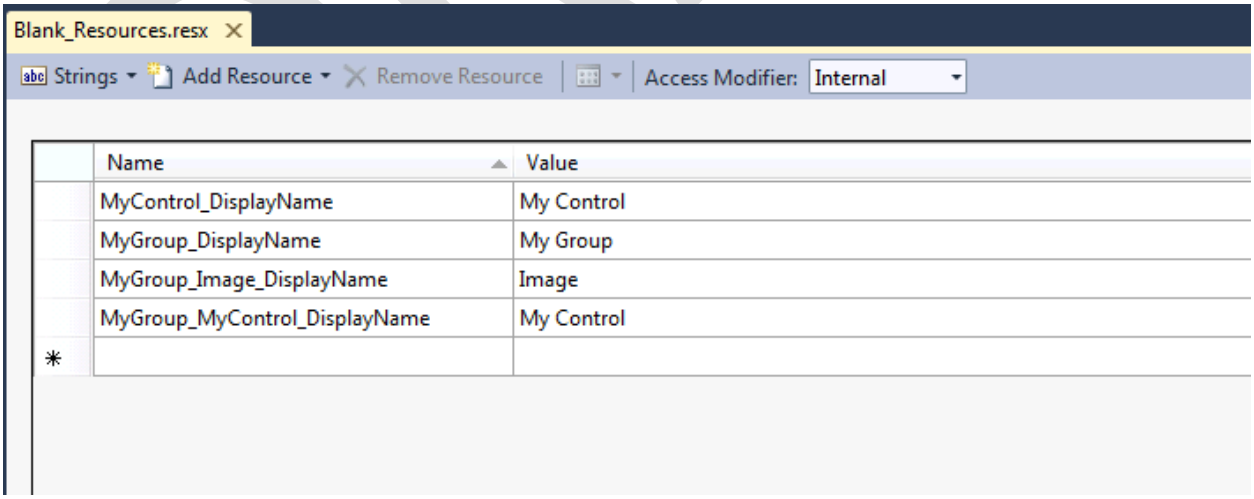
```
<Control Name="MyGroup"
    SupportedContentItemKind="Group"
    DesignerImageResource="Microsoft.LightSwitch.DesignerImages::Label">
    <Control.Attributes>
      <DisplayName Value="$(MyGroup_DisplayName)" />
    </Control.Attributes>
  </Control>
```

In the example above, the code allows a control to be a group control in the visual tree, indicated by the SupportedContentItemKind set to Group.

If you try to debug the application you will see that there is minimal visual implementation of the group control in the run time so far:,you will see the group and you can add child controls under it. Let's now provide hints or what we call placeholders to the developer in the visual tree of what type of controls the group control is expecting. To do this, you need to modify the BlankPresentation.lsml to have the following placeholders:

```
<Control.Placeholders>
      <Placeholder DisplayName="$(MyGroup_Image_DisplayName)" Name="Image" />
      <Placeholder DisplayName="$(MyGroup_MyControl_DisplayName)" Name="Control" />
</Control.Placeholders>
```

The resource file in the common project will also need to be updated as shown in this figure.



Let's now add some code to have the group appear with the correct selection of controls in the

application run-time. In the Client project, create a MyGroup Silverlight user control. In the MyGroup.xaml file add the following code:

```
<UserControl x:Class="Blank.Controls.MyGroup"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:framework="clr-
namespace:Microsoft.LightSwitch.Presentation.Framework;assembly=Microsoft.LightSwitch.Cli
ent"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400">

    <Grid Name="GroupLayout">
        <Grid.ColumnDefinitions>
            <!-- The rest -->
            <ColumnDefinition/>
            <!-- Image -->
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>

        <StackPanel Orientation="Vertical" Grid.Column="0" >
            <!-- My Control -->
            <framework:ContentItemPresenter
            Name="Control"
            ContentItem="{Binding ChildItems[1]}"/>

        </StackPanel>
        <!-- Image (in the right column) -->
        <framework:ContentItemPresenter
            Name="image"
            Grid.Column="1"
            HorizontalAlignment="Stretch"
            VerticalAlignment="Stretch"
            Margin="5,0,0,0"
            ContentItem="{Binding ChildItems[0]}"
            />

    </Grid>
</UserControl>
```

In this code there is a Silverlight grid control and then a stack panel with the control and image properties. The code to take notice of here is the ContentItemPresenter within the LightSwitch Presentation framework assembly that allows you to set the ContentItem on the binding of childitems.

The code-behind file of MyGroup.xaml.cs is similar to a LightSwitch control:

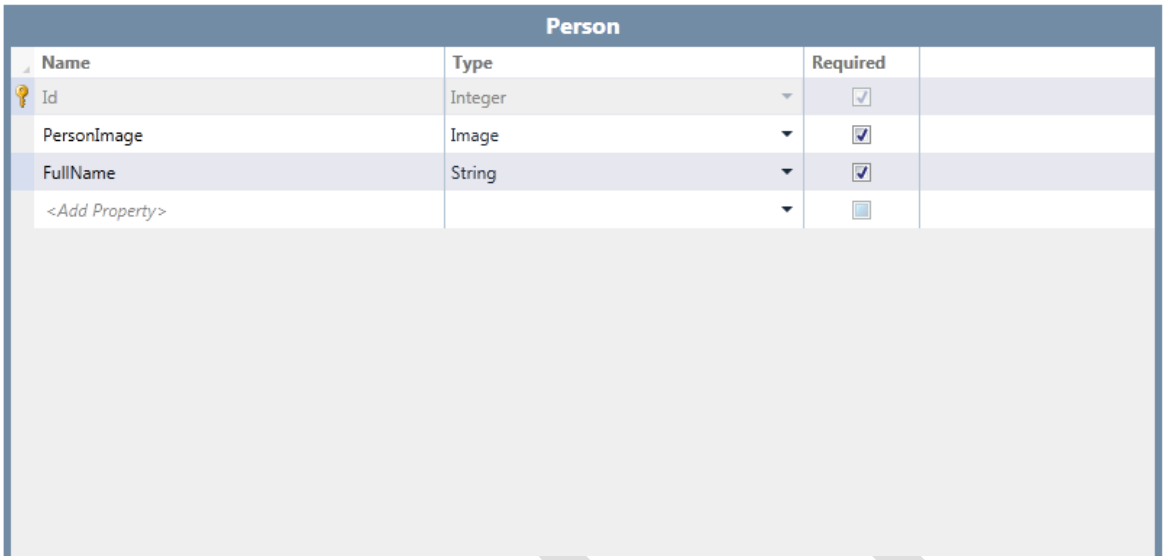
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Windows;
```

```
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;

using Microsoft.LightSwitch.Presentation;
using System.ComponentModel.Composition;
using Microsoft.LightSwitch.Model;
using Microsoft.LightSwitch.Presentation.Framework;

namespace Blank.Controls
{
    public partial class MyGroup : UserControl
    {
        public MyGroup()
        {
            InitializeComponent();
        }
    }

    [Export(typeof(IControlFactory))]
    [ControlFactory(BlankModule.GroupName)]
    internal sealed class MyGroupFactory : BaseControlFactory
    {
        #region Constructor

        public MyGroupFactory() : base("MyGroupTemplate") { }

        #endregion
    }
}
```

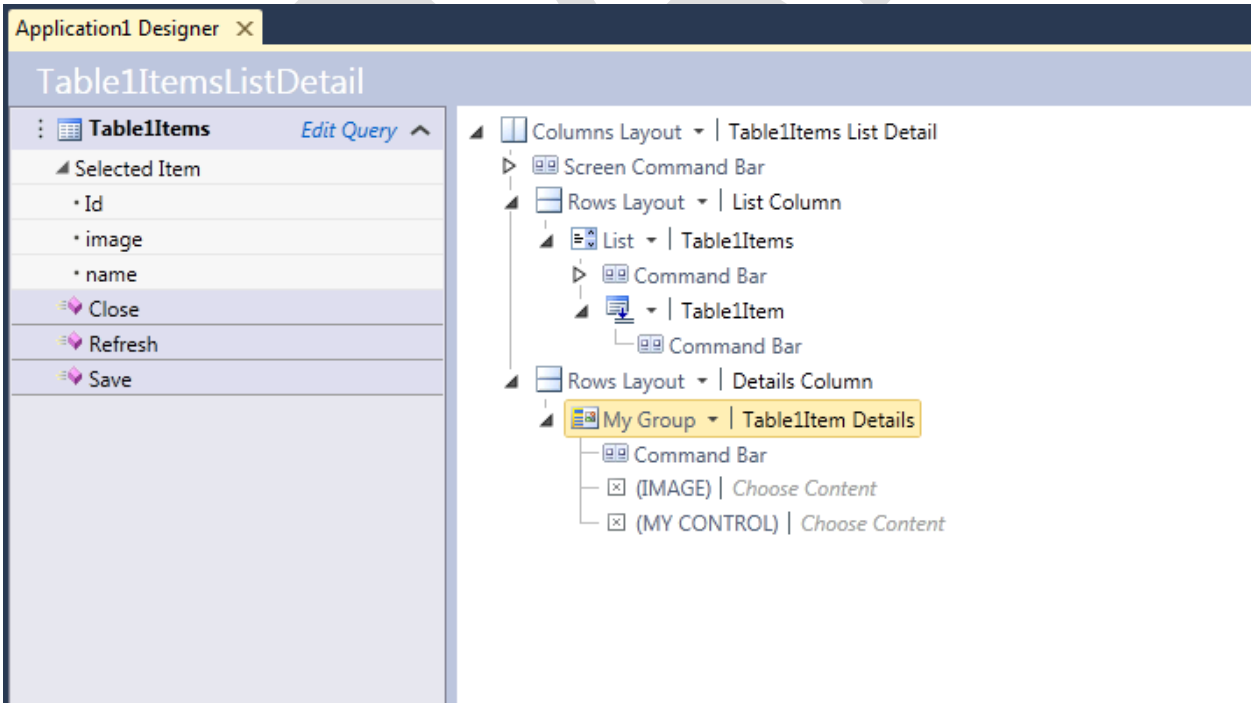
The other update you need to do is to add the control definition to the data template. In the DataTemplates.xaml file add the following:

```
<DataTemplate x:Key="MyGroupTemplate">
        <c:MyGroup />
</DataTemplate>
```

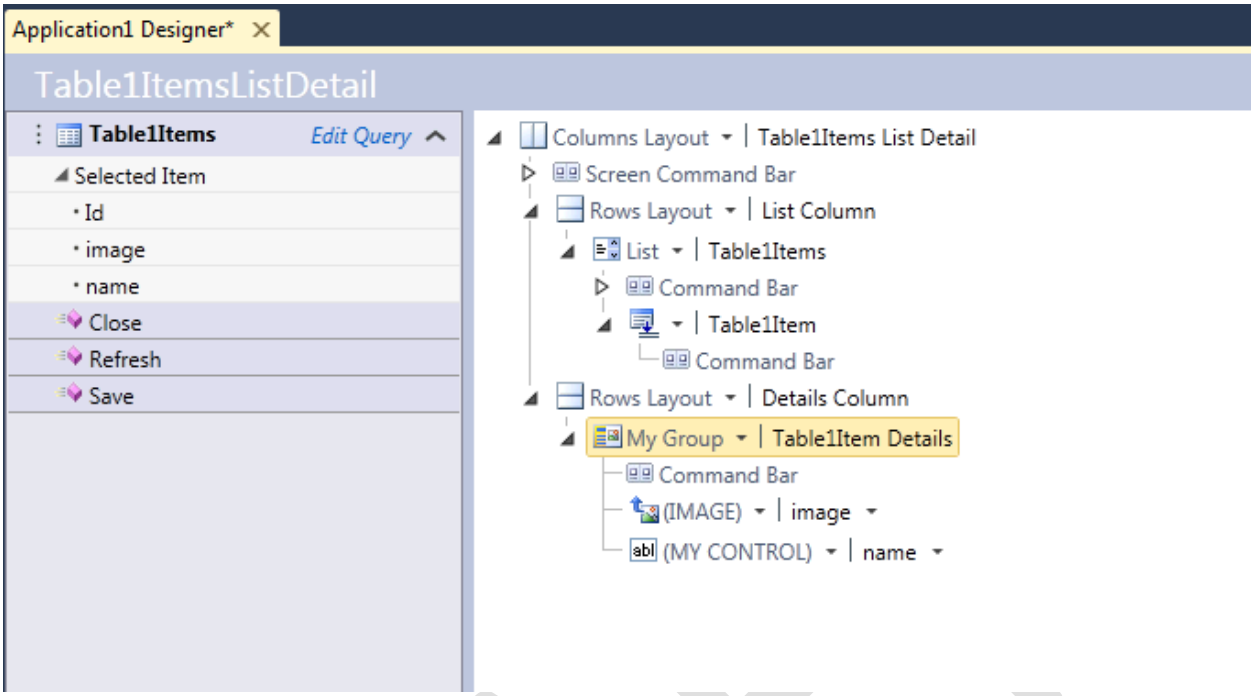
At this point, you can debug the extension in a LightSwitch application. You will need to create the necessary storage types in the entity of the application, such as String and Image.



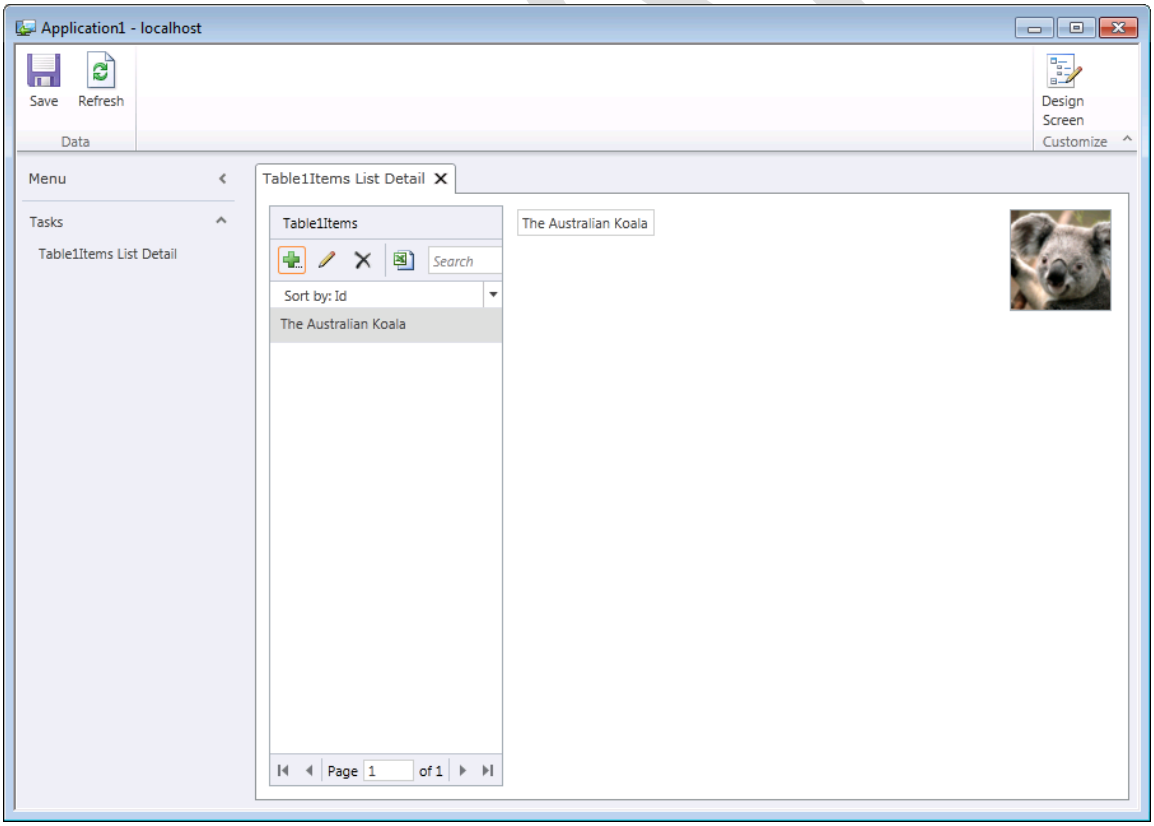
In the screen, select the List and Details screen template and then select MyGroup control and you should have a similar experience to the following:



Then you will be able to select the control that match the placeholder text, as shown in the next figure:



If you debug the LightSwitch application you will see the controls placed based on the MyGroup.xaml, which was a two column, one row grid, and which is shown below:



You have now created your first group control.

Now let's move on to create a collection control. This kind of control is commonly used in a datagrid. In fact, this part of the recipe is going to step you through creating a simple data grid extension.

You will continue to use the same blank solution that has been used so far in this recipe.

The first part of creating a collection control is to define the metadata. With a datagrid there is a fair amount of metadata, due to the fact that a LightSwitch developer can specify the make up of a data row in the data grid by selecting the controls that are children to the datagrid row. The DataGrid control is a Collection control, and the DataGridRow is a Group control. There are a number of properties that need to be modified as well, such as no labels by the controls (AttachedLabelPosition). The code contains comments that indicate what and why the properties are setup the way they are.

Open the BlankPresentation.lsml file and add the following metadata code:

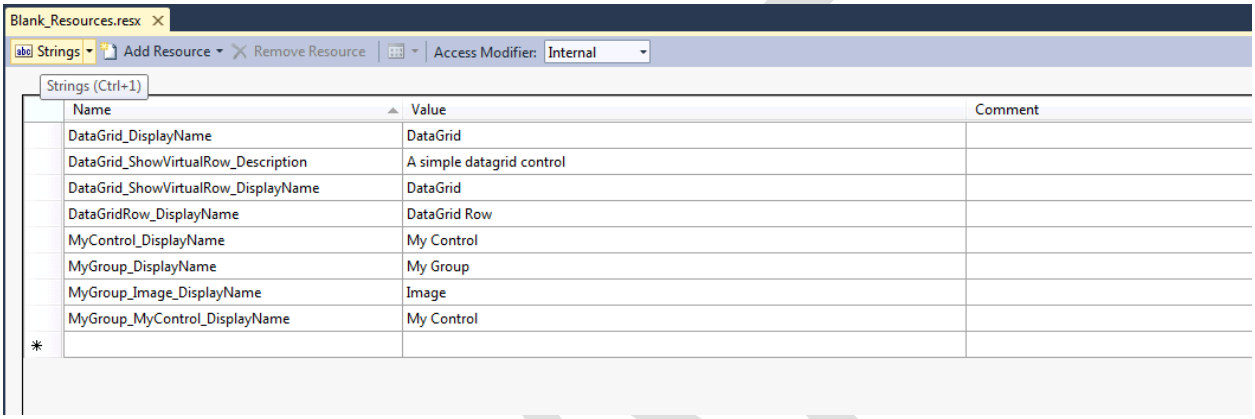
```
<Control Name="DataGrid"
    SupportedContentItemKind="Collection"
    CommandItemsSupport="CustomDisplay"
    AttachedLabelSupport="DisplayedByControl"
    ChildView="DataGridRow"
    CommandChildView=":CollectionButton">
    <Control.Attributes>
      <DisplayName Value="$(DataGrid_DisplayName)" />
    </Control.Attributes>

    <Control.Properties>
      <!-- Show Virtual Row property -->
      <ControlProperty
        Name="ShowVirtualRow"
        PropertyType=":Boolean"
        CategoryName="Appearance"
        EditorVisibility="PropertySheet">
        <ControlProperty.Attributes>
          <DisplayName Value="$(DataGrid_ShowVirtualRow_DisplayName)" />
          <Description Value="$(DataGrid_ShowVirtualRow_Description)" />
        </ControlProperty.Attributes>
        <ControlProperty.DefaultValueSource>
          <ScreenExpressionTree>
            <ConstantExpression ResultType=":Boolean" Value="True"/>
          </ScreenExpressionTree>
        </ControlProperty.DefaultValueSource>
      </ControlProperty>
    </Control.Properties>

    <!-- Override AttachedLabelPosition Property -->
    <Control.PropertyOverrides>
      <ControlPropertyOverride
        Property=":RootControl/Properties[AttachedLabelPosition]"
        EditorVisibility="NotDisplayed">
        <ControlPropertyOverride.DefaultValueSource>
          <ScreenExpressionTree>
            <ConstantExpression ResultType=":String" Value="None" />
          </ScreenExpressionTree>
        </ControlPropertyOverride.DefaultValueSource>
      </ControlPropertyOverride>
```

```
<!-- Support VerticalAlignment Property -->
<ControlPropertyOverride
  Property=":RootControl/Properties[VerticalAlignment]"
  EditorVisibility="PropertySheet">
  <ControlPropertyOverride.DefaultValueSource>
    <ScreenExpressionTree>
      <ConstantExpression ResultType=":String" Value="Stretch"/>
    </ScreenExpressionTree>
  </ControlPropertyOverride.DefaultValueSource>
</ControlPropertyOverride>

<!-- Support WidthSizingMode Property -->
<ControlPropertyOverride
  Property=":RootControl/Properties[WidthSizingMode]"
  EditorVisibility="NotDisplayed">
  <ControlPropertyOverride.DefaultValueSource>
    <ScreenExpressionTree>
      <ConstantExpression ResultType=":String" Value="Auto"/>
    </ScreenExpressionTree>
  </ControlPropertyOverride.DefaultValueSource>
</ControlPropertyOverride>

<!-- Support HeightSizingMode Property -->
<ControlPropertyOverride
  Property=":RootControl/Properties[HeightSizingMode]"
  EditorVisibility="PropertySheet">
  <ControlPropertyOverride.DefaultValueSource>
    <ScreenExpressionTree>
      <ConstantExpression ResultType=":String" Value="Auto"/>
    </ScreenExpressionTree>
  </ControlPropertyOverride.DefaultValueSource>
</ControlPropertyOverride>

<!-- Support Width Property -->
<ControlPropertyOverride
  Property=":RootControl/Properties[Width]"
  EditorVisibility="NotDisplayed">
  <ControlPropertyOverride.DefaultValueSource>
    <ScreenExpressionTree>
      <ConstantExpression ResultType=":Double" Value="NaN"/>
    </ScreenExpressionTree>
  </ControlPropertyOverride.DefaultValueSource>
</ControlPropertyOverride>

<!-- Support Height Property -->
<ControlPropertyOverride
  Property=":RootControl/Properties[Height]"
  EditorVisibility="NotDisplayed">
  <ControlPropertyOverride.DefaultValueSource>
    <ScreenExpressionTree>
      <ConstantExpression ResultType=":Double" Value="NaN"/>
    </ScreenExpressionTree>
  </ControlPropertyOverride.DefaultValueSource>
</ControlPropertyOverride>

<!-- Support MinWidth Property -->
<ControlPropertyOverride
```

```
        Property=":RootControl/Properties[MinWidth]"
        EditorVisibility="NotDisplayed">
        <ControlPropertyOverride.DefaultValueSource>
          <ScreenExpressionTree>
            <ConstantExpression ResultType=":Double" Value="175"/>
          </ScreenExpressionTree>
        </ControlPropertyOverride.DefaultValueSource>
      </ControlPropertyOverride>

      <!-- Support MinHeight Property -->
      <ControlPropertyOverride
        Property=":RootControl/Properties[MinHeight]"
        EditorVisibility="NotDisplayed">
        <ControlPropertyOverride.DefaultValueSource>
          <ScreenExpressionTree>
            <ConstantExpression ResultType=":Double" Value="125"/>
          </ScreenExpressionTree>
        </ControlPropertyOverride.DefaultValueSource>
      </ControlPropertyOverride>

    </Control.PropertyOverrides>

    <Control.ChildItemPropertySources>

      <!-- Set ContainerState for descendants to "Cell", so they can draw themselves
differently inside a datagrid -->
      <ControlPropertySource
        Property=":RootControl/Properties[ContainerState]">
        <ControlPropertySource.Source>
          <ScreenExpressionTree>
            <ConstantExpression ResultType=":String" Value="Cell" />
          </ScreenExpressionTree>
        </ControlPropertySource.Source>
      </ControlPropertySource>

    </Control.ChildItemPropertySources>

  </Control>

  <Control Name="DataGridRow"
    SupportedContentItemKind="Group"
    AttachedLabelSupport="DisplayedByControl"
    IsHidden="True">
    <Control.Attributes>
      <DisplayName Value="$(DataGridRow_DisplayName)" />
    </Control.Attributes>

    <Control.PropertyOverrides>

      <!-- The data grid row shouldn't show attached label position -->
      <ControlPropertyOverride
        Property=":RootControl/Properties[AttachedLabelPosition]"
        EditorVisibility="NotDisplayed"/>

      <!-- Support BrowseOnly (it affects the children, not the group control itself) -->
      <ControlPropertyOverride
        Property=":RootControl/Properties[BrowseOnly]"
        EditorVisibility="PropertySheet"/>
```

```
      <!-- Turn off alignment properties -->
      <ControlPropertyOverride
        Property=":RootControl/Properties[VerticalAlignment]"
        EditorVisibility="NotDisplayed"/>

    </Control.PropertyOverrides>

  </Control>
```

Since there are some resource names used in the metadata, update the Blank_resources.resx file in the common project with the following:



Add the following code to the BlankModule.cs class file with the BlankModule class:

```
public static class DataGrid
{
   private const string Name = "DataGrid";
   public const string GlobalName = GlobalPrefix + Name;
}
```

Now the bulk of the code is complete in the Client project. Let's create the visual of the datagrid. In the controls folder in the client project, add a DataGridVisual.xaml Silverlight User Control, and then add the following code. Please take note that instead of a usercontrol it is a resource dictionary.

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:c="clr-namespace:Blank.Controls"
    xmlns:pfx="clr-
namespace:Microsoft.LightSwitch.Presentation.Framework;assembly=Microsoft.LightSwitch.Cli
ent">

        <Style TargetType="c:DataGridVisual">
        <Setter Property="Template">
            <Setter.Value>
                <ControlTemplate TargetType="c:DataGridVisual">
                    <Grid>
```

```
                        <Grid.RowDefinitions>
                            <RowDefinition Height="Auto"/>
                            <RowDefinition Height="*"/>
                        </Grid.RowDefinitions>
                        <pfx:CommandGroupPresenter
                            Commands="{Binding CommandItems}"
                            HorizontalAlignment="Left"
                            VerticalAlignment="Center"
                            ContainerState="IntegratedCommandGroup"/>
                        <c:DataGridControl x:Name="DataGrid"
                            Grid.Row="1"
                            AutoGenerateColumns="False"
                            ItemsSource="{TemplateBinding CollectionView}"
                            RowContentItem="{Binding ChildItems[0]}"/>
                    </Grid>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>

</ResourceDictionary>
```

In this code file, there is a ControlTemplate that specifies the visual and sets up the data grid rows and columns. There are some bindings that allow command items to be included above the first row and then the datagrid control has the items placed within it by the items source property with the TemplateBinding set to the CollectionView. The row items themselves are set by the RowContentItem with a binding set to the ChildItems array.

The code-behind file needs the following code:

```
using System.ComponentModel;
using System.ComponentModel.Composition;
using System.Windows;
using System.Windows.Controls;

using Microsoft.LightSwitch.Presentation;
using Blank.Converters;

namespace Blank.Controls
{
    public class DataGridVisual :
        Control,
        ICollectionContentVisual,
        ICommitUIChanges,
        IContentVisual
    {
        #region Constructor

        public DataGridVisual()
        {
            this.DefaultStyleKey = typeof(DataGridVisual);
        }

        #endregion

        #region ICollectionContentVisual Members
```

```
        void ICollectionContentVisual.ShowItem(object value)
        {
            if (null != this.DataGrid && this.DataGrid.Columns.Count > 0)
            {
                this.DataGrid.ScrollIntoView(value, this.DataGrid.Columns[0]);
            }
        }

        #endregion

        #region ICommitUIChanges Members

        void ICommitUIChanges.CommitUIChanges()
        {
            if (null != this.DataGrid)
            {
                this.DataGrid.CommitEdit();
            }
        }

        #endregion

        #region IContentVisual Members

        object IContentVisual.Control
        {
            get { return this.DataGrid; }
        }

        void IContentVisual.Show()
        {
            if (null != this.DataGrid)
            {
                if (null != this.DataGrid.SelectedItem)
                {
                    if (null != this.DataGrid.CurrentColumn)
                    {
                        this.DataGrid.ScrollIntoView(this.DataGrid.SelectedItem,
this.DataGrid.CurrentColumn);
                    }
                    else if (this.DataGrid.Columns.Count > 0)
                    {
                        this.DataGrid.ScrollIntoView(this.DataGrid.SelectedItem,
this.DataGrid.Columns[0]);
                    }
                }
            }
        }

        #endregion

        #region Dependency Properties

        public static readonly DependencyProperty CollectionViewProperty =
            DependencyProperty.Register(
                DataGridVisual.CollectionViewPropertyName,
                typeof(ICollectionView),
```

```
                typeof(DataGridVisual),
                new PropertyMetadata(null,
DataGridVisual.OnCollectionViewPropertyChanged));

        public static readonly DependencyProperty ShowVirtualRowProperty =
            DependencyProperty.Register(
                DataGridVisual.ShowVirtualRowPropertyName,
                typeof(bool),
                typeof(DataGridVisual),
                new PropertyMetadata(false,
DataGridVisual.OnShowVirtualRowPropertyChanged));

        #endregion

        #region Overrides

        public override void OnApplyTemplate()
        {
            base.OnApplyTemplate();

            this.DataGrid =
base.GetTemplateChild(DataGridVisual.DataGridTemplatePartName) as
System.Windows.Controls.DataGrid;
        }

        #endregion

        #region Methods

        private static void OnCollectionViewPropertyChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
        {
            ((DataGridVisual)d).SetPlaceholderPosition();
        }

        private static void OnShowVirtualRowPropertyChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
        {
            ((DataGridVisual)d).SetPlaceholderPosition();
        }

        private void SetPlaceholderPosition()
        {
            IEditableCollectionView editableCollectionView = this.CollectionView as
IEditableCollectionView;
            if (editableCollectionView != null)
            {
                if (this.ShowVirtualRow)
                {
                    editableCollectionView.NewItemPlaceholderPosition =
NewItemPlaceholderPosition.AtEnd;
                }
                else
                {
                    editableCollectionView.NewItemPlaceholderPosition =
NewItemPlaceholderPosition.None;
                }
            }
```

```
        }

        #endregion

        #region Properties

        public ICollectionView CollectionView
        {
            get { return
(ICollectionView)base.GetValue(DataGridVisual.CollectionViewProperty); }
            set { base.SetValue(DataGridVisual.CollectionViewProperty, value); }
        }

        private System.Windows.Controls.DataGrid DataGrid { get; set; }

        public bool ShowVirtualRow
        {
            get { return (bool)base.GetValue(DataGridVisual.ShowVirtualRowProperty); }
            set { base.SetValue(DataGridVisual.ShowVirtualRowProperty, value); }
        }

        #endregion

        #region Constants

        private const string CollectionViewPropertyName = "CollectionView";
        private const string ShowVirtualRowPropertyName = "ShowVirtualRow";

        private const string DataGridTemplatePartName = "DataGrid";

        #endregion
    }

    [Export(typeof(IControlFactory))]
    [ControlFactory(BlankModule.DataGrid.GlobalName)]
    internal class DataGridControlFactory :
        BaseControlFactory
    {
        public DataGridControlFactory()
            : base("DataGridTemplate")
        {
        }
    }
}
```

This code file has the IContentCollectionVisual and the CollectionView, which provides the capability to enter data directly into the cell of the datagrid as well as to tab across cells. When a user clicks on the cell, the cell becomes active and allows data to be entered based on what the column storage type is set at. If a storage type on a column is set to Integer and on cell selection in the data grid the user tries to enter text then when the user tabs/or selects out of the cell it will be set to its default value.

The next class file that needs to be created is the DataGridControl.cs class file which is the datagrid implementation. First you need to add some references to the client project.

The following references need to be added to the Client project:

C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\LightSwitch\1.0\Client\Microsoft.LightSwitch.ExportProvider.dll

C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\LightSwitch\1.0\Client\Microsoft.LightSwitch.SDKProxy.dll

C:\Program Files (x86)\Microsoft SDKs\Silverlight\v4.0\Libraries\Client\System.Windows.Controls.Data.dll

C:\Program Files (x86)\Microsoft SDKs\Silverlight\v4.0\Libraries\Client\System.Windows.Data.dll

Now, create the DataGridControl.cs class file and add the following code:

```
using System;
using System.Collections.Generic;
using System.Collections.Specialized;
using System.ComponentModel;
using System.Linq;
using System.Windows;
using System.Windows.Controls;

using Microsoft.LightSwitch.Presentation;
using Microsoft.LightSwitch.Sdk.Proxy;
using Microsoft.LightSwitch.Utilities;

using Microsoft.VisualStudio.ExtensibilityHosting;

namespace Blank.Controls
{
    public class DataGridControl :
        System.Windows.Controls.DataGrid
    {
        #region Dependency Properties

        public static readonly DependencyProperty RowContentItemProperty =
            DependencyProperty.Register(
                DataGridControl.RowContentItemPropertyName,
                typeof(IContentItem),
                typeof(DataGridControl),
                new PropertyMetadata(DataGridControl.OnRowContentItemChanged));

        #endregion

        #region Event Handlers

        private void ColumnContentItem_PropertyChanged(object sender,
PropertyChangedEventArgs e)
        {
            IContentItem changedItem = (IContentItem)sender;

            if (e.PropertyName.Equals("IsVisible", StringComparison.Ordinal))
            {
                if (changedItem.IsVisible)
```

```
                {
                    this.AddColumn(changedItem);
                }
                else
                {
                    this.RemoveColumn(changedItem);
                }
            }
            else
            {
                if (changedItem.IsVisible)
                {
                    this.RegenerateColumn(changedItem);
                }
            }
        }

        private void ColumnContentItems_CollectionChanged(object sender,
NotifyCollectionChangedEventArgs e)
        {
            switch (e.Action)
            {
                case NotifyCollectionChangedAction.Add:
                    IContentItem addedItem = (IContentItem)e.NewItems[0];
                    addedItem.PropertyChanged += this.ColumnContentItem_PropertyChanged;
                    this.AddColumn(addedItem);
                    break;
                case NotifyCollectionChangedAction.Remove:
                    IContentItem removedItem = (IContentItem)e.OldItems[0];
                    this.RemoveColumn(removedItem);
                    removedItem.PropertyChanged -=
this.ColumnContentItem_PropertyChanged;
                    break;
            }
        }

        #endregion

        #region Methods

        private void AddColumn(IContentItem contentItem)
        {
            if (null == this.GetColumnFromContentItem(contentItem))
            {
                IContentItem previousItem = this.RowContentItem.ChildItems.TakeWhile(item
=> item != contentItem).LastOrDefault(item => item.IsVisible);
                ContentItemColumn newColumn = new ContentItemColumn(contentItem);
                if (null == previousItem)
                {
                    this.Columns.Insert(0, newColumn);
                }
                else
                {
                    ContentItemColumn previousColumn =
this.GetColumnFromContentItem(previousItem);
                    int previousIndex = this.Columns.IndexOf(previousColumn);
                    this.Columns.Insert(previousIndex + 1, newColumn);
                }
```

```
            }
        }

        private ContentItemColumn GetColumnFromContentItem(IContentItem contentItem)
        {
            return this.Columns.OfType<ContentItemColumn>().SingleOrDefault(col =>
col.ContentItem == contentItem);
        }

        private static void OnRowContentItemChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
        {
            ((DataGridControl)d).OnRowContentItemChanged((IContentItem)e.OldValue,
(IContentItem)e.NewValue);
        }

        private void OnRowContentItemChanged(IContentItem oldContentItem, IContentItem
newContentItem)
        {
            if (null != oldContentItem)
            {
                this.ColumnContentItems = null;
            }
            this.ResetColumns(oldContentItem, newContentItem);
            if (null != newContentItem)
            {
                this.ColumnContentItems = newContentItem.ChildItems as
INotifyCollectionChanged;
            }
        }

        private void RegenerateColumn(IContentItem contentItem)
        {
            ContentItemColumn column = this.GetColumnFromContentItem(contentItem);
            int itemIndex = -1;
            if (null != column)
            {
                itemIndex = this.Columns.IndexOf(column);
                this.Columns.RemoveAt(itemIndex);
            }
            if (itemIndex >= 0)
            {
                this.Columns.Insert(itemIndex, new ContentItemColumn(contentItem));
            }
            else
            {
                this.AddColumn(contentItem);
            }
        }

        private void ResetColumns(IContentItem oldContentItem, IContentItem
newContentItem)
        {
            if (null != oldContentItem)
            {
                foreach (IContentItem contentItem in oldContentItem.ChildItems)
                {
```

```
                    contentItem.PropertyChanged -=
this.ColumnContentItem_PropertyChanged;
                }
            }

            base.Columns.Clear();

            if (null != newContentItem)
            {
                foreach (IContentItem contentItem in newContentItem.ChildItems)
                {
                    contentItem.PropertyChanged +=
this.ColumnContentItem_PropertyChanged;

                    if (contentItem.IsVisible)
                    {
                        this.Columns.Add(new ContentItemColumn(contentItem));
                    }
                }
            }
        }

        private void RemoveColumn(IContentItem contentItem)
        {
            ContentItemColumn column = this.GetColumnFromContentItem(contentItem);
            if (null != column)
            {
                this.Columns.Remove(column);
            }
        }

        #endregion

        #region Properties

        private INotifyCollectionChanged ColumnContentItems
        {
            get { return this.columnContentItems; }
            set
            {
                if (value != this.columnContentItems)
                {
                    if (null != this.columnContentItems)
                    {
                        this.columnContentItems.CollectionChanged -=
this.ColumnContentItems_CollectionChanged;
                    }
                    this.columnContentItems = value;
                    if (null != this.columnContentItems)
                    {
                        this.columnContentItems.CollectionChanged +=
this.ColumnContentItems_CollectionChanged;
                    }
                }
            }
        }

        public IContentItem RowContentItem
```

```
        {
            get { return
(IContentItem)base.GetValue(DataGridControl.RowContentItemProperty); }
            set { base.SetValue(DataGridControl.RowContentItemProperty, value); }
        }

        #endregion

        #region Fields

        private INotifyCollectionChanged columnContentItems;

        internal static DataTemplate NewItemPlaceholderTemplate =
BaseControlFactory.GetDataTemplate(DataGridControl.NewItemPlaceholderTemplateName);

        #endregion

        #region Constants

        private const string RowContentItemPropertyName = "RowContentItem";

        private const string NewItemPlaceholderTemplateName =
"NewItemPlaceholderTemplate";

        #endregion
    }
}
```

This code provides properties for the datagrid columns are changed, when the entity is changed and in turn changes the collection. Other functions are to add rows, regenerate columns, reset columns, clear columns, and remove columns. The important part to note in this code is that there is a collection which is a list that is retrieved from the IContentItem interface, as this is the only way to get access to the data in the entity as well as the entity structure.

In order to know what storage type is used on a column, you need to access the LightSwitch model from the SDK proxy. This reference will be needed when you create the next class file. Add the following class file called ContentItemColumn.cs to the Client project in the Controls folder. Add the following code:

```
using System;
using System.ComponentModel;
using System.Linq;
using System.Windows;
using System.Windows.Controls;

using Microsoft.LightSwitch.Presentation;
using Microsoft.LightSwitch.Presentation.Framework;
using Microsoft.LightSwitch.Model;
using Microsoft.LightSwitch.Sdk.Proxy;

using Microsoft.VisualStudio.ExtensibilityHosting;

namespace Blank.Controls
{
    public class ContentItemColumn :
        DataGridColumn
```

```
    {
        #region Constructor

        public ContentItemColumn(IContentItem contentItem)
        {
            this.ContentItem = contentItem;

            IPresentationViewDefinition view = contentItem.View;
            IControlFactory factory =
ContentItemColumn.ServiceProxy.ContentItemService.GetControlFactory(view);
            this.simpleCellTemplate = factory.GetDisplayModeDataTemplate(contentItem);
            this.Header = contentItem.DisplayName;

            IMemberDefinition modelItem = contentItem.LastMemberModelItem;
            if (null != modelItem)
            {
                IEntityPropertyDefinition property = modelItem as
IEntityPropertyDefinition;
                if (ContentItemColumn.IsSortable(property))
                {
                    this.CanUserSort = true;

                    if (String.IsNullOrEmpty(contentItem.BindingPath))
                    {
                        this.SortMemberPath = modelItem.Name;
                    }
                    else
                    {
                        this.SortMemberPath = contentItem.BindingPath;
                    }
                }
            }
        }

        #endregion

        #region Overrides

        protected override FrameworkElement GenerateEditingElement(DataGridCell cell,
object dataItem)
        {
            if (this.ContentItem.IsDisposed)
                return null;

            ContentItemPresenter presenter = new ContentItemPresenter();
            presenter.CreateContentItemHandler = this.CreateHandler(dataItem);

            return presenter;
        }

        protected override FrameworkElement GenerateElement(DataGridCell cell, object
dataItem)
        {
            if (this.ContentItem.IsDisposed)
                return null;

            object bindingRoot = null;
            if (dataItem != CollectionPropertyView.NewItemPlaceholder)
```

```
            {
                bindingRoot = dataItem;
            }

            DataTemplate cellTemplate = this.simpleCellTemplate;
            if (null == bindingRoot)
            {
                cellTemplate = DataGridControl.NewItemPlaceholderTemplate;
            }

            ContentItemPresenter presenter = new ContentItemPresenter(cellTemplate, null
!= this.simpleCellTemplate);
            presenter.CreateContentItemHandler = this.CreateHandler(bindingRoot);

            return presenter;
        }

        protected override object PrepareCellForEdit(FrameworkElement editingElement,
RoutedEventArgs editingEventArgs)
        {
            return editingElement;
        }

        #endregion

        #region Methods

        private CreateContentItemHandler CreateHandler(object bindingRoot)
        {
            return () => this.CopyContentItem(bindingRoot);
        }

        private IContentItem CopyContentItem(object bindingRoot)
        {
            return this.ContentItem.CreateTemplatedTreeCopy(bindingRoot);
        }

        private static bool IsSortable(IPropertyDefinition property)
        {
            if (!ContentItemColumn.IsSortable(property.PropertyType))
            {
                return false;
            }
            if
(property.Attributes.OfType<IHiddenAttribute>().FirstOrDefault<IHiddenAttribute>() !=
null)
            {
                return false;
            }
            if
(property.Attributes.OfType<INotSortableAttribute>().FirstOrDefault<INotSortableAttribute
>() != null)
            {
                return false;
            }
            return
!property.Attributes.OfType<IComputedAttribute>().Any<IComputedAttribute>();
        }
```

```
        private static bool IsSortable(IDataType dataType)
        {
            INullableType nullableType = dataType as INullableType;
            if (null != nullableType)
            {
                dataType = nullableType.UnderlyingType;
            }
            while (dataType is ISemanticType)
            {
                dataType = ((ISemanticType)dataType).UnderlyingType;
            }
            return !String.Equals(dataType.Id, ContentItemColumn.BinaryTypeId,
StringComparison.Ordinal);
        }

        #endregion

        #region Properties

        public IContentItem ContentItem { get; private set; }

        #endregion

        #region Fields

        private DataTemplate simpleCellTemplate;

        private static IServiceProxy ServiceProxy =
VsExportProviderService.GetServiceFromCache<IServiceProxy>();

        #endregion

        #region Constants

        private const string BinaryTypeId = "Microsoft.LightSwitch:Binary";

        #endregion
    }
}
```

Next you need to update the DataTemplates.xaml file with the following code:

```
<DataTemplate x:Key="DataGridTemplate">
        <c:DataGridVisual ShowVirtualRow="{Binding
Properties[Blank:DataGrid/ShowVirtualRow]}">
            <c:DataGridVisual.CollectionView>
                <Binding Path="Details">
                    <Binding.Converter>
                        <cvt:CollectionPropertyViewConverter/>
                    </Binding.Converter>
                </Binding>
            </c:DataGridVisual.CollectionView>
        </c:DataGridVisual>
    </DataTemplate>
```

```
    <DataTemplate x:Key="NewItemPlaceholderTemplate">
        <TextBlock/>
    </DataTemplate>
```

The DataGridTemplate binds to a virtual row of the CollectionView that has the binding path to the details of the IContentItem. The other data template is for when you have a data grid with no data entered but you have selected to enter in data.

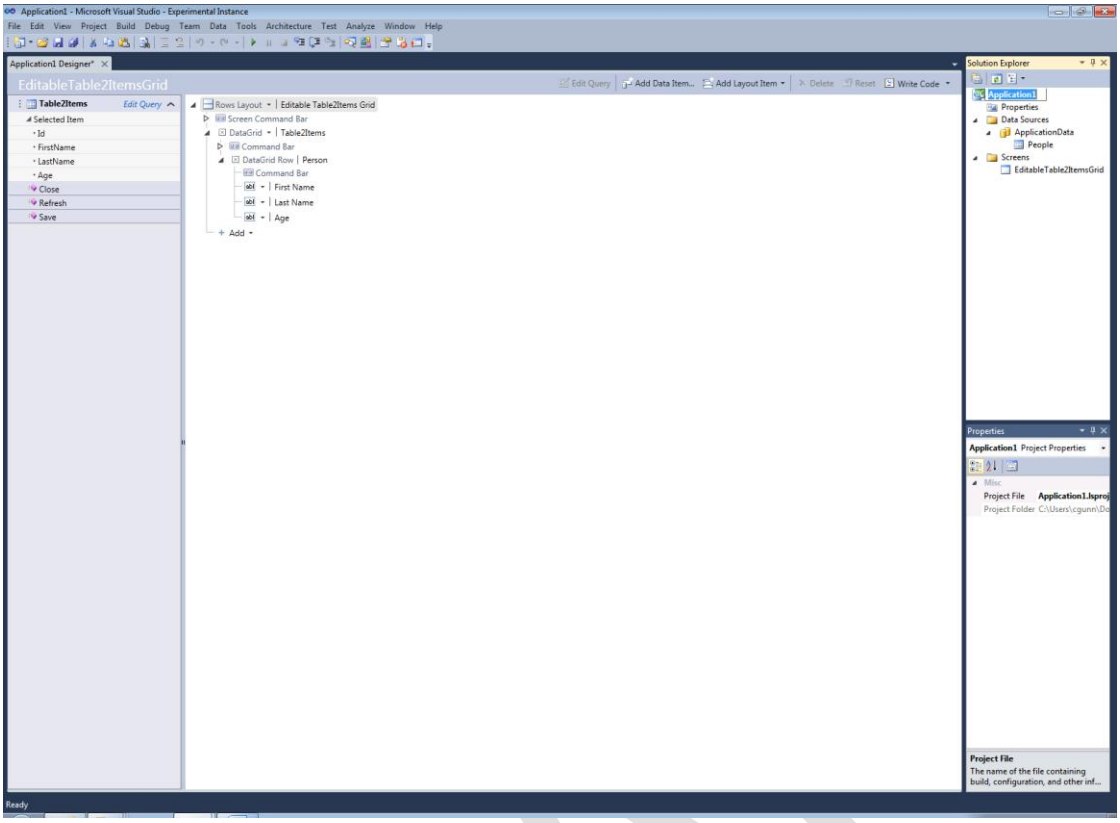
The final step that you need to do in order for the data grid control to work correctly is to ensure that the SDK proxy assembly is available to the Client project when the project assemblies are packaged into a LightSwitch Extension. To do this you need to go to the Blank.LsPkg project and unload the project. Next, edit the Blank.LsPkg.csproj.

Within the first item group of the project file you need to insert the following code:

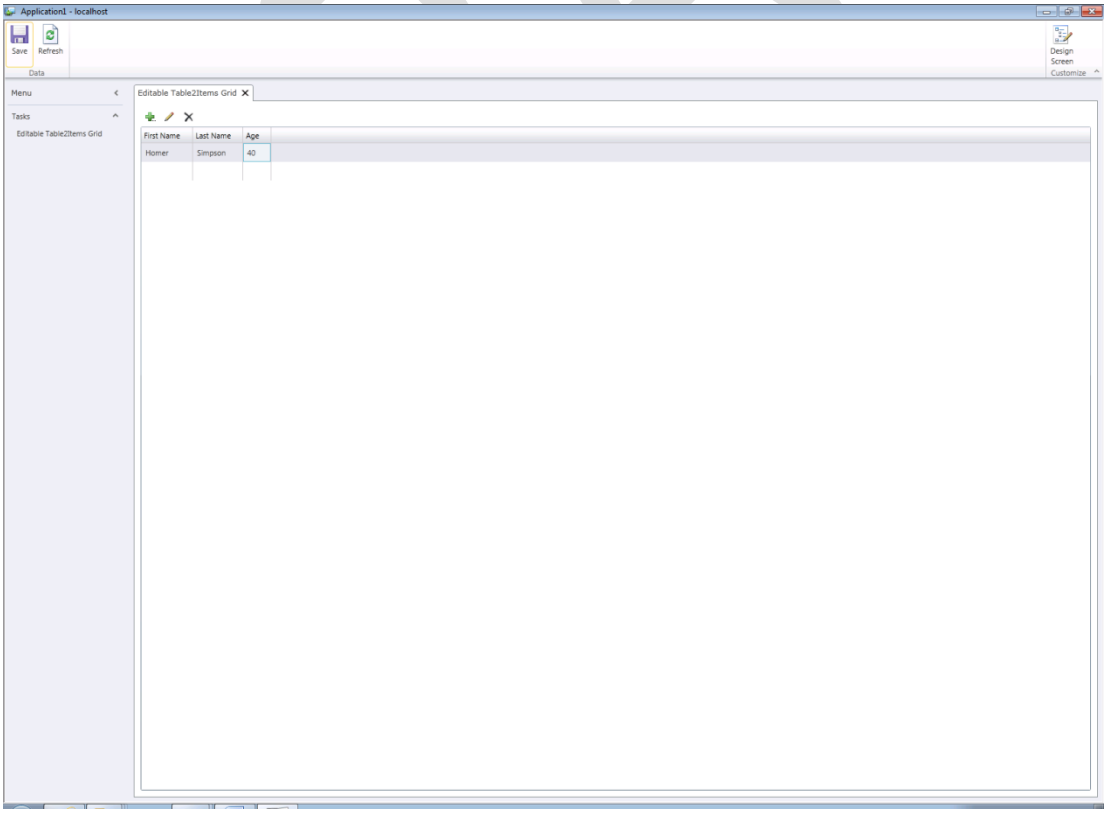
```
<ItemGroup>
    <ClientGeneratedReference Include="C:\Program Files (x86)\Microsoft Visual Studio
10.0\Common7\IDE\LightSwitch\1.0\Client\Microsoft.LightSwitch.SDKProxy.dll" />
    <ClientGeneratedReference Include="..\Blank.Client\$(ProjectPath)\Blank.Client.dll"
/>
    <ClientGeneratedReference Include="..\Blank.Common\$(ProjectPath)\Blank.Common.dll"
/>
    <ClientDebugOnlyReference
Include="..\Blank.ClientDesign\$(ProjectPath)\Blank.Client.Design.dll" />

    <IDEReference Include="..\Blank.Common\$(ProjectPath)\Blank.Common.dll" />
    <IDEReference Include="..\Blank.Design\$(ProjectPath)\Blank.Design.dll" />

    <ServerGeneratedReference Include="..\Blank.Common\$(ProjectPath)\Blank.Common.dll"
/>
    <ServerGeneratedReference Include="..\Blank.Server\$(ProjectPath)\Blank.Server.dll"
/>
  </ItemGroup>
```

After inserting the first line, save the file and reload the project.

Now debug the extension in the LightSwitch test application. Experiment with adding different storage types in an entity. Also try the Editable Grid screen template and select the datagrid control, which will look like below:



And the application running with some data added will look like the following:



Now that you have gotten through the three different kinds of controls, you may have notice that there were no icons in the controls. So, how do we put icons where the grey X is?

The two projects that you need to focus on are the Design and ClientDesign projects. The Design project allows us to see icons of the controls in the screen designer at design-time. The ClientDesign project is only used in run-time debugging where the run-time design screen button is selected.

In the Design project you need to add an icon to the Images folder., The size of the icon needs to be 16x16 32-bit and the build action needs to be set to Resource.

You can add as many icons as you wish to set to your controls, whether they be a group, datagrid, datagrid row control.

In the resources folder of the Design project you need to reference the icons. The Images.xaml file is the file to take note of, especially when it comes to the key reference. In that case, the xaml file it has an entry of Blank16. So, let's use this icon in our Datagrid control. The process for other controls is exactly the same. The Blank16 resource key then gets used in the Common project in the BlankPresentation.lsml file.

Add the following code in the Datagrid control, just under the SupportedContentItemKind in the BlankPresentation.lsml file:

```
DesignerImageResource="Images::Blank"
```

Debug the extension in LightSwitch and inspect the screen you created earlier and you will see the icon now present beside the DataGrid control.

That completes the recipe for creating control extensions in LightSwitch.

Screen Template Extension

This control recipe enables you to create a screen template that allows a developer to create a predefined screen that can use the controls and layout from LightSwitch.

The ingredients that you are going to need to create a screen template are the following:

1. Visual Studio 2010 Professional or higher
2. Visual Studio 2010 Service Pack 1
3. Visual Studio 2010 SDK
4. Visual Studio LightSwitch
5. LightSwitch Blank Extension Solution

Once you have all of the ingredients installed on your machine you are ready to create a screen template extension.

The first step is to get set up with the right environment in order to add your specific code that you want your screen template to display and use. This is done by opening the Blank Extension solution sample, which is called BlankExtension.sln. (You can rename the solution if you would like; this example will keep using the name of Blank.

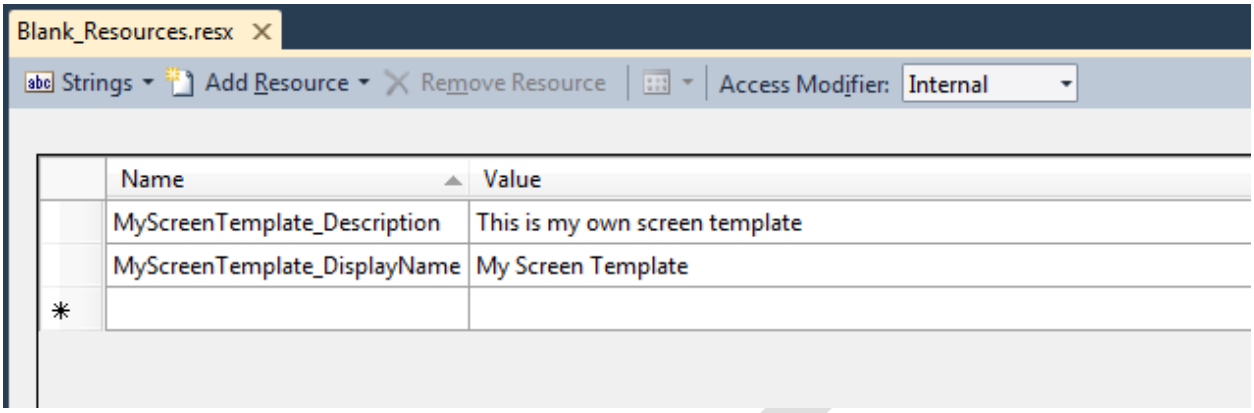
The first step, as when creating a control extension, is to focus on the metadata in the Common project.

In the Common project there is not a lot of metadata that needs to be created. There is no use of a presentation lsml file, only the module file. In the module file (BlankModule.lsml) the current module name is 'Blank'.

```
<Module Name="Blank" />
```

You could rename this to the name of your extension. This is the extension internal namespace used by LightSwitch for the extension you are creating.

The next file you need to define contains the name and description of the screen template, which is stored in a resource file. You will use the Blank_Resource.resx file, and add the following name/value pairs:



Creating a screen template only requires one other project. This is the Design project, as all the interaction is done in the LightSwitch IDE.

For code readability and setting out your project it is recommended that you create the necessary folders to support the code and artifacts that are in a screen template extension.

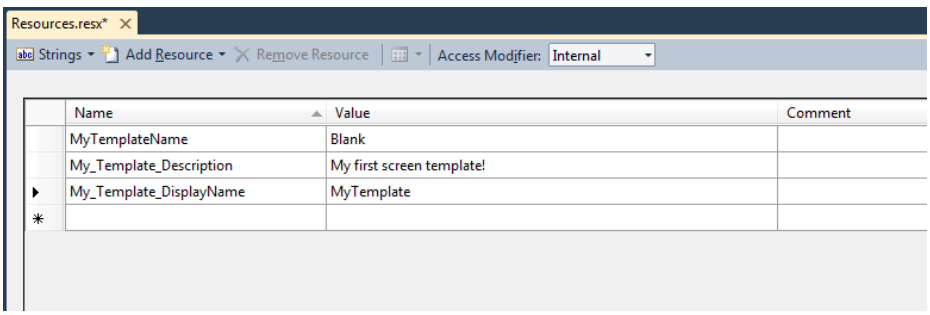
In the Blank.Design file add the following references:

1. C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\PrivateAssemblies\Microsoft.LightSwitch.Design.Designer.dll
2. C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\LightSwitch\1.0\Server\Microsoft.LightSwitch.Model.Xaml.dll

The three folders that you need to create are the following:

1. Images folder – This contains the images that are presented in the Add New Screen dialog when creating a screen. There are two png image files that need to be created:
    a. The first image is the main png image that is on the left side of the dialog. The png image size is 20x16 (32 bit); ensure that the image build type is set to Resource.
    b. The second image is the preview image that is in the center of the dialog. The png image size is 252x173 (32 bit); ensure that the image build type is set to Resource.
2. Resources folder – This contains the design project resources file, Resources.resx.

    The file needs to have the following name/value pairs created:



3. ScreenTemplates folder – There are three files that are required here.
    a. The first file is the Common.cs class file. This file sets up the BaseScreenTemplate class as shown in the code below:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.LightSwitch.Designers.ScreenTemplates.Model;
using Microsoft.LightSwitch.Model.Storage;

namespace Blank.ScreenTemplates
{
    internal class BaseScreenTemplate
    {

        protected void
AddContentItemsForQueryParameters(IScreenTemplateHost host, ContentItem
parent)
        {
            if (host.PrimaryDataSourceParameterProperties != null)
            {
                foreach (ScreenPropertyBase parameter in
host.PrimaryDataSourceParameterProperties)
                {
                    host.AddContentItem(parent, parameter.Name, parameter);
                }
            }
        }
    }
}
```

    b. Create a class file called TemplateNames.cs. This file contains the name to ids for controls used within LightSwitch that you define.

```
namespace Blank.ScreenTemplates
{
    internal static class TemplateNameIds
    {
        public const string MyScreenTemplateId = "MyScreenTemplate";
    }
}
```

    c. Create a class file called MyScreenTemplate.cs. This file contains the implementation code of your screen template extension.

```
using System;
using System.ComponentModel.Composition;
using Microsoft.LightSwitch.Designers.ScreenTemplates.Model;
using Microsoft.LightSwitch.Model;
using Microsoft.LightSwitch.Model.Storage;

namespace Blank.ScreenTemplates
{
    internal class MyScreenTemplate : BaseScreenTemplate, IScreenTemplate
```

```
    {
        #region IScreenTemplate members

        public void Generate(IScreenTemplateHost host)
        {
            // This is where specific code for the screen template goes.
        }

        public string TemplateName
        {
            get { return Resources.Resources.MyTemplateName; }
        }

        public string DisplayName
        {
            get { return Resources.Resources.My_Template_DisplayName; }
        }

        public string Description
        {
            get { return Resources.Resources.My_Template_Description; }
        }

        public Uri SmallIcon
        {
            get { return new
Uri("pack://application:,,/Blank.Design;component/Images/Blank.png"); }
        }

        public Uri PreviewImage
        {
            get { return new
Uri("pack://application:,,/Blank.Design;component/Images/BlankPreview.png")
; }
        }

        public RootDataSourceType RootDataSource
        {
            get { return RootDataSourceType.Collection; }
        }

        public bool SupportsChildCollections
        {
            get { return false; }
        }

        public string ScreenNameFormat
        {
            get { return "{0}BlankScreen"; }
        }

        #endregion
    }

    [Export(typeof(IScreenTemplateFactory))]
    [Template(TemplateNameIds.MyScreenTemplateId)]
    internal class MyScreenTemplateFactory : IScreenTemplateFactory
    {
```

```
        public IScreenTemplate CreateScreenTemplate()
        {
            return new MyScreenTemplate();
        }
    }
}
```

The ScreenNameFormat property sets the name that will be placed in the screen name by default when MyScreenTemplate is selected. You will see that there is a comment indicating in the Generate() function of where additional code can go to manipulate the screen template based on given requirements.

At this point you can build the screen template and you will get a blank screen template; no controls or layout are explicitly defined. You will have to do this yourself, which you will learn in this recipe next.

Let's now look at adding a list/details type screen template to your existing screen template so that you know what is possible.

In the TemplateNames.cs class file, you need to specify some additional control ids. Add the following ids:

```
public const string ControlColumnsLayoutId = "Microsoft.LightSwitch:ColumnsLayout";
public const string ControlListId = "Microsoft.LightSwitch:List";
public const string ControlRowsLayoutId = "Microsoft.LightSwitch:RowsLayout";
public const string ControlTabsId = "Microsoft.LightSwitch:Tabs";
public const string ControlDataGridId = "Microsoft.LightSwitch:DataGrid";
```

This now gives you additional controls that you can use in your Generate() function.

To allow a better definition of the left-hand side of the screen the following constants can be defined:

```
// IScreenTemplate members

        private const string LeftColumnGroupName = "ListColumn";
        private const string RightColumnGroupName = "DetailsColumn";
```

The Generate() function is used to place any control on screen based on what storage type is selected in the entity that the screen template will reference.

```
public void Generate(IScreenTemplateHost host)
        {
            ContentItem mainGroup = host.ScreenLayoutContentItem;
            mainGroup.View = TemplateNameIds.ControlColumnsLayoutId;

            if (host.PrimaryDataSourceProperty != null)
            {
                IDataType primaryDataType =
                    host.FindGlobalModelItem<ISequenceType>(

((ScreenCollectionProperty)host.PrimaryDataSourceProperty).PropertyType
```

```
                ).ElementType;

            string primaryListContentItemName = primaryDataType.Name + "List";
            string primaryDetailsContentItemName = primaryDataType.Name + "Details";
            string childrenDetailsTabName = "ChildCollectionTabs";

            ContentItem listContentItem;

            // LEFT COLUMN - Rows layout with the query parameters and primary
collection inside
            ContentItem leftColumn = host.AddContentItem(mainGroup,
LeftColumnGroupName, ContentItemKind.Group);

            base.AddContentItemsForQueryParameters(host, leftColumn);

            listContentItem = host.AddContentItem(leftColumn,
primaryListContentItemName, host.PrimaryDataSourceProperty);
            listContentItem.View = TemplateNameIds.ControlListId;
            host.ExpandContentItem(listContentItem);

            // RIGHT COLUMN - Rows layout with details and child navigation
collections
            ContentItem rightColumn = host.AddContentItem(mainGroup,
RightColumnGroupName, ContentItemKind.Group);

            if (host.ShowPrimaryDataSourceDetails)
            {
                ChainExpression chain =
host.CreateChainExpression(host.CreateMemberExpression(host.PrimaryDataSourceProperty.Id)
);
                host.AppendMemberExpression(chain,
host.PrimaryDataSourceSelectedItemProperty.Name);

                ContentItem details = host.AddContentItem(
                    rightColumn,
                    primaryDetailsContentItemName,
                    ContentItemKind.Details,
                    chain,
                    primaryDataType.Id);
                host.SetDisplayName(details, null);
                details.View = TemplateNameIds.ControlRowsLayoutId;
                host.ExpandContentItem(details);

                // Resolving the content items
                host.ResolveContentItems();
            }

            ContentItem groupForChildNavigationProperties = rightColumn;
            if (host.ChildCollectionProperties != null)
            {
                if (host.ChildCollectionProperties.Count > 1)
                {
                    // If there are multiple child collection properties, put them in
tabs
                    ContentItem tab = host.AddContentItem(rightColumn,
childrenDetailsTabName, ContentItemKind.Group);
                    tab.View = TemplateNameIds.ControlTabsId;
                    groupForChildNavigationProperties = tab;
```

```
                    }

                    foreach (ScreenPropertyBase navigationProperty in
host.ChildCollectionProperties)
                    {
                        ContentItem details =
host.AddContentItem(groupForChildNavigationProperties, navigationProperty.Name,
navigationProperty);
                        host.SetContentItemView(details,
TemplateNameIds.ControlDataGridId);
                        host.ExpandContentItem(details);
                    }
                }
            }
        }
```

The main parts of this code to take note of are creating the group with the main group and then defining IContentItem to provide access to the storage types on the selected entity.

The content items are then connected by the AddContentItem and the SupportedItemKind defines what kind of control is being used. In the case of the code above there are a couple of groups and details controls.

The code so far makes use of the internal controls based on the storage types used in the entity. To add your own control based on a storage type or explicit name, the following code can be added just after the following code:

```
// Resolving the content items
                    host.ResolveContentItems();
```

First is to understand what storage types are used in the selected entity and build a list, illustrated in the following code:

```
// Getting the Content Item Resolved Info

host.GetContentItemResolvedInfo((ContentItem)((System.Collections.IList)details.ChildCont
entItems)[0]);
```

Now you need to make a choice on how you assign the control to a given storage type. You can explicitly set it as in the example code below. This is looking for the type of storage type, such as String, as well as if it is the storage type item in the entity, which is based on position. The following code shows setting to a TextBox.

```
// Setting the first item in the list to the TextBox Control - which is a String datatype
and has been assigned the builtin TextBox.

host.SetContentItemView((ContentItem)((System.Collections.IList)details.ChildContentItems
)[0], TemplateNameIds.ControlTextBoxId);
```

Another option could be based on the name of the storage type set the appropriate control, such as the following code:

```
foreach(IContentItemDefinition def in details.ChildContentItems)
                    {
                        if (def.Name.StartsWith("Address"))
                        {
                            host.GetContentItemResolvedInfo((ContentItem)def);
                            host.SetContentItemView((ContentItem)def,
TemplateNameIds.ControlTextBoxId);
                        }
                    }
```

This code is looking for the Address on the storage type name and setting the TextBox control.

Now that you have a couple of choices of how to specify the screen template extension, build it and experiment with it inside your LightSwitch test application.

Business Type Extension

The business type extension recipe enables you to create a business type that allows a developer to have defined a custom storage type on the entity, include specifying attributes (properties), validation logic and controls.

The ingredients that you are going to need to create a business type are the following:

1. Visual Studio 2010 Professional or higher
2. Visual Studio 2010 Service Pack 1
3. Visual Studio 2010 SDK
4. Visual Studio LightSwitch
5. LightSwitch Blank Extension Solution

Once you have all of the ingredients installed on your machine you are ready to create a business type extension.

The first step is to get you set up with the right environment. This is done by opening the Blank Extension solution sample, which is called BlankExtension.sln.

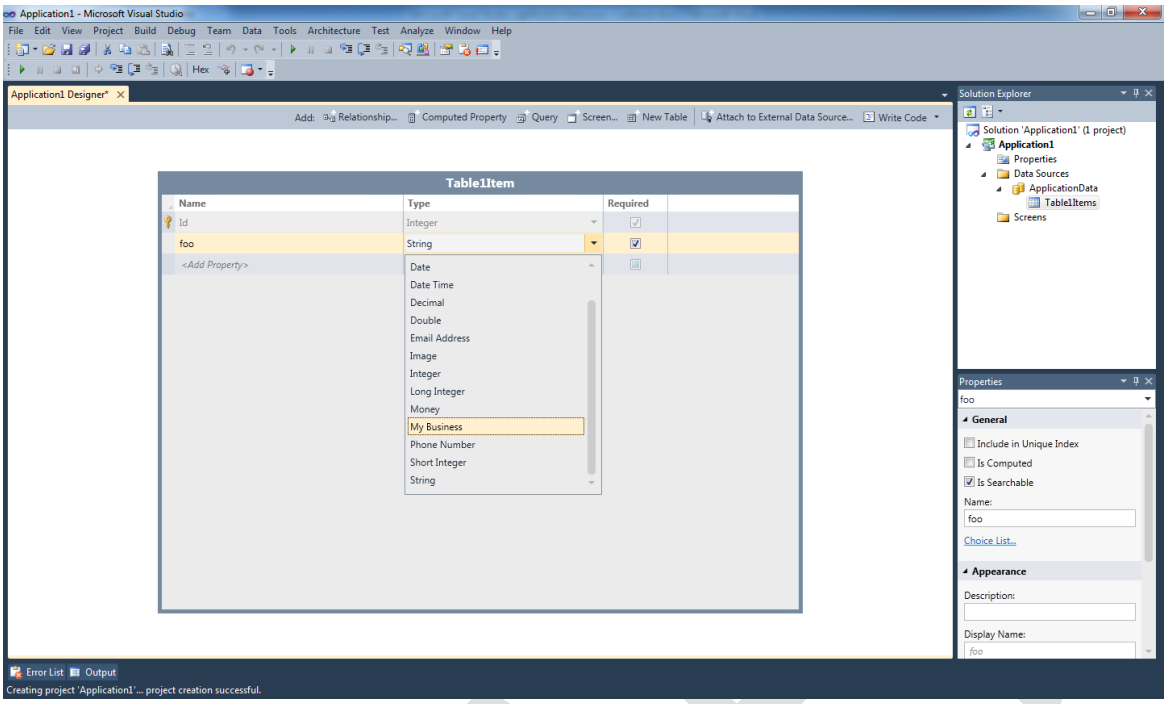
The first step in creating a business type extension is to focus on the metadata in the Common project.

The file that you will work with is the Presentation.lsml file in the metadata folder. This file will contain the details of what your business type is of a particular storage type, as shown in the code below:

```
<?xml version="1.0" encoding="utf-8" ?>
<ModelFragment
  xmlns="http://schemas.microsoft.com/LightSwitch/2010/xaml/model"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <!-- MyBusiness Semantic Type -->
  <SemanticType Name="MyBusiness" UnderlyingType=":String"/>

</ModelFragment>
```

In the code above, there is a relationship that is being established. In that, the MyBusiness type has an underlying storage type of String.

If you build and install the extension as is, you will be able to see that the business type appears in the drop-down of the storage type in an entity that you have created, as shown in the image below:



Right now the business type doesn't really do anything usefulso let's now put in some validation.

In the Presentation.lsml shows what the validator is, shown in the code below:

```
<!-- MyBusiness Semantic Type -->
  <SemanticType Name="MyBusiness" UnderlyingType=":String">
    <SemanticType.Attributes>
      <Attribute Class="@MyBusinessValidation" />
    </SemanticType.Attributes>
  </SemanticType>
```

Here an attribute class is added called MyValidation. Y now need to tell that this class supports validation.

Below the semantic type end tag in the Presentation.lsml file, add the following code:

```
<AttributeClass Name="MyBusinessValidation">
    <AttributeClass.Attributes>
      <Validator />
      <SupportedType Type="MyBusiness?" />
    </AttributeClass.Attributes>
  </AttributeClass>
```

The supported business type of MyBusiness now has a validator, but remember this is just telling that this business type supports a validator. You still need to write the validation code, which will be covered later in this recipe.

You need to attach a control that specifically uses the MyBusiness type. At the beginning of the Presentation.lsml file, add the following control definition:

```
<Control
    Name="MyBusinessControl"
    SupportedContentItemKind="Value"
    DesignerImageResource="Microsoft.LightSwitch.DesignerImages::TextBox">
    <Control.Attributes>
      <DisplayName Value="$(MyBusinessControl_DisplayName)" />
    </Control.Attributes>
    <Control.SupportedDataTypes>
      <SupportedDataType DataType="MyBusiness"/>
    </Control.SupportedDataTypes>
  </Control>
```

This should not be anything new to you, as you are defining a control based on the internal LightSwitch TextBox control. What is different is that the data type is explicitly set to the business type of MyBusiness.

If you build and add this extension you will create a screen based on the entity defined earlier. You will notice that in the screen designer the control will have a blue exclamation mark. This is because LightSwitch doesn't know what control to associate to the MyBusiness business type. To fix this you need to add a default view mapping. Add the following code below the control definition in the Presentation.lsml:

```
<DefaultViewMapping
    ContentItemKind="Value"
    DataType="MyBusiness"
    View="MyBusinessControl"/>
```

All you have done so far is is related to how the business type and an associated control appear in the design-time of the LightSwitch IDE.

Let's now define how the control appears in the runtime. You first need to create the control in the Client project.

Create a folder called 'Controls' and in the folder create a Silverlight User Control called 'MyBusinessControl'.

In the MyBusinessControl.xaml file the code that adds a Silverlight TxtBox and sets the binding to a scalar value is shown below:

```
<UserControl x:Class="Blank.Controls.MyBusinessControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006">

    <TextBox x:Name="MyBizControl" Text="{Binding Details.Value, Mode=TwoWay}"
TextWrapping="NoWrap"/>

</UserControl>
```

The code-behind in the MyBusinessControl.xaml.cs has no logic at the moment, you will see that shortly. What is important is that the control factory and attribute are defined, shown in the code below:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using Microsoft.LightSwitch.Presentation;
using System.ComponentModel.Composition;

namespace Blank.Controls
{
    public partial class MyBusinessControl : UserControl
    {
        public MyBusinessControl()
        {
            InitializeComponent();
        }
    }

    [Export(typeof(IControlFactory))]
    [ControlFactory(BlankModule.EditorName)]
    internal sealed class MyBusinessControlFactory : BaseControlFactory
    {
        #region Constructor

        public MyBusinessControlFactory() : base("MyBusinessControlTemplate") { }

        #endregion
    }
}
```

The control factory reference to the BlankModule.EditorName is in the BlankModule.cs file in the Client project. The control factory constructor references the base control factory that loads the DataTemplate.xaml file.

The following is the code of the BlankModule.cs containing the reference to the editor name:

```
namespace Blank
{
    internal static class BlankModule
    {
        public const string Name = "Blank";
        private const string ModulePrefix = Name + ":";
        public const string EditorName = ModulePrefix + "MyBusinessControl";
```

```
        internal static class MyBusiness
        {
            public const string Name = "MyBusiness";
            public const string GlobalName = ModulePrefix + Name;
        }
    }
}
```

The base control factory, which is called BaseControlFactory.cs is used to provide the control to LightSwitch and is in the Controls folder of the Client project.

```
using System;
using System.Diagnostics;
using System.IO;
using System.Reflection;
using System.Windows;
using System.Windows.Markup;
using System.Windows.Resources;

using Microsoft.LightSwitch.Presentation;


namespace Blank.Controls
{
    internal abstract class BaseControlFactory :
        IControlFactory
    {
        #region Constructors

        static BaseControlFactory()
        {
            // Ger resource
            Uri resourceUri = GetResourceUri();
            StreamResourceInfo streamInfo =
System.Windows.Application.GetResourceStream(resourceUri);
            Debug.Assert(null != streamInfo, "ResourceDictionary load failed.", "Stream
not found: {0}", resourceUri.ToString());

            if (null != streamInfo)
            {
                // Read resource contents
                string contents = string.Empty;
                using (var reader = new StreamReader(streamInfo.Stream))
                {
                    contents = reader.ReadToEnd();
                }

                // Ensure contents is not empty
                Debug.Assert(!String.IsNullOrEmpty(contents), "ResourceDictionary load
failed.", "Resource was empty: {0}", resourceUri.ToString());

                // Load the contents as resource dictionary
                BaseControlFactory.Dictionary = XamlReader.Load(contents) as
ResourceDictionary;
                Debug.Assert(null != BaseControlFactory.Dictionary, "ResourceDictionary
load failed.", "Resource wasn't ResourceDictionary: {0}", resourceUri.ToString());
            }
```

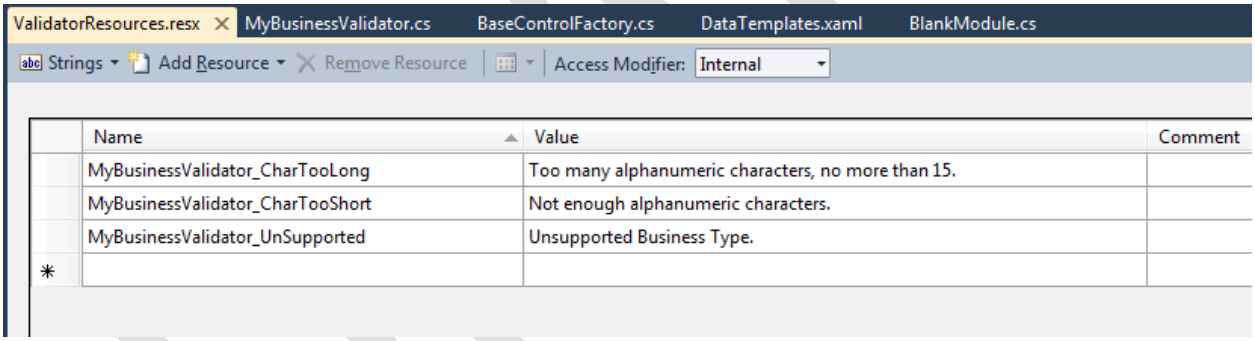
```
        }

        public BaseControlFactory(string templateKey)
        {
            _dataTemplate = BaseControlFactory.GetDataTemplate(templateKey);
        }

        #endregion

        #region IControlFactory Members

        DataTemplate IControlFactory.DataTemplate
        {
            get
            {
                return _dataTemplate;
            }
        }

        DataTemplate IControlFactory.GetDisplayModeDataTemplate(IContentItem contentItem)
        {
            return null;
        }

        #endregion

        #region Private Methods

        private static DataTemplate GetDataTemplate(string templateKey)
        {
            Debug.Assert(BaseControlFactory.Dictionary.Contains(templateKey),
"DataTemplate not found.", "Looking for resource key '{0}'", templateKey);

            return BaseControlFactory.Dictionary[templateKey] as DataTemplate;
        }

        private static Uri GetResourceUri()
        {
            Assembly assembly = Assembly.GetExecutingAssembly();
            AssemblyName thisAssemblyName = new AssemblyName(assembly.FullName);
            return new Uri(thisAssemblyName.Name +
";component/Controls/DataTemplates.xaml", UriKind.Relative);
        }

        #endregion

        #region Private Fields

        private DataTemplate _dataTemplate;

        private static ResourceDictionary Dictionary;

        #endregion
    }
}
```

The resource file that provides the code to the base control factory is the DataTemplate.xaml, which is also contained in the controls folder in the client project and is the following:

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:v="clr-namespace:Blank.Controls;assembly=Blank.Client">

    <DataTemplate x:Key="MyBusinessControlTemplate">
        <v:MyBusinessControl/>
    </DataTemplate>

</ResourceDictionary>
```

So far you have created a business type, default view mapping and the control. In the metadata you indicated that youwant to have validation, but there is no implementation code. You will now add the implementation code.

First create a resource file that will hold the event messages. Create a folder called 'Resources' in the Client project and then create a ValidatorResources.resx file. Add the following name/value pairs:



With this resource file you need to set the Custom Tool Namespace property to Blank.Resources, this allows no namespace issues when you use this file in the Server project.

Now add some additional constants to the BlankModule.cs class file which will support your validation.

```
internal static class ValidationAttribute
        {
            public const string Name = "MyBusinessValidation";
            public const string GlobalName = AttributePrefix + Name;

            public const string MyBusinessRequiredPropertyName = "MyBusinessRequired";
            public const bool MyBusinessRequiredPropertyDefaultValue = true;
        }
```

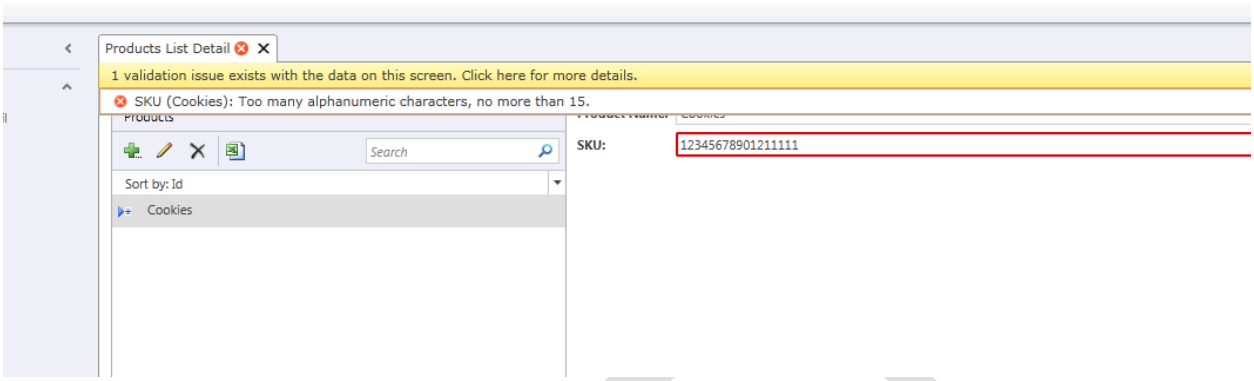
Next, create the MyBusinessValidator.cs class file in the Client project.

```
using System;
using System.Linq;
using System.Collections.Generic;
using Microsoft.LightSwitch.Model;
using Microsoft.LightSwitch.Runtime.Rules;
using Microsoft.LightSwitch;
using System.Text.RegularExpressions;
using System.Diagnostics;
using System.Globalization;
using System.ComponentModel.Composition;

namespace Blank
{
    internal class MyBusinessValidator : IAttachedPropertyValidation
    {
        public void Validate(object value, IPropertyValidationResultsBuilder results)
        {
            string validateValue = value as string;

            if (!String.IsNullOrEmpty(validateValue))
            {
                validateValue = validateValue.Trim();
                int charCount = validateValue.Where(c =>
Char.IsLetterOrDigit(c)).Count();

                if (charCount > 15)
                {
                    results.AddPropertyResult(

Blank.Resources.ValidatorResources.MyBusinessValidator_CharTooLong,
                        ValidationSeverity.Error,

Blank.Resources.ValidatorResources.MyBusinessValidator_CharTooLong);
                }
                else if (charCount < 1)
                {
                    results.AddPropertyResult(

Blank.Resources.ValidatorResources.MyBusinessValidator_CharTooShort,
                        ValidationSeverity.Error,

Blank.Resources.ValidatorResources.MyBusinessValidator_CharTooShort);
                }
            }
        }
    }

    [Export(typeof(IValidationCodeFactory))]
    [ValidationCodeFactory(BlankModule.ValidationAttribute.GlobalName)]
    public class MyBusinessValidatorFactory : IValidationCodeFactory
    {
        public IAttachedValidation Create(IStructuralItem modelItem,
IEnumerable<IAttribute> attributes)
        {
            if (!IsValid(modelItem))
            {
```

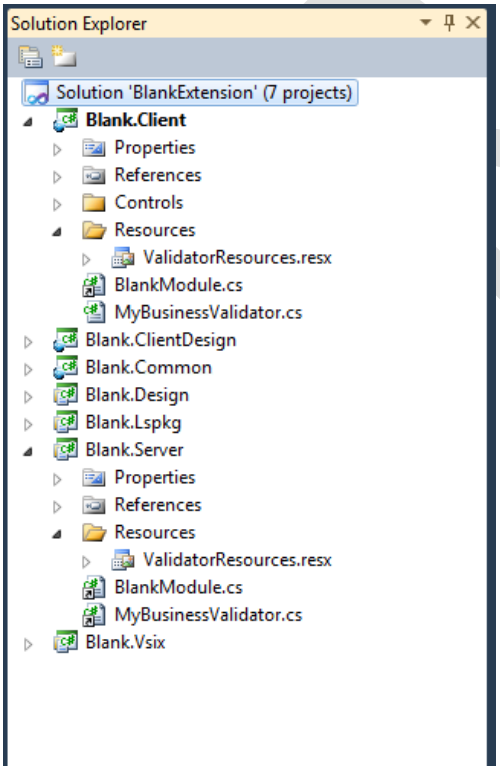
```
                throw new
InvalidOperationException(Blank.Resources.ValidatorResources.MyBusinessValidator_UnSuppor
ted);
            }

            return new MyBusinessValidator();
        }

        public bool IsValid(IStructuralItem modelItem)
        {
            INullableType nullableType = modelItem as INullableType;

            // Get underlying type if it is a INullableType
            modelItem = null != nullableType ? nullableType.UnderlyingType : modelItem;

            return modelItem is ISemanticType &&
String.Equals(((ISemanticType)modelItem).Id, BlankModule.MyBusiness.GlobalName,
StringComparison.Ordinal);
        }
    }
}
```

There are three parts to this class file, the first being an internal class that inherits the IAttachedPropertyValidation interface. This interface provides the capability of displaying the error, warning or informational messages within the screen based on a defined expression; in this case it is character length.
The last part of this code is the ValidatorFactory that allows the validation to be attached to the control.

Now you have provided validation for the client, which is the validation being triggered when a user enters data.

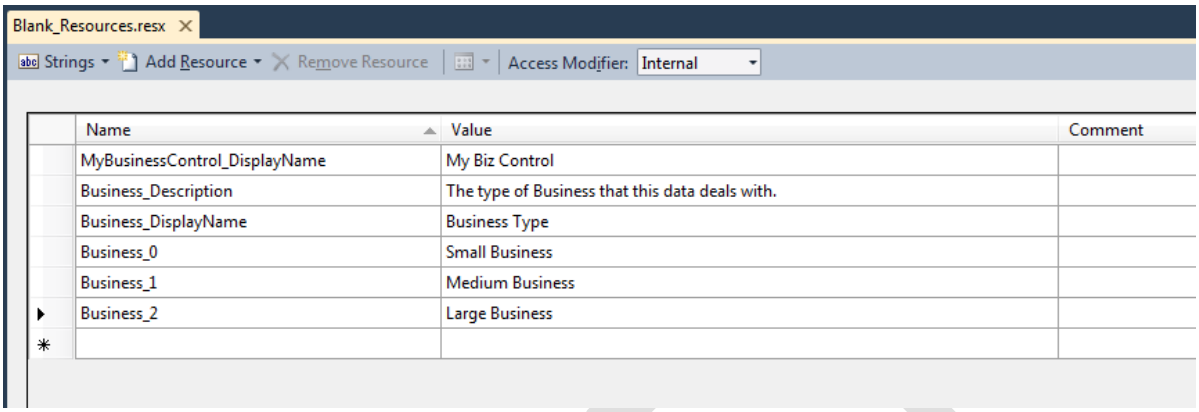


If the record is dirty it is indicated by a glyph; if the user saves the data then the data is saved to the data store. If you required further validation then the same validation code can be linked on the Server project. There may also be a requirement to provide another level of validation on the server tier. Link the MyBusinessValidator.cs class file from the Client project to the Server project and create a Resources folder, and then copy the ValidatorResources.resx file into it. An illustration shows what the end result should look like:



Now that you have a working business type with validation, you can specify an attribute on the business type. You need to add the additional text to the existing resource file, which is Blank_Resource.resx.

For this recipe, add the following name/value pairs:



Name	Value	Comment
MyBusinessControl_DisplayName	My Biz Control	
Business_Description	The type of Business that this data deals with.	
Business_DisplayName	Business Type	
Business_0	Small Business	
Business_1	Medium Business	
Business_2	Large Business	

Next, you need to update the Presentation.lsml file, and change the Attribute Class code to the following:

```
<AttributeClass Name="MyBusinessValidation">
    <AttributeClass.Attributes>
      <Validator />
      <SupportedType Type="MyBusiness?" />
    </AttributeClass.Attributes>
    <AttributeProperty Name="Business" MetaType="Int32">
      <AttributeProperty.Attributes>
        <DisplayName Value="$(Business_DisplayName)" />
        <Description Value="$(Busienss_Description)" />
        <Category Value="Appearance" />
        <SupportedValue DisplayName="$(Business_0)" Value="0" />
        <SupportedValue DisplayName="$(Business_1)" Value="1" />
        <SupportedValue DisplayName="$(Business_2)" Value="2" />
        <SupportedValuesExclusive />
      </AttributeProperty.Attributes>
    </AttributeProperty>
  </AttributeClass>
```

Here the code is specifying the attribute of Business and providing three options. Remember that in this case the attributes are being provided for additional information on the storage type; there is no implementation necessary.

Now build and work with your new business type in a LightSwitch test application. That is the end of the recipe for building a business type.

Shell Extension

This theme recipe enables you to create a shell that allows a developer to create the look and feel of a LightSwitch application.

The ingredients that you are going to need to create a shell extension are the following:

1. Visual Studio 2010 Professional or higher
2. Visual Studio 2010 Service Pack 1
3. Visual Studio 2010 SDK
4. Visual Studio LightSwitch
5. LightSwitch Blank Extension Solution

Once you have all of the ingredients installed on your machine you are ready to create a shell extension.

The first step is to get set up with the right environment. This is done by open the Blank Extension solution sample, which is called BlankExtension.sln.

The first step like when creating a shell extension is to focus on the metadata in the Common project.

The first file that you will add code is the Presentation.lsml file, in the metadata folder. This code is used when you need to specify what is displayed in the properties main page to select the shell.

Add the following code which provides the name and the description of the shell:

```
<Shell Name="MyShell">
  <Shell.Attributes>
    <DisplayName Value="My LightSwitch Shell"/>
    <Description Value="MyCompany"/>
  </Shell.Attributes>
</Shell>
```

You need to update the BlankModule.cs class file to include a static class of the shell name and global name, so that the shell can be referenced by LightSwitch. Add the following code to the file:

```
namespace Blank
{
    internal static class BlankModule
    {
        public const string Name = "Blank";
        private const string ModulePrefix = Name + ":";

        public static class MyShell
        {
            private const string Name = "MyShell";
            public const string GlobalName = ModulePrefix + Name;
        }
    }
}
```

That is all that is required in the Common project. Next you will turn your attention to the Client project, as the shell is all about presentation in the runtime.

The recipe that is shown has been refactored to break up classes to reflect specific functionality, the following files you will create in this list as well as what role these classes play in creating a shell extension.

The MyShellPage.xaml and code behind file provides the following functionality.

- Load event for the shell,
- Access to the Shell Proxy which is the application bridge to LightSwitch Shell internal functionality,
- When an active screen is changed.
- Load the view models. This function gets the view models loaded and cached by the proxy service. An important point here is that the shell is only GUI and binds to the necessary view models. Since, the shell initially has no understanding of the view models the shell needs to load them manually.
- The navigation view model will only return those links that can be executed. In order to do that, it will call the CanExecute() method on the links - which is an asynchronous operation. So, since this shell doesn't bind to the view models there needs to be a way to "prime" the view model by asking for the links. The first command entered in the shell will not work (all commands after that will work).
- To supply an event when active screens change (OnActiveScreensChanged), in particular what action has occurred – add or remove.
- Provide an event on when the current screen has changed (OnCurrentScreenChanged), which gets the notification arguments and updates the screen area accordingly.

The next set of files handles the command support within a shell.

- Command.cs

  The command class that provides the abstract class to the IServiceProxy interface.

- CommandHandler.cs

  The command handler class provides the capability to process a command to first see if the command is one defined by the shell. If the command is not a command defined by the shell, see if it is one that is associated with the active link.

  The support for the CommandViewModel via the GetActiveLinkCommand function which is the commands view model reflects the commands that are associated with the currently active

screen. Search through them for the one that matches the command being passed to this function.

- ExecuteLinkCommand.cs

  The execute function gets the navigation object that represents the link to be executed. The Execute the command by obtaining the dispatcher and invoking the execute method in the correct thread context.

  The FindLink function recursively descends into the navigation object model to find the navigation item with the specified name.

- GlobalCommands.cs

  This code file contains the list of commands that are global across the shell, such as List, Execute, Kill, Goto, and Diagnose screens. These commands in turn reference to the code files below.

  - GotoScreenCommands.cs

    The execute function goes through the collection of active screens and find the one that matches the argument passed in. If the shell finds the screen, set the 'Current' property on the active screens view model to screen. Otherwise, report an error.

    Usage is the following:

    ```
    <goto | g> <screen name> [sequence], where sequence is an integer value indicating to which instance to navigate.
    ```

  - KillScreenCommand.cs

    The execute function goes through the collection of active screens and find the one that matches the argument passed in. If shell finds the screen, close it. Otherwise, report an error.

    Usage is the following:

    ```
    <kill | k> <screen name> [sequence], where sequence is an integer value indicating the instance to kill.
    ```

  - LinkCommand.cs

    The link command code has the LinkCommand function which executes the command by obtaining the dispatcher and invoking the execute method in the correct thread context.

  - ListLinksCommand.cs

This code file contains the execute function that iterates through the navigation view model and displays the available and active links.

The first file you need to create is the MyShellPage Silverlight Page. You are going to divide the page similar to the built-in shell with a command bar, navigation page and the active screen area.

For the purpose of this recipe, it will provide a page that will allow you to interact with the SDKProxy through to the view/model of LightSwitch.

Create the following Silverlight Page called MyShellPage.xaml and add the following xaml code:

```
<navigation:Page x:Class="Blank.MyShellPage"
           xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
           xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
           xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
           mc:Ignorable="d"
           xmlns:navigation="clr-
namespace:System.Windows.Controls;assembly=System.Windows.Controls.Navigation"
           xmlns:myShell="clr-namespace:Blank"
           xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
           d:DesignWidth="640" d:DesignHeight="480"
           Title="MyShell Page">

    <Grid x:Name="LayoutRoot" Background="White">
        <Grid.RowDefinitions>
            <RowDefinition Height="5*"/>
            <RowDefinition Height="5"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <ContentPresenter x:Name="ScreenArea" Grid.Row="0" />
        <sdk:GridSplitter Grid.Row="1" Height="5" Name="gridSplitter1"
Background="LightGray" Margin="0" HorizontalAlignment="Stretch" BorderThickness="2"
BorderBrush="Black"/>
        <Grid Grid.Row="2" Background="White">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="0" />
            </Grid.ColumnDefinitions>
            <myShell:MyRichTextBox x:Name="console" Grid.Column="0" />
            <sdk:GridSplitter Grid.Column="0" Visibility="Collapsed" Width="5"
Name="gridSplitter2" Background="LightGray" Margin="0" VerticalAlignment="Stretch"
BorderThickness="2" BorderBrush="Black"/>
            <sdk:TabControl x:Name="DiagnosticArea" Grid.Column="1"
Visibility="Collapsed" />
        </Grid>
    </Grid>
</navigation:Page>
```

The code above splits the shell into two panes the screen area and command line area.

The following references need to be added to the client project.

C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\LightSwitch\1.0\Client\Microsoft.LightSwitch.ExportProvider.dll

C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\LightSwitch\1.0\Client\Microsoft.LightSwitch.SDKProxy.dll

C:\Program Files (x86)\Microsoft SDKs\Silverlight\v4.0\Libraries\Client\System.Windows.Controls.dll

The next lot of code to add is to the code behine the MyShellPage.xaml.cs, which is the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using System.Windows.Navigation;
using Microsoft.VisualStudio.ExtensibilityHosting;
using Microsoft.LightSwitch.Sdk.Proxy;
using Microsoft.LightSwitch.Runtime.Shell;
using Microsoft.LightSwitch.Runtime.Shell.View;
using Microsoft.LightSwitch.Runtime.Shell.ViewModels.Navigation;
using Microsoft.LightSwitch.Runtime.Shell.ViewModels.Notifications;
using Microsoft.LightSwitch.BaseServices.Notifications;
using Microsoft.LightSwitch.Client;
using Microsoft.LightSwitch.Framework.Client;

namespace Blank
{
    /// <summary>
    /// Represents the main application control.
    /// </summary>
    public partial class MyShellPage : Page, IWeakNotificationSupport
    {
        private IServiceProxy serviceProxy;
        private List<object> weakHelperObjects = new List<object>();

        public MyShellPage()
        {
            this.InitializeComponent();

            this.Loaded += new RoutedEventHandler(MyShellPage_Loaded);
```

```
            // Subscribe to the screen selection changed notification in order to get
            // notified when the active screen referred to by the active screen view
            // model changes.

this.ServiceProxy.NotificationService.Subscribe(typeof(ScreenSelectionChangedNotification
), this.OnCurrentScreenChanged);

this.ServiceProxy.NotificationService.Subscribe(typeof(ScreenOpenedNotification),
this.OnScreenOpened);

this.ServiceProxy.NotificationService.Subscribe(typeof(ScreenClosedNotification),
this.OnScreenClosed);

            this.LoadViewModels();
            this.console.ShowPrompt();
        }

        private IServiceProxy ServiceProxy
        {
            get
            {
                // Get the service proxy which provides access to the needed
                // LightSwitch services.
                if (null == this.serviceProxy)
                    this.serviceProxy =
VsExportProviderService.GetExportedValue<IServiceProxy>();

                return this.serviceProxy;
            }
        }

        #region IWeakNotificationSupport methods
        public void HoldObject(object o)
        {
            this.weakHelperObjects.Add(o);
        }
        #endregion

        private void LoadViewModels()
        {
            // Get the view models loaded (and cached) by the proxy service.
            //
            // A GUI shell will probably end up binding to these view models.  Since this
            // shell doesn't do that, the view models need to be loaded manually.
            if (null == this.ServiceProxy.NavigationViewModel)
                throw new System.InvalidOperationException("Unable to retrieve navigation
view model");
            else
            {
                // The navigation view model will only return those links that can be
executed.  In order
                // to do that, it will call the CanExecute() method on the links - which
is an
                // asynchronous operation.  So, since this shell doesn't bind to the view
models I need
                // to "prime" the view model by asking for the links.  If I don't do
this, the first
```

```
                // command entered in the shell will not work (all commands after that
will work).
                this.ServiceProxy.NavigationViewModel.NavigationItems.Count();
            }

            if (null == this.ServiceProxy.CommandsViewModel)
                throw new System.InvalidOperationException("Unable to retrieve commands
view model");

            if (null == this.ServiceProxy.ActiveScreensViewModel)
                throw new System.InvalidOperationException("Unable to retrieve active
screens view model");
        }

        private void OnScreenOpened(Notification n)
        {
            ScreenOpenedNotification screenOpenedNotification =
(ScreenOpenedNotification)n;
            IScreenObject screenObject = screenOpenedNotification.Screen;
            IScreenView view =
this.ServiceProxy.ScreenViewService.GetScreenView(screenObject);

            // Set the screen area to the UI element of the screen view.
            this.ScreenArea.Content = view.RootUI;

            // Set the currently active screen in the active screens view model.
            this.ServiceProxy.ActiveScreensViewModel.Current = screenObject;
        }

        private void OnScreenClosed(Notification n)
        {
            // A screen has been close and therefore removed from the application's
            // collection of active screens.  Get the IScreenView from the active screen
            // object and set the screen area content to its RootUI property.
            int count = ClientApplicationProvider.Current.ActiveScreens.Count();
            if (count > 0)
            {
                ScreenClosedNotification screenClosedNotification =
(ScreenClosedNotification)n;
                IScreenObject screenObject = screenClosedNotification.Screen;
                IScreenView view =
this.ServiceProxy.ScreenViewService.GetScreenView(screenObject);
                this.ScreenArea.Content = view.RootUI;
            }
            else
                this.ScreenArea.Content = null;
        }

        private void OnCurrentScreenChanged(Notification n)
        {
            // The current active screen has changed.  Get the screen object from
            // the notification arguments and update the screen area content
            // accordingly.
            ScreenSelectionChangedNotification sscn =
(ScreenSelectionChangedNotification)n;
            IScreenObject screenObject = sscn.SelectedScreen;
            IScreenView view =
this.ServiceProxy.ScreenViewService.GetScreenView(screenObject);
```

```
            this.ScreenArea.Content = view.RootUI;
        }

        private void MyShellPage_Loaded(object sender, RoutedEventArgs e)
        {
            try
            {
                System.Windows.Browser.HtmlPage.Plugin.Focus();
            }
            catch (InvalidOperationException)
            {
            }

            this.console.Focus();
        }
    }
}
```

The code in this file is well commented and you can see how it works, where the main interactive events are recognized such as screen opened, closed, on change and when the shell is loaded.

The next class file to create in the client project is the MyShell.cs class file, which contains the following code:

```
using System;

using System.ComponentModel.Composition;
using Microsoft.LightSwitch.Runtime.Shell;

namespace Blank
{
    /// <summary>
    /// The Custom shell.
    /// </summary>
    [Export(typeof(IShell))]
    [Shell(BlankModule.MyShell.GlobalName)]
    internal class MyShell : IShell
    {
         public string Name
        {
            get { return BlankModule.MyShell.GlobalName; }
        }

        public Uri ShellUri
        {
            get { return new Uri(@"/Blank.Client;component/MyShellPage.xaml",
UriKind.Relative); }
        }
    }
}
```

The MyShell class utilizes the IShell interface which allows the sample to leverage the LightSwitch shell capabilities. Another piece of code within this file is the uri reference to what the shell UI is composed of.

Let's now create a rich text box control that will be used in the command pane of the shell.

Create a class file called MyRichTextBox.cs to the client project and add the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Ink;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using Microsoft.LightSwitch.Sdk.Proxy;
using Microsoft.VisualStudio.ExtensibilityHosting;

namespace Blank
{
    public class MyRichTextBox : RichTextBox
    {
        private List<string> commandHistory;
        private int insertionIndex;
        private int firstCommandIndex;
        private int currentIndex;
        private bool gotFirstWidth;
        private GridLength diagnosticWidth;
        private const int maxHistoryIndex = 64;
        private const int maxHistory = 50;

        protected override void OnKeyDown(KeyEventArgs e)
        {
            if (e.Key == Key.Up)
                this.OnUpArrowPressed(this, e);
            else if (e.Key == Key.Down)
                this.OnDownArrowPressed(this, e);
            else if (e.Key == Key.Enter)
                this.OnEnterPressed(this, e);
            else if (e.Key == Key.Escape)
                this.OnEscPressed(this, e);

            if (!e.Handled)
                base.OnKeyDown(e);
        }

        private List<string> CommandHistory
        {
            get
```

```
            {
                if (null == this.commandHistory)
                    this.commandHistory = new
List<string>(MyRichTextBox.maxHistoryIndex);

                return this.commandHistory;
            }
        }

        private void StoreCommand(string command)
        {
            bool replace = this.CommandHistory.Count == MyRichTextBox.maxHistoryIndex;

            if (replace)
                this.CommandHistory[this.insertionIndex] = command;
            else
                this.CommandHistory.Add(command);

            if (this.insertionIndex == MyRichTextBox.maxHistoryIndex - 1)
                this.insertionIndex = 0;
            else
                this.insertionIndex++;

            if (this.IndexDifferential(this.insertionIndex, this.firstCommandIndex) ==
(MyRichTextBox.maxHistory + 1))
            {
                if (this.firstCommandIndex == MyRichTextBox.maxHistoryIndex - 1)
                    this.firstCommandIndex = 0;
                else
                    this.firstCommandIndex++;
            }
        }

        private int IndexDifferential(int index1, int index2)
        {
            if (index1 - index2 >= 0)
                return index1 - index2;

            return (MyRichTextBox.maxHistoryIndex - index2) + index1;
        }

        private int GetPreceedingCommandIndex(int index)
        {
            if (index == 0)
                return (this.firstCommandIndex == 0) ? 0 : this.CommandHistory.Count - 1;

            if (index == this.firstCommandIndex)
                return this.firstCommandIndex;

            return --index;
        }

        private int GetNextCommandIndex(int index)
        {
            if (index == MyRichTextBox.maxHistoryIndex - 1)
                return (this.insertionIndex == 0) ? index : 0;

            if (index == this.insertionIndex - 1)
```

```
            return index;

        if (index == this.insertionIndex)
            return index - 1;

        return ++index;
    }

    private string GetPreviousCommand()
    {
        this.currentIndex = this.GetPreceedingCommandIndex(this.currentIndex);

        if (this.CommandHistory.Count > 0)
            return this.CommandHistory[this.currentIndex];

        return String.Empty;
    }

    private string GetNextCommand()
    {
        this.currentIndex = this.GetNextCommandIndex(this.currentIndex);

        if (this.CommandHistory.Count > 0)
            return this.CommandHistory[this.currentIndex];

        return String.Empty;
    }

    private void ShowCommand(string command)
    {
        int blockCount = this.Blocks.Count();
        Paragraph p = (Paragraph)this.Blocks[blockCount - 1];
        Run r = new Run();

        r.Text = command;

        p.Inlines.Add(r);

        this.Selection.Select(this.ContentEnd, this.ContentEnd);
    }

    private void OnUpArrowPressed(object sender, KeyEventArgs e)
    {
        this.ClearPrompt();
        this.ShowCommand(this.GetPreviousCommand());

        e.Handled = true;
    }

    private void OnDownArrowPressed(object sender, KeyEventArgs e)
    {
        this.ClearPrompt();
        this.ShowCommand(this.GetNextCommand());

        e.Handled = true;
    }

    private void OnEnterPressed(object sender, KeyEventArgs e)
```

```
        {
            this.ProcessNewCommand(this.RetrieveTypedCommand());
            this.ShowPrompt();

            e.Handled = true;
        }

        private void OnEscPressed(object sender, KeyEventArgs e)
        {
            this.ClearPrompt();
        }

        private void ClearPrompt()
        {
            int blockCount = this.Blocks.Count();

            this.Blocks.RemoveAt(blockCount - 1);

            this.ShowPrompt();
        }

        private string RetrieveTypedCommand()
        {
            int blockCount = this.Blocks.Count();
            Paragraph p = (Paragraph)this.Blocks[blockCount - 1];
            Run r = (Run)p.Inlines[p.Inlines.Count() - 1];

            return r.Text.Trim();
        }

        private void ProcessNewCommand(string command)
        {
            string[] args = command.Split(new char[] { ' ' },
StringSplitOptions.RemoveEmptyEntries);

            try
            {
                string commandResult = CommandHandler.ProcessCommand(args);

                if ((null != commandResult) && (commandResult.Length > 0))
                {
                    if (String.Equals("#sd", commandResult))
                        this.ShowDiagnosticControls(true);
                    else if (String.Equals("#hd", commandResult))
                        this.ShowDiagnosticControls(false);
                    else
                        this.WriteLine(commandResult);
                }
            }
            finally
            {
                this.StoreCommand(command);

                this.currentIndex = this.insertionIndex;
            }
        }

        private void ShowDiagnosticControls(bool show)
```

```
        {
            IEnumerable<Lazy<Control, IDictionary<string, object>>> controlExports =
VsExportProviderService.GetExports<Control, IDictionary<string,
object>>("LightSwitch.DiagnosticControl");

            if (controlExports.Count() > 0)
            {
                Grid parent = (Grid)this.Parent;
                ColumnDefinitionCollection columns = parent.ColumnDefinitions;

                if (show)
                {
                    if (!this.gotFirstWidth)
                        columns[1].Width = new GridLength(1, GridUnitType.Star);
                    else
                        columns[1].Width = this.diagnosticWidth;

                    foreach (UIElement element in parent.Children)
                    {
                        GridSplitter splitter = element as GridSplitter;

                        if (null != splitter)
                            splitter.Visibility = Visibility.Visible;
                        else
                        {
                            TabControl tabControl = element as TabControl;

                            if (null != tabControl)
                            {
                                tabControl.Visibility = Visibility.Visible;

                                if (0 == tabControl.Items.Count)
                                {
                                    foreach (Lazy<Control, IDictionary<string, object>>
controlExport in controlExports)
                                    {
                                        TabItem item = new TabItem();

                                        item.Header = controlExport.Metadata["Name"];
                                        item.Content = controlExport.Value;

                                        tabControl.Items.Add(item);
                                    }
                                }
                            }
                        }
                    }
                }
                else
                {
                    this.gotFirstWidth = true;
                    this.diagnosticWidth = columns[1].Width;

                    columns[1].Width = new GridLength(0);

                    foreach (UIElement element in parent.Children)
                    {
                        GridSplitter splitter = element as GridSplitter;
```

```
                        if (null != splitter)
                            splitter.Visibility = Visibility.Collapsed;
                        else
                        {
                            TabControl tabControl = element as TabControl;

                            if (null != tabControl)
                                tabControl.Visibility = Visibility.Collapsed;
                        }
                    }
                }
            }
            else
                this.WriteLine("There are no diagnostic controls installed.");
        }

        private void WriteLine(string line)
        {
            Paragraph p = new Paragraph();
            Run r = new Run();

            r.Text = line;

            p.Inlines.Add(r);

            this.Blocks.Add(p);
        }

        internal void ShowPrompt()
        {
            Paragraph p = new Paragraph();
            Bold b = new Bold();
            Run r = new Run();

            b.Inlines.Add(">");
            r.Text = " ";

            p.Inlines.Add(b);
            p.Inlines.Add(r);

            this.Blocks.Add(p);
            this.Selection.Select(this.ContentEnd, this.ContentEnd);
        }
    }
}
```

The code here is straight forward and passes commands on to the other classes which we will cover in this recipe.

Create the command.cs class file with the following code:

```
using System;
using System.Net;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
```

```
using System.Windows.Ink;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using Microsoft.VisualStudio.ExtensibilityHosting;
using Microsoft.LightSwitch.Sdk.Proxy;
using Microsoft.LightSwitch.Runtime.Shell.ViewModels.Navigation;

namespace Blank
{
    internal abstract class Command
    {
        private IServiceProxy   serviceProxy;

        protected IServiceProxy ServiceProxy
        {
            get
            {
                if ( null == this.serviceProxy )
                    this.serviceProxy =
VsExportProviderService.GetExportedValue<IServiceProxy>();

                return this.serviceProxy;
            }
        }

        internal abstract string Execute(string[] args);
        internal abstract string DisplayHelp();
    }
}
```

This class provides access the sdk proxy via the IServiceProxy interface.

The next file that handles the commands is the CommandHandler.cs class file, which has the following code:

```
using System;
using System.Collections.Generic;
using System.Collections.Specialized;
using Microsoft.LightSwitch.Sdk.Proxy;
using Microsoft.LightSwitch.Runtime.Shell.View;
using Microsoft.LightSwitch.Runtime.Shell.ViewModels.Notifications;
using Microsoft.LightSwitch.Runtime.Shell.ViewModels.Commands;
using Microsoft.LightSwitch.BaseServices.Notifications;
using Microsoft.LightSwitch.Client;
using Microsoft.LightSwitch.Framework.Client;
using Microsoft.VisualStudio.ExtensibilityHosting;

namespace Blank
{
    internal static class CommandHandler
    {
        private static IServiceProxy serviceProxy;

        private static IServiceProxy ServiceProxy
        {
```

```
            get
            {
                if ( null == CommandHandler.serviceProxy )
                    CommandHandler.serviceProxy =
VsExportProviderService.GetExportedValue<IServiceProxy>();

                return CommandHandler.serviceProxy;
            }
        }

        internal static string ProcessCommand(string[] args)
        {
            // First, see if the command is one defined by this shell.
            Command command = GlobalCommands.GetGlobalCommand(args[0]);

            if ( null != command )
                return command.Execute(args);

            // This is not a command defined by the shell, so see if it
            // is one that is associated with the active link.
            command = CommandHandler.GetActiveLinkCommand(args[0]);

            if ( null != command )
                return command.Execute(args);

            return String.Format("'{0}' is not a known command.", args[0]);
        }

        internal static Command GetActiveLinkCommand(string command)
        {
            // The commands view model reflects the commands that are associated with the
            // currently active screen.  Search through them for the one that matches
            // the command being passed to this function.
            foreach(IShellCommand shellCommand in
CommandHandler.ServiceProxy.CommandsViewModel.ShellCommands)
            {
                if ( String.Equals(shellCommand.DisplayName, command,
StringComparison.OrdinalIgnoreCase) )
                    return new LinkCommand(shellCommand);
            }

            return null;
        }
    }
}
```

The next class file to create in the client project is the ExecuteLinkCommand.cs file. Add the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
```

```
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using Microsoft.VisualStudio.ExtensibilityHosting;
using Microsoft.LightSwitch.Sdk.Proxy;
using Microsoft.LightSwitch.Runtime.Shell.ViewModels.Navigation;
using Microsoft.LightSwitch.Client;
using Microsoft.LightSwitch.Framework.Client;
using Microsoft.LightSwitch.Threading;
using Microsoft.LightSwitch.Details;

namespace Blank
{
    internal class ExecuteLinkCommand : Command
    {
        internal override string Execute(string[] args)
        {
            string result = null;
            string linkName = null;

            if (args.Length > 2)
            {
                linkName = String.Join(" ", args, 1, args.Length - 1);

                // Remove any quotes
                if (linkName[0] == '"')
                    linkName = linkName.Substring(1);

                if (linkName[linkName.Length - 1] == '"')
                    linkName = linkName.Substring(0, linkName.Length - 1);

            }
            else
                linkName = args[1];

            if ((null != linkName) && (linkName.Length > 0))
            {
                // Get the navigation object that represents the link to
                // be executed.
                INavigationScreen link = this.FindLink(linkName);

                if (null != link)
                {
                    // Execute the command by obtaining the dispacther and invoking
                    // the execute method in the correct thread context.
                    IBusinessCommand businessCommand =
(IBusinessCommand)link.ExecutableObject;
                    IDispatcherObject dispatcher = (IDispatcherObject)businessCommand;

                    dispatcher.Dispatcher.BeginInvoke(delegate()
                    {
                        if (businessCommand.CanExecute)
                            businessCommand.Execute();
                        else
                            result = "Unable to execute link.";
                    });
```

```
                    result = String.Format("Link '{0}' has been executed.", linkName);
                }
                else
                    result = String.Format("A link named '{0}' could not be found.",
linkName);
            }

            if (null == result)
                result = this.DisplayHelp();

            return result;
        }

        internal override string DisplayHelp()
        {
            return "Help yourself!";
        }

        private INavigationScreen FindLink(string name)
        {
            // Recursively descend into the navigation object model to find the
            // navigation item with the specified name.
            INavigationScreen link = null;

            foreach (INavigationItem navItem in
this.ServiceProxy.NavigationViewModel.NavigationItems)
            {
                if (navItem is INavigationGroup)
                    link = this.FindLink((INavigationGroup)navItem, name);
                else if (navItem is INavigationScreen)
                {
                    if (String.Equals(((INavigationScreen)navItem).DisplayName, name,
StringComparison.OrdinalIgnoreCase))
                        link = (INavigationScreen)navItem;
                }

                if (null != link)
                    break;
            }

            return link;
        }

        private INavigationScreen FindLink(INavigationGroup navGroup, string name)
        {
            INavigationScreen link = null;

            foreach (INavigationItem navItem in navGroup.Children)
            {
                if (navItem is INavigationGroup)
                    link = this.FindLink((INavigationGroup)navItem, name);
                else if (navItem is INavigationScreen)
                {
                    if (String.Equals(((INavigationScreen)navItem).DisplayName, name,
StringComparison.OrdinalIgnoreCase))
                        link = (INavigationScreen)navItem;
                }
```

```
                if (null != link)
                    break;
            }

            return link;
        }
    }
}
```

Continue to create another class file called GlobalCommands.cs with the following code in the client project:

```
using System;
using System.Collections.Generic;
using System.Collections.Specialized;
using Microsoft.VisualStudio.ExtensibilityHosting;
using Microsoft.LightSwitch.Sdk.Proxy;
using Microsoft.LightSwitch.Runtime.Shell.View;
using Microsoft.LightSwitch.Runtime.Shell.ViewModels.Navigation;
using Microsoft.LightSwitch.Runtime.Shell.ViewModels.Notifications;
using Microsoft.LightSwitch.BaseServices.Notifications;
using Microsoft.LightSwitch.Client;
using Microsoft.LightSwitch.Framework.Client;

namespace Blank
{
    internal static class GlobalCommands
    {
        internal static Command GetGlobalCommand(string commandName)
        {
            Command command = null;

            if (GlobalCommands.IsListCommand(commandName))
                command = new ListLinksCommand();
            else if (GlobalCommands.IsExecuteCommand(commandName))
                command = new ExecuteLinkCommand();
            else if (GlobalCommands.IsKillCommand(commandName))
                command = new KillScreenCommand();
            else if (GlobalCommands.IsGotoCommand(commandName))
                command = new GotoScreenCommand();

            return command;
        }

        private static bool IsListCommand(string command)
        {
            if (String.Equals(command, "list", StringComparison.OrdinalIgnoreCase) ||
String.Equals(command, "l", StringComparison.OrdinalIgnoreCase))
                return true;

            return false;
        }

        private static bool IsExecuteCommand(string command)
        {
            if (String.Equals(command, "execute", StringComparison.OrdinalIgnoreCase) ||
String.Equals(command, "x", StringComparison.OrdinalIgnoreCase))
```

```
                return true;

            return false;
        }

        private static bool IsKillCommand(string command)
        {
            if (String.Equals(command, "kill", StringComparison.OrdinalIgnoreCase) ||
String.Equals(command, "k", StringComparison.OrdinalIgnoreCase))
                return true;

            return false;
        }

        private static bool IsGotoCommand(string command)
        {
            if (String.Equals(command, "Goto", StringComparison.OrdinalIgnoreCase) ||
String.Equals(command, "g", StringComparison.OrdinalIgnoreCase))
                return true;

            return false;
        }

        private static bool IsDiagnosticsCommand(string command)
        {
            if (String.Equals(command, "Diagnostics", StringComparison.OrdinalIgnoreCase)
|| String.Equals(command, "d", StringComparison.OrdinalIgnoreCase))
                return true;

            return false;
        }
    }
}
```

Create the GotoScreenCommand.cs class file with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using Microsoft.VisualStudio.ExtensibilityHosting;
using Microsoft.LightSwitch.Sdk.Proxy;
using Microsoft.LightSwitch.Runtime.Shell.ViewModels.Navigation;
using Microsoft.LightSwitch.Client;
using Microsoft.LightSwitch.Framework.Client;
using Microsoft.LightSwitch.Threading;
using Microsoft.LightSwitch.Details;
```

```
namespace Blank
{
    internal class GotoScreenCommand : Command
    {
        internal override string Execute(string[] args)
        {
            string screenName = null;
            int desiredInstance = 1;

            if (args.Length > 2)
            {
                // Quotes are required if screen name has spaces.
                bool foundQuote = false;
                int argsInName = 0;

                foreach (string arg in args)
                {
                    if (arg[0] == '"')
                    {
                        foundQuote = true;
                        screenName = arg;
                        argsInName++;
                    }
                    else if ((arg[arg.Length - 1] == '"') && foundQuote)
                    {
                        screenName = String.Concat(screenName, String.Format(" {0}",
arg));
                        argsInName++;
                        break;
                    }
                    else if (foundQuote)
                    {
                        screenName = String.Concat(screenName, String.Format(" {0}",
arg));
                        argsInName++;
                    }
                }

                if (foundQuote)
                {
                    // Remove any quotes
                    if (screenName[0] == '"')
                        screenName = screenName.Substring(1);

                    if (screenName[screenName.Length - 1] == '"')
                        screenName = screenName.Substring(0, screenName.Length - 1);

                    // Was a desired instance specified?
                    if (argsInName == (args.Length - 2))
                    {
                        if (!Int32.TryParse(args[args.Length - 1], out desiredInstance))
                            return this.DisplayHelp();
                    }
                }
                else if (args.Length == 3)
                {
                    screenName = args[1];
```

```
                    if (!Int32.TryParse(args[2], out desiredInstance))
                        return this.DisplayHelp();
                }

            }
            else
                screenName = args[1];

            if (null != screenName)
            {
                IScreenObject desiredScreen = null;
                int instance = 0;


                // Go through the collection of active screens and find the one that
matches
                // the argument passed in.
                foreach (IActiveScreen screen in
ClientApplicationProvider.Current.ActiveScreens)
                {
                    IScreenObject screenObject =
ClientApplicationProvider.Current.Details.GetScreen(screen);

                    if (String.Equals(screenObject.DisplayName, screenName,
StringComparison.OrdinalIgnoreCase))
                    {
                        if (++instance == desiredInstance)
                        {
                            desiredScreen = screenObject;
                            break;
                        }
                    }
                }

                // If we found the screen, set the 'Current' property on the active
screens
                // view model to screen.  Otherwise, report an error.
                if (null != desiredScreen)
                {
                    this.ServiceProxy.ActiveScreensViewModel.Current = desiredScreen;

                    return null;
                }
                else
                    return String.Format("There is not a {0}{1} instance of a screen
named '{2}' to navigate to.",
                                         desiredInstance,
                                         desiredInstance % 10 == 1 ? "st" :
(desiredInstance % 10 == 2 ? "nd" : (desiredInstance % 10 == 3 ? "rd" : "th")),
                                         args[1]);
            }

            return this.DisplayHelp();
        }

        internal override string DisplayHelp()
        {
```

```
            return "usage: <goto | g> <screen name> [sequence], where sequence is an
integer value indicating to which instance to navigate.";
        }
    }
}
```

Add the KillScreenCommand.cs class file with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using Microsoft.VisualStudio.ExtensibilityHosting;
using Microsoft.LightSwitch.Sdk.Proxy;
using Microsoft.LightSwitch.Runtime.Shell.ViewModels.Navigation;
using Microsoft.LightSwitch.Client;
using Microsoft.LightSwitch.Framework.Client;
using Microsoft.LightSwitch.Threading;
using Microsoft.LightSwitch.Details;

namespace Blank
{
    internal class KillScreenCommand : Command
    {
        internal override string Execute(string[] args)
        {
            string screenName = null;
            int desiredInstance = 1;

            if (args.Length > 2)
            {
                // Quotes are required if screen name has spaces.
                bool foundQuote = false;
                int argsInName = 0;

                foreach (string arg in args)
                {
                    if (arg[0] == '"')
                    {
                        foundQuote = true;
                        screenName = arg;
                        argsInName++;
                    }
                    else if ((arg[arg.Length - 1] == '"') && foundQuote)
                    {
                        screenName = String.Concat(screenName, String.Format(" {0}",
arg));
                        argsInName++;
                        break;
```

```
                    }
                    else if (foundQuote)
                    {
                        screenName = String.Concat(screenName, String.Format(" {0}",
arg));
                        argsInName++;
                    }
                }

                if (foundQuote)
                {
                    // Remove any quotes
                    if (screenName[0] == '"')
                        screenName = screenName.Substring(1);

                    if (screenName[screenName.Length - 1] == '"')
                        screenName = screenName.Substring(0, screenName.Length - 1);

                    // Was a desired instance specified?
                    if (argsInName == (args.Length - 2))
                    {
                        if (!Int32.TryParse(args[args.Length - 1], out desiredInstance))
                            return this.DisplayHelp();
                    }
                }
                else if (args.Length == 3)
                {
                    screenName = args[1];

                    if (!Int32.TryParse(args[2], out desiredInstance))
                        return this.DisplayHelp();
                }

            }
            else
                screenName = args[1];

            if (null != screenName)
            {
                IActiveScreen activeScreen = null;
                int instance = 0;


                // Go through the collection of active screens and find the one that
matches
                // the argument passed in.
                foreach (IActiveScreen screen in
ClientApplicationProvider.Current.ActiveScreens)
                {
                    IScreenObject screenObject =
ClientApplicationProvider.Current.Details.GetScreen(screen);

                    if (String.Equals(screenObject.DisplayName, screenName,
StringComparison.OrdinalIgnoreCase))
                    {
                        if (++instance == desiredInstance)
                        {
                            activeScreen = screen;
```

```
                            break;
                        }
                    }
                }

                // If we found the screen, close it.  Otherwise, report an error.
                if (null != activeScreen)
                {
                    activeScreen.Close(false);

                    return null;
                }
                else
                    return String.Format("There is not a {0}{1} instance of a screen
named '{2}' to kill.",
                                         desiredInstance,
                                         desiredInstance % 10 == 1 ? "st" :
(desiredInstance % 10 == 2 ? "nd" : (desiredInstance % 10 == 3 ? "rd" : "th")),
                                         args[1]);
            }

            return this.DisplayHelp();
        }

        internal override string DisplayHelp()
        {
            return "usage: <kill | k> <screen name> [sequence], where sequence is an
integer value indicating the instance to kill.";
        }
    }
}
```

Add the LinkCommand.cs class file to the Client project with the following code:

```
using System;
using System.Collections.Generic;
using Microsoft.VisualStudio.ExtensibilityHosting;
using Microsoft.LightSwitch.Sdk.Proxy;
using Microsoft.LightSwitch.Runtime.Shell.ViewModels.Navigation;
using Microsoft.LightSwitch.Runtime.Shell.ViewModels.Commands;
using Microsoft.LightSwitch.Threading;
using Microsoft.LightSwitch.Details;

namespace Blank
{
    internal class LinkCommand : Command
    {
        private IShellCommand command;

        internal LinkCommand(IShellCommand shellCommand)
        {
            this.command = shellCommand;
        }

        internal override string Execute(string[] args)
        {
            // Execute the command by obtaining the dispacther and invoking
```

```
            // the execute method in the correct thread context.
            string result = String.Empty;
            IBusinessCommand businessCommand =
(IBusinessCommand)this.command.ExecutableObject;
            IDispatcherObject dispatcher = (IDispatcherObject)businessCommand;

            dispatcher.Dispatcher.BeginInvoke(delegate()
            {
                if (businessCommand.CanExecute)
                    businessCommand.Execute();
                else
                    result = "Unable to execute link.";
            });

            return result;
        }

        internal override string DisplayHelp()
        {
            return this.command.Description;
        }
    }
}
```

Add the ListLinksCommand.cs class file with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;
using Microsoft.VisualStudio.ExtensibilityHosting;
using Microsoft.LightSwitch.Sdk.Proxy;
using Microsoft.LightSwitch.Runtime.Shell.ViewModels.Navigation;
using Microsoft.LightSwitch.Client;
using Microsoft.LightSwitch.Framework.Client;

namespace Blank
{
    internal class ListLinksCommand : Command
    {
        internal override string Execute(string[] args)
        {
            StringBuilder sb = new StringBuilder(1024);

            foreach (INavigationItem navItem in
this.ServiceProxy.NavigationViewModel.NavigationItems)
            {
                if (navItem is INavigationGroup)
                    sb.Append(this.DisplayGroup((INavigationGroup)navItem));
```

```
                else if (navItem is INavigationScreen)
                {
                    bool isDefault = (null !=
this.ServiceProxy.NavigationViewModel.DefaultItem) ?
(this.ServiceProxy.NavigationViewModel.DefaultItem == navItem ? true : false) : false;

                    sb.Append(this.DisplayLink((INavigationScreen)navItem, isDefault));
                }
            }

            sb.Append(this.DisplayActiveLinks());

            return sb.ToString();
        }

        internal override string DisplayHelp()
        {
            return "Help yourself!";
        }

        private string DisplayGroup(INavigationGroup navGroup)
        {
            StringBuilder sb = new StringBuilder(1024);

            sb.Append(String.Format("{0} [G]\r\n", navGroup.DisplayName));

            foreach (INavigationItem navItem in navGroup.Children)
            {
                if (navItem is INavigationGroup)
                    sb.Append(this.DisplayGroup((INavigationGroup)navItem));
                else if (navItem is INavigationScreen)
                {
                    bool isDefault = (null != navGroup.DefaultChild) ?
(navGroup.DefaultChild == navItem ? true : false) : false;

                    sb.Append(this.DisplayLink((INavigationScreen)navItem, isDefault));
                }
            }

            return sb.ToString();
        }

        private string DisplayLink(INavigationScreen navScreen, bool isDefault)
        {
            return String.Format("    {0} [L]{1}\r\n", navScreen.DisplayName, isDefault ?
"[D]" : "");
        }

        private string DisplayActiveLinks()
        {
            if (ClientApplicationProvider.Current.ActiveScreens.Count() > 0)
            {
                StringBuilder sb = new StringBuilder(1024);
                Dictionary<string, int> activeLinks = new Dictionary<string, int>();
                List<string> linkOrder = new List<string>();

                sb.Append("Active Links:\r\n");
```

```
                foreach (IActiveScreen activeLink in
ClientApplicationProvider.Current.ActiveScreens)
                {
                    IScreenObject screenObject =
ClientApplicationProvider.Current.Details.GetScreen(activeLink);
                    int count = 0;

                    if (!activeLinks.TryGetValue(screenObject.DisplayName, out count))
                    {
                        activeLinks.Add(screenObject.DisplayName, 1);

                        linkOrder.Add(screenObject.DisplayName);
                    }
                    else
                        activeLinks[screenObject.DisplayName] = ++count;
                }

                foreach (string linkName in linkOrder)
                {
                    int count = activeLinks[linkName];

                    if (count == 1)
                        sb.Append(String.Format("    {0}\r\n", linkName));
                    else
                        sb.Append(String.Format("    {0} [{1}]\r\n", linkName, count));
                }

                return sb.ToString();
            }

            return String.Empty;
        }
    }
}
```

Now that all the code is added to the shell extension. You also need to include the SDK Proxy assembly as part of the LsPkg project. To do this you need to go to the Blank.LsPkg project and unload the project. Next, edit the Blank.LsPkg.csproj.

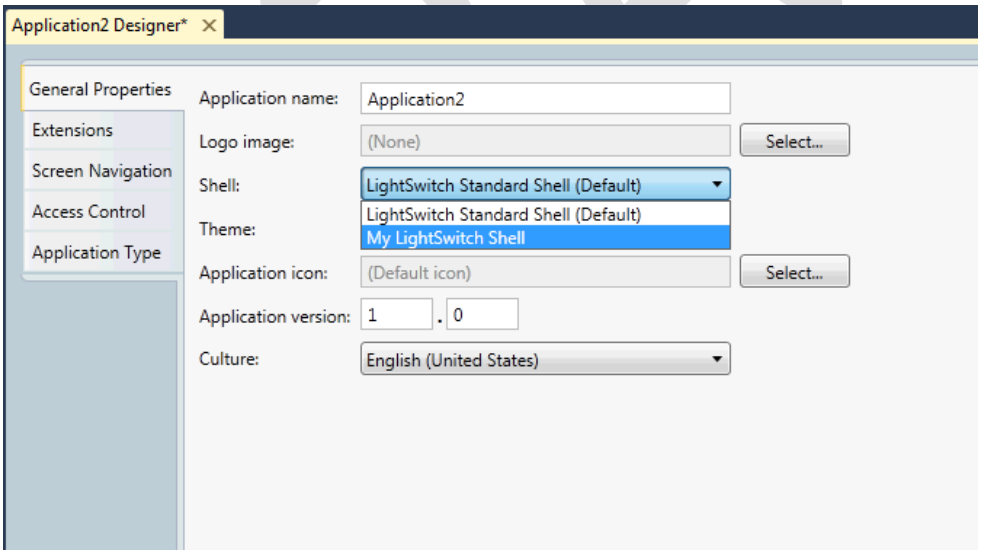
Within the first item group of the project file you need to insert the following code:

```
<ItemGroup>
    <ClientGeneratedReference Include="..\Blank.Client\$(ProjectPath)\Blank.Client.dll"
/>
    <ClientGeneratedReference Include="..\Blank.Common\$(ProjectPath)\Blank.Common.dll"
/>
    <ClientDebugOnlyReference
Include="..\Blank.ClientDesign\$(ProjectPath)\Blank.Client.Design.dll" />
    <ClientGeneratedReference
Include="$(DevEnvDir)\LightSwitch\1.0\Client\Microsoft.LightSwitch.SdkProxy.dll" />

    <IDEReference Include="..\Blank.Common\$(ProjectPath)\Blank.Common.dll" />
    <IDEReference Include="..\Blank.Design\$(ProjectPath)\Blank.Design.dll" />

    <ServerGeneratedReference Include="..\Blank.Common\$(ProjectPath)\Blank.Common.dll"
/>
```

```
    <ServerGeneratedReference Include="..\Blank.Server\$(ProjectPath)\Blank.Server.dll"
/>
  </ItemGroup>
```

After inserting the first line, save the file and reload the project.

Now add the extension to a new or existing LightSwitch Application. After you add the extension in the extensions page, go to the general page and select from the dropdown My LightSwitch Shell, shown in the figure below:



To use the shell extension, you can create some entities and screens and use the following commands:

- Type execute or x <screen name>. A message will display “Link '<screen name>' has been executed.” And the screen will now be displayed.
- Type list or l. You will now see a list of screens.

- You can display another instance of the screen by typing x <same screen name> and now type list and you will see that the shell now has two instances of <screen name> screens.
- To confirm that these screens hold separate data, in the current screen type some text and then execute the following command g <screen name> 1 your will notice that this is a different screen instance. You can return to the other screen by g <screen name> 2.
- To close the second instance of <screen name> screen, execute the k <screen name> 2 command, because the cache for this screen is dirty, that is contains data you are asked if you want to save, discard or cancel the action. Here you can see how the kill command in interacting with the view model of LightSwitch. That is, the heavy lifting work is done for you.

That completes the recipe for a shell extension.

Theme Extension

The theme extension recipe enables you to create a theme that allows a developer to select a color and font palette in LightSwitch.

The ingredients that you are going to need to create a theme extension are the following:

1. Visual Studio 2010 Professional or higher
2. Visual Studio 2010 Service Pack 1
3. Visual Studio 2010 SDK
4. Visual Studio LightSwitch
5. LightSwitch Blank Extension Solution

Once you have all of the ingredients installed on your machine you are ready to create a theme extension.

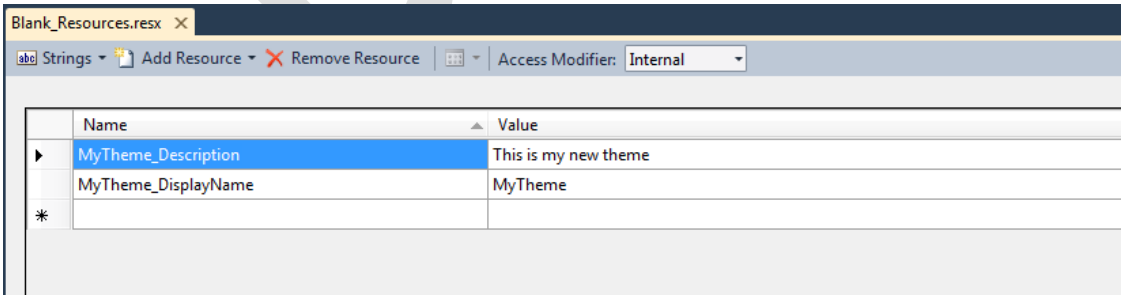
The first step is to get set up with the right environment. This is done by open the Blank Extension solution sample, which is called BlankExtension.sln.

The first step, as when creating a control extension, is to focus on the metadata in the common project.

The first file where you will add code is the Presentation .lsml file, in the metadata folder. This code is used when you need to specify what is displayed in the properties main page to select the theme.

```
<Theme Name="MyTheme">
    <Theme.Attributes>
      <DisplayName Value="$(MyTheme_DisplayName)"/>
      <Description Value="$(MyTheme_Description)"/>
    </Theme.Attributes>
  </Theme>
```

The next file you will modify is the Blank_Resources.resx file, which contains the following name/value pairs for the display name and description of the theme:



Next you need to update the BlankModule.cs class file to now take into account the version number of the theme. Update the code to the following:

```
namespace Blank
{
    internal static class BlankModule
    {
        public const string Name = "Blank";
        private const string ModulePrefix = Name + ":";

        public static class MyTheme
        {
            private const string Name = "MyTheme";
            public const string GlobalName = ModulePrefix + Name;
            public const string Version = "1";
        }
    }
}
```

The next project, in which you will create a folder and some files is the Blank.Client project.

In this project, create a folder called Themes and then another folder underneath it called MyTheme. Again the reason for this is that you may have more than one theme.

In the MyTheme folder create the following file called MyTheme.cs

```
using System;
using Microsoft.LightSwitch.Theming;
using System.ComponentModel.Composition;
using Blank;

namespace Blank.Themes.MyTheme
{
    [Export(typeof(ITheme))]
    [Theme(BlankModule.MyTheme.GlobalName, BlankModule.MyTheme.Version)]
    public class MyTheme : ITheme
    {
        public string Id
        {
            get { return BlankModule.MyTheme.GlobalName; }
        }

        public string Version
        {
            get { return BlankModule.MyTheme.Version; }
        }

        public Uri ColorAndFontScheme
        {
            get { return new
Uri(@"/Blank.Client;component/Themes/MyTheme/MyThemeVisualPalette.xaml",
UriKind.Relative); }
        }
    }
}
```

The next file is the visual palette xaml file called MyThemeVisualPalette.xaml. The xaml file that is listed below contains all the resource keys for the built-in shell. You are free to modify it as required and if you are building a custom shell then these keys can be used or new ones can be defined.

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:internalstyle="clr-
namespace:Microsoft.LightSwitch.Presentation.Framework.Styles.Internal;assembly=Microsoft
.LightSwitch.Client"
    xmlns:sys="clr-namespace:System;assembly=mscorlib">

    <!-- ************************************************************************
-->
    <!-- Control: Text Styles - Can be displayed on ScreenBackground, if not use
FontBackgroundBrush -->
    <!-- States:  FontFamily, FontStyle, FontSize, FontWeight, FontBrush,
FontBackgroundBrush -->

    <!-- Font Style: Normal -->
    <!-- The standard font style -->
    <FontFamily x:Key="NormalFontFamily">Segoe UI, Arial</FontFamily>
    <sys:Double x:Key="NormalFontSize">11</sys:Double>
    <FontWeight x:Key="NormalFontWeight">Normal</FontWeight>
    <FontStyle x:Key="NormalFontStyle">Normal</FontStyle>
    <SolidColorBrush x:Key="NormalFontBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="NormalFontBackgroundBrush" Color="White"/>

    <!-- Font Style: Heading1 -->
    <FontFamily x:Key="Heading1FontFamily">Segoe UI, Arial</FontFamily>
    <sys:Double x:Key="Heading1FontSize">16</sys:Double>
    <FontWeight x:Key="Heading1FontWeight">Bold</FontWeight>
    <FontStyle x:Key="Heading1FontStyle">Normal</FontStyle>
    <SolidColorBrush x:Key="Heading1FontBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="Heading1FontBackgroundBrush" Color="White"/>

    <!-- Font Style: Heading2 -->
    <FontFamily x:Key="Heading2FontFamily">Segoe UI, Arial</FontFamily>
    <sys:Double x:Key="Heading2FontSize">14</sys:Double>
    <FontWeight x:Key="Heading2FontWeight">Normal</FontWeight>
    <FontStyle x:Key="Heading2FontStyle">Normal</FontStyle>
    <SolidColorBrush x:Key="Heading2FontBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="Heading2FontBackgroundBrush" Color="White"/>

    <!-- Font Style: Strong -->
    <FontFamily x:Key="StrongFontFamily">Segoe UI, Arial</FontFamily>
    <sys:Double x:Key="StrongFontSize">11</sys:Double>
    <FontWeight x:Key="StrongFontWeight">Bold</FontWeight>
    <FontStyle x:Key="StrongFontStyle">Normal</FontStyle>
    <SolidColorBrush x:Key="StrongFontBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="StrongFontBackgroundBrush" Color="White"/>

    <!-- Font Style: Emphasis -->
    <FontFamily x:Key="EmphasisFontFamily">Segoe UI, Arial</FontFamily>
    <sys:Double x:Key="EmphasisFontSize">11</sys:Double>
    <FontWeight x:Key="EmphasisFontWeight">Normal</FontWeight>
    <FontStyle x:Key="EmphasisFontStyle">Italic</FontStyle>
```

```
    <SolidColorBrush x:Key="EmphasisFontBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="EmphasisFontBackgroundBrush" Color="White"/>

    <!-- Font Style: Warning -->
    <!-- Generally red -->
    <FontFamily x:Key="WarningFontFamily">Segoe UI, Arial</FontFamily>
    <sys:Double x:Key="WarningFontSize">11</sys:Double>
    <FontWeight x:Key="WarningFontWeight">Bold</FontWeight>
    <FontStyle x:Key="WarningFontStyle">Normal</FontStyle>
    <SolidColorBrush x:Key="WarningFontBrush" Color="#FFDC000C"/>
    <SolidColorBrush x:Key="WarningFontBackgroundBrush" Color="#FFFFC7CE"/>

    <!-- Font Style: Note -->
    <FontFamily x:Key="NoteFontFamily">Segoe UI, Arial</FontFamily>
    <sys:Double x:Key="NoteFontSize">9</sys:Double>
    <FontWeight x:Key="NoteFontWeight">Normal</FontWeight>
    <FontStyle x:Key="NoteFontStyle">Normal</FontStyle>
    <SolidColorBrush x:Key="NoteFontBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="NoteFontBackgroundBrush" Color="White"/>

    <!-- Font Style: Good -->
    <!-- Generally green -->
    <FontFamily x:Key="GoodFontFamily">Segoe UI, Arial</FontFamily>
    <sys:Double x:Key="GoodFontSize">11</sys:Double>
    <FontWeight x:Key="GoodFontWeight">Normal</FontWeight>
    <FontStyle x:Key="GoodFontStyle">Normal</FontStyle>
    <SolidColorBrush x:Key="GoodFontBrush" Color="#FF006100"/>
    <SolidColorBrush x:Key="GoodFontBackgroundBrush" Color="#FFC6EFCE"/>

    <!-- Font Style: Bad -->
    <!-- Generally red -->
    <FontFamily x:Key="BadFontFamily">Segoe UI, Arial</FontFamily>
    <sys:Double x:Key="BadFontSize">11</sys:Double>
    <FontWeight x:Key="BadFontWeight">Normal</FontWeight>
    <FontStyle x:Key="BadFontStyle">Normal</FontStyle>
    <SolidColorBrush x:Key="BadFontBrush" Color="#FFDC000C"/>
    <SolidColorBrush x:Key="BadFontBackgroundBrush" Color="#FFFFC7CE"/>

    <!-- Font Style: Neutral -->
    <!-- Generally yellow/brown -->
    <FontFamily x:Key="NeutralFontFamily">Segoe UI, Arial</FontFamily>
    <sys:Double x:Key="NeutralFontSize">11</sys:Double>
    <FontWeight x:Key="NeutralFontWeight">Normal</FontWeight>
    <FontStyle x:Key="NeutralFontStyle">Normal</FontStyle>
    <SolidColorBrush x:Key="NeutralFontBrush" Color="#FFAD653E"/>
    <SolidColorBrush x:Key="NeutralFontBackgroundBrush" Color="#FFFFEB9C"/>

    <!-- Font Style: ToolTip -->
    <!-- Style used for tooltips -->
    <FontFamily x:Key="ToolTipFontFamily">Segoe UI, Arial</FontFamily>
    <sys:Double x:Key="ToolTipFontSize">11</sys:Double>
    <FontWeight x:Key="ToolTipFontWeight">Normal</FontWeight>
    <FontStyle x:Key="ToolTipFontStyle">Normal</FontStyle>
    <SolidColorBrush x:Key="ToolTipFontBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="ToolTipFontBackgroundBrush" Color="White"/>

    <!-- ********************************************************************************
-->
```

```
    <!-- Control: Screen -->
    <!-- States:  NormalBrush -->

    <!-- ScreenBackground - The background of the screen. -->
    <SolidColorBrush x:Key="ScreenBackgroundBrush" Color="White"/>

    <!-- ScreenLoadingBackground - The background of a control that is loading -->
    <SolidColorBrush x:Key="ScreenLoadingBackgroundBrush" Color="#FFF7F7F7"/>

    <!-- ScreenControlBorder - The border of a control that is not based on another
control already defined -->
    <SolidColorBrush x:Key="ScreenControlBorderBrush" Color="#FFA5ACB5"/>

    <!-- ************************************************************************
-->
    <!-- Control: Button -->
    <!-- States:  NormalBrush, FocusedBrush, MouseOverBrush, DisabledBrush, PressedBrush
-->

    <!-- ButtonBorder -->
    <SolidColorBrush x:Key="ButtonBorderBrush" Color="#FF6593CF"/>
    <SolidColorBrush x:Key="ButtonBorderFocusedBrush" Color="#FFFB9141"/>
    <SolidColorBrush x:Key="ButtonBorderMouseOverBrush" Color="#FFFFCC99"/>
    <SolidColorBrush x:Key="ButtonBorderDisabledBrush" Color="#FFDDDDDD"/>
    <SolidColorBrush x:Key="ButtonBorderPressedBrush" Color="#FFFB9141"/>
    <!-- ButtonInnerBorder -->

    <!-- ButtonBackground -->
    <LinearGradientBrush x:Key="ButtonBackgroundBrush" StartPoint="0,0" EndPoint="1,0">
        <GradientStop Color="#FFB0C7E1" Offset="0"/>
        <GradientStop Color="#FFDBEAFF" Offset="0.5"/>
        <GradientStop Color="#FFB0C7E1" Offset="1"/>
    </LinearGradientBrush>
    <LinearGradientBrush x:Key="ButtonBackgroundFocusedBrush" StartPoint="0,0"
EndPoint="1,0">
        <GradientStop Color="#FFB0C7E1" Offset="0"/>
        <GradientStop Color="#FFDBEAFF" Offset="0.5"/>
        <GradientStop Color="#FFB0C7E1" Offset="1"/>
    </LinearGradientBrush>
    <LinearGradientBrush x:Key="ButtonBackgroundMouseOverBrush" StartPoint="0,0"
EndPoint="1,0">
        <GradientStop Color="#FFFEE19C" Offset="0"/>
        <GradientStop Color="White" Offset="0.5"/>
        <GradientStop Color="#FFFEE19C" Offset="1"/>
    </LinearGradientBrush>
    <SolidColorBrush x:Key="ButtonBackgroundDisabledBrush" Color="#72F7F7F7"/>
    <LinearGradientBrush x:Key="ButtonBackgroundPressedBrush" StartPoint="0,0"
EndPoint="1,0">
        <GradientStop Color="#FFFB9141" Offset="0"/>
        <GradientStop Color="#FFFEE19C" Offset="0.5"/>
        <GradientStop Color="#FFFB9141" Offset="1"/>
    </LinearGradientBrush>

    <!-- ButtonText -->
    <SolidColorBrush x:Key="ButtonTextBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="ButtonTextMouseOverBrush" Color="#FF404040"/>

    <!-- ButtonGlyph - The foreground color of a non-icon glyph such as a down arrow. -->
```

```
    <SolidColorBrush x:Key="ButtonGlyphBrush" Color="White"/>

    <!-- ButtonGlyphBorder - The border around the glyph -->
    <SolidColorBrush x:Key="ButtonGlyphBorderBrush" Color="#FFA5ACB5"/>

    <!-- ButtonFocusBorder - Some themes may display a rectangle around the text content
but inside of the border when the button has keyboard focus. -->
    <SolidColorBrush x:Key="ButtonFocusBorderFocusedBrush" Color="Transparent"/>

    <!-- ButtonBackgroundOverlayEffect - A translucent overlay over the background. -->
    <RadialGradientBrush x:Key="ButtonBackgroundOverlayEffectBrush" Opacity=".3">
        <GradientStop Color="White" Offset="0"/>
        <GradientStop Color="Transparent" Offset="1"/>
    </RadialGradientBrush>

    <!-- ButtonBackgroundWaveEffect - A translucent path over the background. -->
    <LinearGradientBrush x:Key="ButtonBackgroundWaveEffectBrush" StartPoint="0,0"
EndPoint="1,0" Opacity=".4">
        <GradientStop Color="White" Offset="0"/>
        <GradientStop Color="#4CFCFAFA" Offset="1"/>
    </LinearGradientBrush>

    <!-- ********************************************************************************
-->
    <!-- Control: TextBox - The selected state is used for the color of selected text in
the textbox. -->
    <!-- States:  NormalBrush, FocusedBrush, MouseOverBrush, DisabledBrush,
ReadOnlyBrush, SelectedBrush -->

    <!-- TextBoxBorder -->
    <SolidColorBrush x:Key="TextBoxBorderBrush" Color="#FFCCCCCC"/>
    <SolidColorBrush x:Key="TextBoxBorderFocusedBrush" Color="#FFFB9141"/>
    <SolidColorBrush x:Key="TextBoxBorderMouseOverBrush" Color="#FFFFCC99"/>
    <SolidColorBrush x:Key="TextBoxBorderDisabledBrush" Color="#FFDDDDDD"/>
    <SolidColorBrush x:Key="TextBoxBorderReadOnlyBrush" Color="#FFDDDDDD"/>

    <!-- TextBoxBackground -->
    <SolidColorBrush x:Key="TextBoxBackgroundBrush" Color="White"/>
    <SolidColorBrush x:Key="TextBoxBackgroundFocusedBrush" Color="White"/>
    <SolidColorBrush x:Key="TextBoxBackgroundMouseOverBrush" Color="White"/>
    <SolidColorBrush x:Key="TextBoxBackgroundDisabledBrush" Color="#FFF1F1F1"/>
    <SolidColorBrush x:Key="TextBoxBackgroundReadOnlyBrush" Color="#FFF1F1F1"/>
    <SolidColorBrush x:Key="TextBoxBackgroundSelectedBrush" Color="#FF6593CF"/>

    <!-- TextBoxText -->
    <SolidColorBrush x:Key="TextBoxTextBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="TextBoxTextDisabledBrush" Color="#FF707070"/>
    <SolidColorBrush x:Key="TextBoxTextReadOnlyBrush" Color="#FF707070"/>
    <SolidColorBrush x:Key="TextBoxTextSelectedBrush" Color="White"/>

    <!-- TextBoxInfoText - The color of text that appears as a hint or default value.
Usually a lighter color and in italics. -->
    <SolidColorBrush x:Key="TextBoxInfoTextBrush" Color="#FF808080"/>

    <!-- TextBoxCaret - The color of the text entry caret (the blinking thing) -->
    <SolidColorBrush x:Key="TextBoxCaretBrush" Color="Black"/>
```

```
    <!-- ********************************************************************************
-->
    <!-- Control: ComboBox/TextBox Button - In this case the 'Normal' state is displayed
when the TextBox has focus or if another area of the TextBox is MouseOver -->
    <!-- States:  NormalBrush, FocusedBrush, MouseOverBrush, PressedBrush -->

    <!-- TextBoxButtonBackground - A button that appears in the boundary of the textbox
and can get keyboard focus. Like search button in Grid/List -->
    <LinearGradientBrush x:Key="TextBoxButtonBackgroundBrush" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="#FFF7F7F7" Offset="0"/>
        <GradientStop Color="#FFE7E7E7" Offset="1"/>
    </LinearGradientBrush>
    <LinearGradientBrush x:Key="TextBoxButtonBackgroundFocusedBrush" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="#FFF7F7F7" Offset="0"/>
        <GradientStop Color="#FFE7E7E7" Offset="1"/>
    </LinearGradientBrush>
    <LinearGradientBrush x:Key="TextBoxButtonBackgroundMouseOverBrush" StartPoint="0,0"
EndPoint="1,0">
        <GradientStop Color="#FFFEE19C" Offset="0"/>
        <GradientStop Color="White" Offset="0.5"/>
        <GradientStop Color="#FFFEE19C" Offset="1"/>
    </LinearGradientBrush>
    <LinearGradientBrush x:Key="TextBoxButtonBackgroundPressedBrush" StartPoint="0,0"
EndPoint="1,0">
        <GradientStop Color="#FFFB9141" Offset="0"/>
        <GradientStop Color="#FFFEE19C" Offset="0.5"/>
        <GradientStop Color="#FFFB9141" Offset="1"/>
    </LinearGradientBrush>

    <!-- TextBoxButtonBorder -->
    <SolidColorBrush x:Key="TextBoxButtonBorderFocusedBrush" Color="#FFFB9141"/>
    <SolidColorBrush x:Key="TextBoxButtonBorderMouseOverBrush" Color="#FFFB9141"/>
    <SolidColorBrush x:Key="TextBoxButtonBorderPressedBrush" Color="#FFFB9141"/>

    <!-- TextBoxButtonGlyph - Text or glyph color -->
    <SolidColorBrush x:Key="TextBoxButtonGlyphBrush" Color="#FFA5ACB5"/>

    <!-- ********************************************************************************
-->
    <!-- Control: Link -->
    <!-- States:  NormalBrush, FocusedBrush -->

    <!-- LinkText -->
    <SolidColorBrush x:Key="LinkTextBrush" Color="#FF0066CC"/>
    <SolidColorBrush x:Key="LinkTextFocusedBrush" Color="#FF0066CC"/>

    <!-- LinkFocusBorder - Some themes may display a rectangle around the link to
indicate it has keyboard foucs. -->
    <SolidColorBrush x:Key="LinkFocusBorderFocusedBrush" Color="#FF0066CC"/>

    <!-- ********************************************************************************
-->
    <!-- Control: Label -->
    <!-- States:  NormalBrush, DisabledBrush -->

    <!-- LabelText -->
```

```
    <SolidColorBrush x:Key="LabelTextBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="LabelTextDisabledBrush" Color="#FF707070"/>

    <!-- ************************************************************************
-->
    <!-- Control: CheckBox -->
    <!-- States:  NormalBrush, FocusedBrush, MouseOverBrush, DisabledBrush -->

    <!-- CheckBoxBorder - The square that contains the check glyph -->
    <SolidColorBrush x:Key="CheckBoxBorderBrush" Color="#FFA5ACB5"/>
    <SolidColorBrush x:Key="CheckBoxBorderFocusedBrush" Color="#FFFB9141"/>
    <SolidColorBrush x:Key="CheckBoxBorderMouseOverBrush" Color="#FFA5ACB5"/>
    <SolidColorBrush x:Key="CheckBoxBorderDisabledBrush" Color="#FFDDDDDD"/>

    <!-- CheckBoxBackground -->
    <SolidColorBrush x:Key="CheckBoxBackgroundBrush" Color="White"/>
    <SolidColorBrush x:Key="CheckBoxBackgroundDisabledBrush" Color="#FFF1F1F1"/>

    <!-- CheckBoxGlyph - The color of the check glyph -->
    <SolidColorBrush x:Key="CheckBoxGlyphBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="CheckBoxGlyphDisabledBrush" Color="#FF707070"/>
    <!-- Label -->

    <!-- ************************************************************************
-->
    <!-- Control: TabControl -->
    <!-- States:  NormalBrush, FocusedBrush, MouseOverBrush, DisabledBrush, ActiveBrush -
->

    <!-- TabControlBorder - The border around the entire tab control -->
    <SolidColorBrush x:Key="TabControlBorderBrush" Color="#FFA5ACB5"/>
    <SolidColorBrush x:Key="TabControlBorderDisabledBrush" Color="#FFDDDDDD"/>

    <!-- TabControlBackground - The background of the tab panel, this should be the same
or similar to the ScreenBackground color -->
    <SolidColorBrush x:Key="TabControlBackgroundBrush" Color="White"/>

    <!-- TabControlTabBackground - The background of the tab label. -->
    <LinearGradientBrush x:Key="TabControlTabBackgroundBrush" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="White" Offset="0"/>
        <GradientStop Color="#FFB0C7E1" Offset="1"/>
    </LinearGradientBrush>
    <SolidColorBrush x:Key="TabControlTabBackgroundMouseOverBrush" Color="White"/>
    <SolidColorBrush x:Key="TabControlTabBackgroundDisabledBrush" Color="#72F7F7F7"/>
    <SolidColorBrush x:Key="TabControlTabBackgroundActiveBrush" Color="White"/>

    <!-- TabControlTabText - The text of the tab label. -->
    <SolidColorBrush x:Key="TabControlTabTextBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="TabControlTabTextDisabledBrush" Color="#FF707070"/>
    <SolidColorBrush x:Key="TabControlTabTextActiveBrush" Color="#FF404040"/>

    <!-- TabControlTabFocusBorder - Some themes may display a rectangle around the text
content but inside of the border when the tab has keyboard focus. -->
    <SolidColorBrush x:Key="TabControlTabFocusBorderFocusedBrush" Color="#FFFB9141"/>

    <!-- ************************************************************************
-->
```

```
    <!-- Control: Toolbar - A toolbar can be used in any control that needs additional
buttons or a header. Like a List, Grid or future control -->
    <!-- States:  NormalBrush, FocusedBrush, MouseOverBrush, DisabledBrush, PressedBrush
-->

    <!-- ToolbarBackground - The background of the toolbar -->
    <SolidColorBrush x:Key="ToolbarBackgroundBrush" Color="#FFE9EDF1"/>

    <!-- ToolbarSeparator - Any separator lines drawn between buttons or rows -->
    <SolidColorBrush x:Key="ToolbarSeparatorBrush" Color="#FFBEC3CB"/>

    <!-- ToolbarButtonBorder -->
    <SolidColorBrush x:Key="ToolbarButtonBorderBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="ToolbarButtonBorderFocusedBrush" Color="#FFFB9141"/>
    <SolidColorBrush x:Key="ToolbarButtonBorderMouseOverBrush" Color="#FFFFCC99"/>
    <SolidColorBrush x:Key="ToolbarButtonBorderDisabledBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="ToolbarButtonBorderPressedBrush" Color="#FFFB9141"/>

    <!-- ToolbarButtonBackground -->
    <SolidColorBrush x:Key="ToolbarButtonBackgroundBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="ToolbarButtonBackgroundFocusedBrush" Color="Transparent"/>
    <LinearGradientBrush x:Key="ToolbarButtonBackgroundMouseOverBrush" StartPoint="0,0"
EndPoint="1,0">
        <GradientStop Color="#FFFEE19C" Offset="0"/>
        <GradientStop Color="White" Offset="0.5"/>
        <GradientStop Color="#FFFEE19C" Offset="1"/>
    </LinearGradientBrush>
    <SolidColorBrush x:Key="ToolbarButtonBackgroundDisabledBrush" Color="Transparent"/>
    <LinearGradientBrush x:Key="ToolbarButtonBackgroundPressedBrush" StartPoint="0,0"
EndPoint="1,0">
        <GradientStop Color="#FFFB9141" Offset="0"/>
        <GradientStop Color="#FFFEE19C" Offset="0.5"/>
        <GradientStop Color="#FFFB9141" Offset="1"/>
    </LinearGradientBrush>

    <!-- ToolbarButtonText -->
    <SolidColorBrush x:Key="ToolbarButtonTextBrush" Color="Black"/>
    <SolidColorBrush x:Key="ToolbarButtonTextFocusedBrush" Color="Black"/>
    <SolidColorBrush x:Key="ToolbarButtonTextMouseOverBrush" Color="Black"/>
    <SolidColorBrush x:Key="ToolbarButtonTextDisabledBrush" Color="#FF707070"/>
    <SolidColorBrush x:Key="ToolbarButtonTextPressedBrush" Color="Black"/>

    <!-- ************************************************************************
-->
    <!-- Control: ScrollBar -->
    <!-- States:  NormalBrush, MouseOverBrush, DisabledBrush, PressedBrush -->

    <!-- ScrollBarBackground -->
    <SolidColorBrush x:Key="ScrollBarBackgroundBrush" Color="#FFEDEDED"/>
    <SolidColorBrush x:Key="ScrollBarBackgroundDisabledBrush" Color="#FFEDEDED"/>

    <!-- ScrollBarButtonBorder - The directional buttons, up, down, left ,right -->
    <SolidColorBrush x:Key="ScrollBarButtonBorderBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="ScrollBarButtonBorderMouseOverBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="ScrollBarButtonBorderDisabledBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="ScrollBarButtonBorderPressedBrush" Color="Transparent"/>

    <!-- ScrollBarButtonBackground -->
```

```
    <SolidColorBrush x:Key="ScrollBarButtonBackgroundBrush" Color="Transparent"/>
    <LinearGradientBrush x:Key="ScrollBarButtonBackgroundMouseOverBrush" StartPoint="0,0"
EndPoint="1,0">
        <GradientStop Color="White" Offset="0.013"/>
        <GradientStop Color="#FFE9F3FF" Offset="0.369"/>
        <GradientStop Color="#FFCDE0F5" Offset="0.603"/>
        <GradientStop Color="#FFDEEDFF" Offset="1"/>
    </LinearGradientBrush>
    <SolidColorBrush x:Key="ScrollBarButtonBackgroundDisabledBrush" Color="Transparent"/>
    <LinearGradientBrush x:Key="ScrollBarButtonBackgroundPressedBrush" StartPoint="0,0"
EndPoint="1,0">
        <GradientStop Color="#FFE6EFFB" Offset="0.013"/>
        <GradientStop Color="#FFBAD5F5" Offset="0.369"/>
        <GradientStop Color="#FFB6CDE5" Offset="0.603"/>
        <GradientStop Color="#FFD0E5FF" Offset="1"/>
    </LinearGradientBrush>

    <!-- ScrollBarButtonGlyph -->
    <SolidColorBrush x:Key="ScrollBarButtonGlyphBrush" Color="#FF333333"/>
    <SolidColorBrush x:Key="ScrollBarButtonGlyphMouseOverBrush" Color="#FF333333"/>
    <SolidColorBrush x:Key="ScrollBarButtonGlyphDisabledBrush" Color="#FF333333"/>
    <SolidColorBrush x:Key="ScrollBarButtonGlyphPressedBrush" Color="#FF333333"/>

    <!-- ScrollBarThumbBorder - The thumb bar that displays/adjusts the current position.
-->
    <SolidColorBrush x:Key="ScrollBarThumbBorderBrush" Color="#FFA5ACB5"/>
    <SolidColorBrush x:Key="ScrollBarThumbBorderMouseOverBrush" Color="#FFA5ACB5"/>
    <SolidColorBrush x:Key="ScrollBarThumbBorderDisabledBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="ScrollBarThumbBorderPressedBrush" Color="#FFA5ACB5"/>

    <!-- ScrollBarThumbBackground -->
    <SolidColorBrush x:Key="ScrollBarThumbBackgroundBrush" Color="White"/>
    <LinearGradientBrush x:Key="ScrollBarThumbBackgroundMouseOverBrush" StartPoint="0,0"
EndPoint="1,0">
        <GradientStop Color="White" Offset="0.013"/>
        <GradientStop Color="#FFE9F3FF" Offset="0.369"/>
        <GradientStop Color="#FFCDE0F5" Offset="0.603"/>
        <GradientStop Color="#FFDEEDFF" Offset="1"/>
    </LinearGradientBrush>
    <SolidColorBrush x:Key="ScrollBarThumbBackgroundDisabledBrush" Color="Transparent"/>
    <LinearGradientBrush x:Key="ScrollBarThumbBackgroundPressedBrush" StartPoint="0,0"
EndPoint="1,0">
        <GradientStop Color="#FFE6EFFB" Offset="0.013"/>
        <GradientStop Color="#FFBAD5F5" Offset="0.369"/>
        <GradientStop Color="#FFB6CDE5" Offset="0.603"/>
        <GradientStop Color="#FFD0E5FF" Offset="1"/>
    </LinearGradientBrush>

    <!-- ScrollBarThumbGlyph -->
    <SolidColorBrush x:Key="ScrollBarThumbGlyphBrush" Color="#FF333333"/>
    <SolidColorBrush x:Key="ScrollBarThumbGlyphMouseOverBrush" Color="#FF333333"/>
    <SolidColorBrush x:Key="ScrollBarThumbGlyphDisabledBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="ScrollBarThumbGlyphPressedBrush" Color="#FF333333"/>

    <!-- ********************************************************************************
-->
    <!-- Control: List/Grid - The drag state appears when the column header is dragged,
the Border is used for the drop position color -->
```

```
    <!-- States:  NormalBrush, FocusedBrush, MouseOverBrush, PressedBrush, DraggedBrush -
->

    <!-- ListHeaderBackground - The header at the top of a list/grid that shows the
column header and can be clicked on the change sorting -->
    <SolidColorBrush x:Key="ListHeaderBackgroundBrush" Color="#FFFCFCFC"/>
    <LinearGradientBrush x:Key="ListHeaderBackgroundFocusedBrush" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="White" Offset="0"/>
        <GradientStop Color="#FFE9F2F9" Offset="1"/>
    </LinearGradientBrush>
    <LinearGradientBrush x:Key="ListHeaderBackgroundMouseOverBrush" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="White" Offset="0"/>
        <GradientStop Color="#FFE9F2F9" Offset="1"/>
    </LinearGradientBrush>
    <LinearGradientBrush x:Key="ListHeaderBackgroundPressedBrush" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="White" Offset="0"/>
        <GradientStop Color="#FFE9F2F9" Offset="1"/>
    </LinearGradientBrush>
    <SolidColorBrush x:Key="ListHeaderBackgroundDraggedBrush" Color="#66808080"/>

    <!-- ListHeaderBorder - Separator between columns, in grid also used as grid lines --
>
    <SolidColorBrush x:Key="ListHeaderBorderBrush" Color="#FFC2C2C2"/>
    <SolidColorBrush x:Key="ListHeaderBorderFocusedBrush" Color="#FFC2C2C2"/>
    <SolidColorBrush x:Key="ListHeaderBorderMouseOverBrush" Color="#FFC2C2C2"/>
    <SolidColorBrush x:Key="ListHeaderBorderPressedBrush" Color="#FFC2C2C2"/>
    <SolidColorBrush x:Key="ListHeaderBorderDraggedBrush" Color="#FFFB9141"/>

    <!-- ListHeaderText -->
    <SolidColorBrush x:Key="ListHeaderTextBrush" Color="#FF404040"/>

    <!-- ListBorder - The border around the entire List/Grid -->
    <SolidColorBrush x:Key="ListBorderBrush" Color="#FFA5ACB5"/>

    <!-- ************************************************************************
-->
    <!-- Control: Grid -->
    <!-- States:  NormalBrush, MouseOverBrush, SelectedBrush, UnfocusedSelectedBrush,
MouseOverSelectedBrush -->

    <!-- GridRowBackground - The background of a grid row -->
    <SolidColorBrush x:Key="GridRowBackgroundBrush" Color="White"/>
    <SolidColorBrush x:Key="GridRowBackgroundMouseOverBrush" Color="#FFFFEFC6"/>
    <SolidColorBrush x:Key="GridRowBackgroundSelectedBrush" Color="#FFC4DCF7"/>
    <SolidColorBrush x:Key="GridRowBackgroundUnfocusedSelectedBrush" Color="#FFDDDDDD"/>
    <SolidColorBrush x:Key="GridRowBackgroundMouseOverSelectedBrush" Color="#FFFFEFC6"/>

    <!-- GridAlternateRowBackground - The alternating background of a grid row. This is
only used in the Normal state, the colors from GridRowBackground1 are used for other
states. -->
    <SolidColorBrush x:Key="GridAlternateRowBackgroundBrush" Color="#FFE9F2F9"/>
    <!-- GridRowBorder -->

    <!-- GridActiveCellBorder - Border displayed around a selected cell (even when the
grid isn't focused) -->
```

```
    <SolidColorBrush x:Key="GridActiveCellBorderBrush" Color="#FFFB9141"/>

    <!-- GridEditedBackground - The background of a row after the record is edited -->
    <SolidColorBrush x:Key="GridEditedBackgroundBrush" Color="#FFF9F9F9"/>

    <!-- GridAddedBackground - The background of a row after the record is created -->
    <SolidColorBrush x:Key="GridAddedBackgroundBrush" Color="#FFF9F9F9"/>

    <!-- GridDeletedBackground - The background of a row after the record is deleted -->
    <SolidColorBrush x:Key="GridDeletedBackgroundBrush" Color="#FFF9F9F9"/>

    <!-- ********************************************************************************
-->
    <!-- Control: List -->
    <!-- States:  NormalBrush, MouseOverBrush, SelectedBrush, UnfocusedSelectedBrush,
MouseOverSelectedBrush -->

    <!-- ListRowBackground - The background of a list row -->
    <SolidColorBrush x:Key="ListRowBackgroundBrush" Color="White"/>
    <SolidColorBrush x:Key="ListRowBackgroundMouseOverBrush" Color="#FFFFEFC6"/>
    <SolidColorBrush x:Key="ListRowBackgroundSelectedBrush" Color="#FFC4DCF7"/>
    <SolidColorBrush x:Key="ListRowBackgroundUnfocusedSelectedBrush" Color="#FFDDDDDD"/>
    <SolidColorBrush x:Key="ListRowBackgroundMouseOverSelectedBrush" Color="#FFC4DCF7"/>

    <!-- ListRowBorder - The border around the entire row. Not used for our standard
theme. But useful on high contrast. -->
    <SolidColorBrush x:Key="ListRowBorderBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="ListRowBorderMouseOverBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="ListRowBorderSelectedBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="ListRowBorderUnfocusedSelectedBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="ListRowBorderMouseOverSelectedBrush" Color="Transparent"/>

    <!-- ListRowSeparator - The horizontal line displayed between two rows -->
    <SolidColorBrush x:Key="ListRowSeparatorBrush" Color="#FFDBDEE1"/>

    <!-- ********************************************************************************
-->
    <!-- Control: Popup/Menu - Colors used either for popups, like combobox, or menus. --
>
    <!-- States:  NormalBrush, MouseOverBrush, SelectedBrush, MouseOverSelectedBrush,
PressedBrush -->

    <!-- MenuBackground - Background for the entire popup -->
    <SolidColorBrush x:Key="MenuBackgroundBrush" Color="White"/>

    <!-- MenuBorder - Border around the entire popup -->
    <SolidColorBrush x:Key="MenuBorderBrush" Color="#FFA5ACB5"/>

    <!-- MenuItemBackground -->
    <SolidColorBrush x:Key="MenuItemBackgroundMouseOverBrush" Color="#FFFFEFC6"/>
    <SolidColorBrush x:Key="MenuItemBackgroundSelectedBrush" Color="#FFC4DCF7"/>
    <SolidColorBrush x:Key="MenuItemBackgroundMouseOverSelectedBrush" Color="#FFFFEFC6"/>
    <SolidColorBrush x:Key="MenuItemBackgroundPressedBrush" Color="#FFFB9141"/>

    <!-- MenuItemBorder -->
    <SolidColorBrush x:Key="MenuItemBorderMouseOverBrush" Color="#FFFFEFC6"/>
    <SolidColorBrush x:Key="MenuItemBorderSelectedBrush" Color="#FFFB9141"/>
    <SolidColorBrush x:Key="MenuItemBorderMouseOverSelectedBrush" Color="#FFFB9141"/>
```

```
    <SolidColorBrush x:Key="MenuItemBorderPressedBrush" Color="#FFFB9141"/>

    <!-- MenuItemText -->
    <SolidColorBrush x:Key="MenuItemTextBrush" Color="#FF404040"/>

    <!-- ********************************************************************************
-->
    <!-- The following are Shell Theme colors
-->
    <!-- ********************************************************************************
-->

    <!-- ********************************************************************************
-->
    <!-- Control: Ribbon -->
    <!-- States:  NormalBrush, FocusedBrush, MouseOverBrush, PressedBrush, DisabledBrush
-->

    <!-- RibbonBackground - The background of the ribbon menu -->
    <LinearGradientBrush x:Key="RibbonBackgroundBrush" StartPoint="0,0" EndPoint="0,1">
        <GradientStop Color="White" Offset="0"/>
        <GradientStop Color="#FFFBFBFB" Offset="0.769"/>
        <GradientStop Color="#FFF0F1F3" Offset="0.918"/>
        <GradientStop Color="#FFEDEFF2" Offset="1"/>
    </LinearGradientBrush>

    <!-- RibbonBorder - The border between the ribbon and the screen -->
    <SolidColorBrush x:Key="RibbonBorderBrush" Color="#FF8B9097"/>

    <!-- RibbonSeparator - The separator between groups -->
    <LinearGradientBrush x:Key="RibbonSeparatorBrush" StartPoint="0,0" EndPoint="0,1">
        <GradientStop Color="#FFF5F6F7" Offset="0"/>
        <GradientStop Color="#FFB0B6BC" Offset="1"/>
    </LinearGradientBrush>

    <!-- RibbonSeparator2 - The padding between the separator and the backgorund -->
    <SolidColorBrush x:Key="RibbonSeparator2Brush" Color="#FFFBFBFB"/>

    <!-- RibbonGroupText - The text labeling the ribbon group -->
    <SolidColorBrush x:Key="RibbonGroupTextBrush" Color="#FF686B6E"/>

    <!-- RibbonButtonBorder -->
    <SolidColorBrush x:Key="RibbonButtonBorderBrush" Color="Transparent"/>
    <LinearGradientBrush x:Key="RibbonButtonBorderFocusedBrush" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="#FFF1CA58" Offset="0"/>
        <GradientStop Color="#FFF4D649" Offset="1"/>
    </LinearGradientBrush>
    <LinearGradientBrush x:Key="RibbonButtonBorderMouseOverBrush" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="#FFF1CA58" Offset="0"/>
        <GradientStop Color="#FFF4D649" Offset="1"/>
    </LinearGradientBrush>
    <LinearGradientBrush x:Key="RibbonButtonBorderPressedBrush" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="#FFC28A30" Offset="0"/>
        <GradientStop Color="#FFC2A44D" Offset="1"/>
    </LinearGradientBrush>
```

```
    <!-- RibbonButtonInnerBorder -->
    <SolidColorBrush x:Key="RibbonButtonInnerBorderBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="RibbonButtonInnerBorderMouseOverBrush" Color="#FFFDF6DE"/>
    <SolidColorBrush x:Key="RibbonButtonInnerBorderPressedBrush" Color="Transparent"/>

    <!-- RibbonButtonBackground -->
    <SolidColorBrush x:Key="RibbonButtonBackgroundBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="RibbonButtonBackgroundFocusedBrush" Color="Transparent"/>
    <RadialGradientBrush x:Key="RibbonButtonBackgroundMouseOverBrush" RadiusX="4"
RadiusY="1.1" Opacity="0.9">
        <RadialGradientBrush.RelativeTransform>
            <CompositeTransform CenterY="0.5" CenterX="0.5" TranslateY="0.5"/>
        </RadialGradientBrush.RelativeTransform>
        <GradientStop Color="White" Offset="0"/>
        <GradientStop Color="#FFFCE38A" Offset="0.172"/>
        <GradientStop Color="#FFFDE389" Offset="0.424"/>
        <GradientStop Color="#FFFDE389" Offset="0.756"/>
        <GradientStop Color="#FFFDEEB3" Offset="0.872"/>
    </RadialGradientBrush>
    <RadialGradientBrush x:Key="RibbonButtonBackgroundPressedBrush" RadiusX="4"
RadiusY="1.1">
        <RadialGradientBrush.RelativeTransform>
            <CompositeTransform CenterY="0.5" CenterX="0.5" TranslateY="0.5"/>
        </RadialGradientBrush.RelativeTransform>
        <GradientStop Color="#FFFFF480" Offset="0"/>
        <GradientStop Color="#FFFFDB6D" Offset="0.28"/>
        <GradientStop Color="#FFFFDB6D" Offset="0.76"/>
        <GradientStop Color="#FFFFE575" Offset="0.968"/>
    </RadialGradientBrush>

    <!-- RibbonButtonText -->
    <SolidColorBrush x:Key="RibbonButtonTextBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="RibbonButtonTextFocusedBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="RibbonButtonTextMouseOverBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="RibbonButtonTextPressedBrush" Color="#FF404040"/>

    <!-- RibbonDropShadow -->
    <LinearGradientBrush x:Key="RibbonDropShadowBrush" StartPoint="0,0" EndPoint="0,1">
        <GradientStop Color="#FFC6C9CD" Offset="0"/>
        <GradientStop Color="#FFD8DCE0" Offset="0.528"/>
        <GradientStop Color="#FFE9EDF1" Offset="1"/>
    </LinearGradientBrush>

    <!-- ************************************************************************
-->
    <!-- Control: Navigation Menu -->
    <!-- States:  NormalBrush, FocusedBrush, MouseOverBrush, PressedBrush -->

    <!-- NavShellBackground - The color of the area "behind" the screen tab -->
    <SolidColorBrush x:Key="NavShellBackgroundBrush" Color="#FFEBEDF0"/>

    <!-- NavBackground - The background of the navigation menu -->
    <SolidColorBrush x:Key="NavBackgroundBrush" Color="#FFEBEDF0"/>

    <!-- NavGroupBackground - The buttons that represent navigation groups -->
    <SolidColorBrush x:Key="NavGroupBackgroundBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="NavGroupBackgroundFocusedBrush" Color="Transparent"/>
```

```
    <LinearGradientBrush x:Key="NavGroupBackgroundMouseOverBrush" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="#FFDEE4EB" Offset="0"/>
        <GradientStop Color="#FFD6DCE4" Offset="0.438"/>
        <GradientStop Color="#FFD4DAE2" Offset="0.528"/>
        <GradientStop Color="#FFEFF2F5" Offset="1"/>
    </LinearGradientBrush>

    <!-- NavGroupBorder -->
    <SolidColorBrush x:Key="NavGroupBorderBrush" Color="#FFC5CED8"/>
    <LinearGradientBrush x:Key="NavGroupBorderFocusedBrush" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="#FFF1CA58" Offset="0"/>
        <GradientStop Color="#FFF4D649" Offset="1"/>
    </LinearGradientBrush>
    <LinearGradientBrush x:Key="NavGroupBorderMouseOverBrush" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="#FFF1CA58" Offset="0"/>
        <GradientStop Color="#FFF4D649" Offset="1"/>
    </LinearGradientBrush>

    <!-- NavGroupInnerBorder - An inner glow on a selected nav group -->
    <SolidColorBrush x:Key="NavGroupInnerBorderBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="NavGroupInnerBorderFocusedBrush" Color="#80FFFFFF"/>
    <SolidColorBrush x:Key="NavGroupInnerBorderMouseOverBrush" Color="Transparent"/>

    <!-- NavGroupText -->
    <SolidColorBrush x:Key="NavGroupTextBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="NavGroupTextFocusedBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="NavGroupTextMouseOverBrush" Color="#FF404040"/>

    <!-- NavItemBackground - The buttons that represent navigation items. Clicking opens
a screen. -->
    <SolidColorBrush x:Key="NavItemBackgroundBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="NavItemBackgroundFocusedBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="NavItemBackgroundMouseOverBrush" Color="#FFF4E7AE"/>
    <SolidColorBrush x:Key="NavItemBackgroundPressedBrush" Color="#FFF4E7AE"/>

    <!-- NavItemBorder -->
    <SolidColorBrush x:Key="NavItemBorderBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="NavItemBorderFocusedBrush" Color="#FFFB9141"/>
    <SolidColorBrush x:Key="NavItemBorderMouseOverBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="NavItemBorderPressedBrush" Color="#FFFB9141"/>

    <!-- NavItemText -->
    <SolidColorBrush x:Key="NavItemTextBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="NavItemTextFocusedBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="NavItemTextMouseOverBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="NavItemTextPressedBrush" Color="#FF404040"/>

    <!-- ************************************************************************
-->
    <!-- Control: ScreenTab -->
    <!-- States:  NormalBrush, MouseOverBrush, ActiveBrush -->

    <!-- ScreenTabBorder - The border around the entire screen tab -->
    <SolidColorBrush x:Key="ScreenTabBorderBrush" Color="#FF8B9097"/>
```

```
    <!-- ScreenTabBackground - The background of just the tab header -->
    <LinearGradientBrush x:Key="ScreenTabBackgroundBrush" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="#FFF6F7F8" Offset="0"/>
        <GradientStop Color="#FFDDE0E4" Offset="1"/>
    </LinearGradientBrush>
    <LinearGradientBrush x:Key="ScreenTabBackgroundMouseOverBrush" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="#FFF6F7F8" Offset="0"/>
        <GradientStop Color="#FFF7E7E3" Offset="1"/>
    </LinearGradientBrush>

    <!-- ScreenTabText - The text of the tab header -->
    <SolidColorBrush x:Key="ScreenTabTextBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="ScreenTabTextMouseOverBrush" Color="#FF404040"/>
    <SolidColorBrush x:Key="ScreenTabTextActiveBrush" Color="#FF404040"/>

    <!-- ************************************************************************
-->
    <!-- Control: Shell Component -->
    <!-- States:  NormalBrush, FocusedBrush, MouseOverBrush, DisabledBrush, PressedBrush
-->

    <!-- ShellGlyphButtonBorder - The small button used to expand the nav menu/the
ribbon/close a tab/close modal window -->
    <SolidColorBrush x:Key="ShellGlyphButtonBorderBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="ShellGlyphButtonBorderFocusedBrush" Color="#FFFB9141"/>
    <SolidColorBrush x:Key="ShellGlyphButtonBorderMouseOverBrush" Color="#FFFFCC99"/>
    <SolidColorBrush x:Key="ShellGlyphButtonBorderDisabledBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="ShellGlyphButtonBorderPressedBrush" Color="#FFFB9141"/>

    <!-- ShellGlyphButtonBackground -->
    <SolidColorBrush x:Key="ShellGlyphButtonBackgroundBrush" Color="Transparent"/>
    <SolidColorBrush x:Key="ShellGlyphButtonBackgroundFocusedBrush" Color="Transparent"/>
    <LinearGradientBrush x:Key="ShellGlyphButtonBackgroundMouseOverBrush"
StartPoint="0,0" EndPoint="1,0">
        <GradientStop Color="#FFFEE19C" Offset="0"/>
        <GradientStop Color="White" Offset="0.5"/>
        <GradientStop Color="#FFFEE19C" Offset="1"/>
    </LinearGradientBrush>
    <SolidColorBrush x:Key="ShellGlyphButtonBackgroundDisabledBrush"
Color="Transparent"/>
    <LinearGradientBrush x:Key="ShellGlyphButtonBackgroundPressedBrush" StartPoint="0,0"
EndPoint="1,0">
        <GradientStop Color="#FFFB9141" Offset="0"/>
        <GradientStop Color="#FFFEE19C" Offset="0.5"/>
        <GradientStop Color="#FFFB9141" Offset="1"/>
    </LinearGradientBrush>

    <!-- ShellGlyphButtonArrowGlyph - The color of the expand/collapse arrow -->
    <SolidColorBrush x:Key="ShellGlyphButtonArrowGlyphBrush" Color="#FF757677"/>

    <!-- ShellGlyphButtonCloseGlyph - The color of the screen close glyph -->
    <SolidColorBrush x:Key="ShellGlyphButtonCloseGlyphBrush" Color="Black"/>

    <!-- ************************************************************************
-->
```

```
    <!-- Control: Validation Popup - The floating message that appears over a control
when an error exists in the control. -->
    <!-- States:  NormalBrush -->

    <!-- ValidationPopupBackground -->
    <SolidColorBrush x:Key="ValidationPopupBackgroundBrush" Color="White"/>

    <!-- ValidationPopupText - The text color -->
    <SolidColorBrush x:Key="ValidationPopupTextBrush" Color="Black"/>

    <!-- ValidationPopupErrorBorder - The border used when the message is an error -->
    <SolidColorBrush x:Key="ValidationPopupErrorBorderBrush" Color="#FFDC000C"/>

    <!-- ValidationPopupMessageBorder - The border used when the message is an
informative message -->
    <SolidColorBrush x:Key="ValidationPopupMessageBorderBrush" Color="Black"/>

    <!-- ************************************************************************
-->
    <!-- Control: Screen Tab Validation - The control that appears at the top of a screen
when validation errors occur. This has a header and an popup list of errors that can be
clicked on. -->
    <!-- States:  NormalBrush, FocusedBrush, MouseOverBrush, PressedBrush -->

    <!-- TabValidationBackground - The background of the validation header. -->
    <LinearGradientBrush x:Key="TabValidationBackgroundBrush" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="#FFFFF6BD" Offset="0"/>
        <GradientStop Color="#FFFFEC82" Offset="1"/>
    </LinearGradientBrush>
    <LinearGradientBrush x:Key="TabValidationBackgroundMouseOverBrush" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="#FFFEF8D8" Offset="0"/>
        <GradientStop Color="#FFFDEFA8" Offset="1"/>
    </LinearGradientBrush>
    <LinearGradientBrush x:Key="TabValidationBackgroundPressedBrush" StartPoint="0,0"
EndPoint="0,1">
        <GradientStop Color="#FFFEF8D8" Offset="0"/>
        <GradientStop Color="#FFFDEFA8" Offset="1"/>
    </LinearGradientBrush>

    <!-- TabValidationBorder -->
    <SolidColorBrush x:Key="TabValidationBorderBrush" Color="#FFB4A555"/>
    <SolidColorBrush x:Key="TabValidationBorderFocusedBrush" Color="#FFFB9141"/>
    <SolidColorBrush x:Key="TabValidationBorderMouseOverBrush" Color="#FFB4A555"/>
    <SolidColorBrush x:Key="TabValidationBorderPressedBrush" Color="#FFB4A555"/>

    <!-- TabValidationGlyph - The glyph that expands the popup -->
    <SolidColorBrush x:Key="TabValidationGlyphBrush" Color="#FF8E813E"/>

    <!-- TabValidationText -->
    <SolidColorBrush x:Key="TabValidationTextBrush" Color="#FF404040"/>

    <!-- TabValidationItemBackground - The background of an individual validation error -
->
    <SolidColorBrush x:Key="TabValidationItemBackgroundMouseOverBrush"
Color="#FFFEE19C"/>
```

```
    <LinearGradientBrush x:Key="TabValidationItemBackgroundPressedBrush" StartPoint="0,0"
EndPoint="1,0">
        <GradientStop Color="#FFFB9141" Offset="0"/>
        <GradientStop Color="#FFFEE19C" Offset="0.5"/>
        <GradientStop Color="#FFFB9141" Offset="1"/>
    </LinearGradientBrush>

    <!-- TabValidationItemBorder -->
    <SolidColorBrush x:Key="TabValidationItemBorderFocusedBrush" Color="#FFFB9141"/>
    <SolidColorBrush x:Key="TabValidationItemBorderPressedBrush" Color="#FFFB9141"/>

    <!-- TabValidationItemText -->
    <SolidColorBrush x:Key="TabValidationItemTextBrush" Color="#FF404040"/>

    <!-- ********************************************************************************
-->
    <!-- Control: Modal Window -->
    <!-- States:  NormalBrush -->

    <!-- WindowTitleBackground - The background of the title bar -->
    <SolidColorBrush x:Key="WindowTitleBackgroundBrush" Color="#FFF3F7FD"/>

    <!-- WindowTitleText - The text of the title bar -->
    <SolidColorBrush x:Key="WindowTitleTextBrush" Color="#FF404040"/>

    <!-- WindowBorder - The border around the window -->
    <SolidColorBrush x:Key="WindowBorderBrush" Color="#FFA5ACB5"/>

    <!-- WindowBackground - The background of the body of the dialog. In most cases if UI
is diplayed on the background it is in a control that uses ScreenBackground. However
message box dialogs display text on this color. -->
    <SolidColorBrush x:Key="WindowBackgroundBrush" Color="#FFF3F7FD"/>
</ResourceDictionary>
```

Do a search for NavShellBackgroundBrush and change the color to Red. Now build and install the theme and see how the color has changed from Blue to Red. This is the end of the Theme extension recipe.

Custom Data Source Extension

The custom data source recipe enables you to create a RIA Service to bring in external data into your LightSwitch application.

The ingredients that you are going to need to create a custom data source are the following:

1. Visual Studio 2010 Professional or higher
2. Visual Studio 2010 Service Pack 1
3. Visual Studio 2010 SDK
4. Visual Studio LightSwitch
5. LightSwitch Blank Extension Solution

Once you have all of the ingredients installed on your machine you are ready to create a custom data source extension.

The first step is to get set up with the right environment. This is done by open the Blank Extension solution sample, which is called BlankExtension.sln.

When creating a custom data source, you are essentially creating a WCF RIA service with the necessary support for the RIA service to be consumed by LightSwitch as an extension.

For the purposes of this recipe, you will see how a custom xml data source is created that creates an XML file based on a given location that is specified by the developer in LightSwitch when adding the custom data source.

Let's look at the Common project. Both the BlankPresentation.lsml and BlankModule.lsml files do not need to be modified unless you wish to give your extension a different namespace than that what the template is using. This is done in the BlankModule.lsml file. The reason we do not need to modify any lsml files is that there is no client presentation when creating a custom data source extension, it is all server side.

Moving on to the Server project, you will start building out our custom data source.

In the Server project you will need to add the following assembly references:

C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.0\System.ComponentModel.DataAnnotations.dll

C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.0\System.Data.Entity.dll

C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.0\System.Runtime.Serialization.dll

C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.0\System.Security.dll

C:\Program Files (x86)\Microsoft SDKs\RIA Services\v1.0\Libraries\Server\System.ServiceModel.DomainServices.Hosting.dll

C:\Program Files (x86)\Microsoft SDKs\RIA Services\v1.0\Libraries\Server\System.ServiceModel.DomainServices.Server.dll

C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.0\System.Web.dll

C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\.NETFramework\v4.0\System.configuration.dll

These assemblies provide the access to WCF RIA Services as well as web configuration assemblies provide the capability to provide connection string related information, which in this case is the local file path. This also could be server, username, password credentials based on the type of data source that you are trying to access.

The first file to create is the Service class. In this case you will create a ProductsService.cs class file in the Server project., It should contain the following code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.ServiceModel.DomainServices.Hosting;
using System.ServiceModel.DomainServices.Server;
using System.Xml.Linq;
using System.Configuration;
using System.Web.Configuration;

namespace Blank
{

    //The description attribute will be displayed in the LightSwitch data attach wizard.
    [Description("Please provide the path to the XML file")]
    public class ProductsService : DomainService
    {
        private XElement _db;
        private string _filePath;

        public override void Initialize(DomainServiceContext context)
        {
            base.Initialize(context);

            //get the file path from the config
            if
(WebConfigurationManager.ConnectionStrings[typeof(ProductsService).FullName] != null)
```

```
                _filePath =
WebConfigurationManager.ConnectionStrings[typeof(ProductsService).FullName].ConnectionStr
ing;

            if (String.IsNullOrWhiteSpace(_filePath))
            {
                throw new Exception("The filepath must be provided in the web.config in
the connection strings section under " + typeof(ProductsService).FullName);
            }
            else
            {
                if (!System.IO.File.Exists(_filePath))
                {
                    XElement x = new XElement("DataRoot");
                    x.Save(_filePath);
                }

                _db = XElement.Load(_filePath);

                //verify file
                if (_db.Name != "DataRoot")
                    throw new Exception("Corrupt file.");

                //add a product categories node if one does not exist
                if (_db.Element("ProductCategories") == null)
                    _db.Add(new XElement("ProductCategories"));

                //add a products node if one does not exist
                if (_db.Element("Products") == null)
                    _db.Add(new XElement("Products"));
            }
        }

        public override bool Submit(ChangeSet changeSet)
        {
            ChangeSet c = new ChangeSet(changeSet.ChangeSetEntries.OrderBy(entry =>
entry.Entity, new ProductEntitiesComparer()));

            Boolean baseResult = base.Submit(changeSet);
            if (baseResult)
            {
                _db.Save(_filePath);
                return true;
            }
            else
                return false;
        }


        #region Queries

        protected override int Count<T>(IQueryable<T> query)
        {
            return query.Count();
        }

        [Query(IsDefault = true)]
        public IQueryable<Product> GetProducts()
```

```
        {
            List<Product> products = new List<Product>();
            foreach (XElement pElem in _db.Descendants("Product"))
            {
                products.Add(GetProduct(pElem));
            }
            return products.AsQueryable();
        }

        [Query(IsDefault = true)]
        public IQueryable<ProductCategory> GetProductCategories()
        {
            List<ProductCategory> categories = new List<ProductCategory>();
            foreach (XElement catElem in _db.Descendants("ProductCategory"))
            {
                categories.Add(GetProductCategory(catElem));
            }

            return categories.AsQueryable();
        }
        #endregion

        #region CustomQueries

        //All query parameters need to be nullable.
        public IQueryable<Product> GetProductsByCategory(Nullable<Guid> categoryID)
        {
            List<Product> products = new List<Product>();

            //We will return no products if categoryID as null
            if (categoryID.HasValue)
            {
                foreach (XElement pElem in _db.Descendants("Product"))
                {
                    Product p = GetProduct(pElem);
                    if (p.CategoryID == categoryID.Value)
                    {
                        products.Add(p);
                    }
                }
            }
            return products.AsQueryable();
        }

        //Query that returns a single item
        //Need to mark queries that return as single item as composable = false
        [Query(IsComposable = false)]
        public Product GetProductByID(Nullable<Guid> productID)
        {
            Product product = null;

            //Only return a result if we are passed a value
            if (productID.HasValue)
            {
                XElement pElem = (from XElement p in _db.Descendants("Product")
                                  where p.Element("ProductID").Value ==
productID.Value.ToString()
                                  select p).FirstOrDefault();
```

```
            if (pElem != null)
            {
                product = GetProduct(pElem);
            }
        }
        return product;

    }
    #endregion


    #region Category Update/Delete/Insert Methods
    public void InsertProductCategory(ProductCategory pc)
    {
        try
        {
            pc.CategoryID = Guid.NewGuid();
            XElement catElem = GetProductCategoryElem(pc);

            //update the category ID on any related product entities
            foreach (ChangeSetEntry e in ChangeSet.ChangeSetEntries)
            {
                if (e.Entity is Product)
                {
                    if (((Product)e.Entity).Category == pc)
                    {
                        ((Product)e.Entity).CategoryID = pc.CategoryID;
                    }
                }
            }


            //update the xml doc
            _db.Element("ProductCategories").Add(catElem);
        }
        catch (Exception ex)
        {
            throw new Exception("Error inserting ProductCategory " + pc.CategoryName,
ex);
        }
    }

    public void UpdateProductCategory(ProductCategory pc)
    {
        try
        {
            //get existing item from the XML db
            XElement storeCatElem =
                (from c in _db.Descendants("ProductCategory")
                 where c.Element("CategoryID").Value == pc.CategoryID.ToString()
                 select c).FirstOrDefault();

            if (storeCatElem == null)
            {
                //Category does not exist.  Need to indicate item has already been
deleted
                ChangeSetEntry entry =
                    (from e in ChangeSet.ChangeSetEntries
```

```
                        where e.Entity == pc
                        select e).First();
                    entry.IsDeleteConflict = true;
                }
                else
                {
                    ProductCategory storeCategory = GetProductCategory(storeCatElem);

                    //find entry in the changeset to compare original values
                    ChangeSetEntry entry =
                        (from e in ChangeSet.ChangeSetEntries
                         where e.Entity == pc
                         select e).First();

                    //list of conflicting fields
                    List<String> conflictMembers = new List<String>();

                    if (storeCategory.CategoryName !=
((ProductCategory)entry.OriginalEntity).CategoryName)
                        conflictMembers.Add("CategoryName");
                    if (storeCategory.Description !=
((ProductCategory)entry.OriginalEntity).Description)
                        conflictMembers.Add("Description");

                    //set conflict members
                    entry.ConflictMembers = conflictMembers;
                    entry.StoreEntity = storeCategory;

                    if (conflictMembers.Count < 1)
                    {
                        //update the xml _db
                        storeCatElem.ReplaceWith(GetProductCategoryElem(pc));
                    }
                }
            }
            catch (Exception ex)
            {
                throw new Exception("Error updating Category " + pc.CategoryID.ToString()
+ ":" + pc.CategoryName, ex);
            }
        }

        public void DeleteProductCategory(ProductCategory pc)
        {
            try
            {
                XElement storeCatElem =
                    (from c in _db.Descendants("ProductCategory")
                     where c.Element("CategoryID").Value == pc.CategoryID.ToString()
                     select c).FirstOrDefault();

                if (storeCatElem == null)
                {
                    //category does not exist.  Need to indicate item has already been
deleted
                    ChangeSetEntry entry =
                        (from e in ChangeSet.ChangeSetEntries
                         where e.Entity == pc
```

```
                        select e).First();
                    entry.IsDeleteConflict = true;
                }
                else
                {
                    storeCatElem.Remove();
                }
            }
            catch (Exception ex)
            {
                throw new Exception("Error deleting Category " + pc.CategoryID.ToString()
+ ":" + pc.CategoryName, ex);
            }
        }

        #endregion


        #region Product Update/Delete/Insert Methods
        public void InsertProduct(Product p)
        {
            try
            {
                p.ProductID = Guid.NewGuid();
                XElement productElem = GetProductElem(p);
                _db.Element("Products").Add(productElem);
            }
            catch (Exception ex)
            {
                throw new Exception("Error inserting Product " + p.ProductName, ex);
            }
        }

        public void UpdateProduct(Product p)
        {
            try
            {
                XElement storeProductElem =
                    (from c in _db.Descendants("Product")
                     where c.Element("ProductID").Value == p.ProductID.ToString()
                     select c).FirstOrDefault();

                if (storeProductElem == null)
                {
                    //product does not exist.  Need to indicate item has already been
deleted
                    ChangeSetEntry entry =
                        (from e in ChangeSet.ChangeSetEntries
                         where e.Entity == p
                         select e).First();
                    entry.IsDeleteConflict = true;
                }
                else
                {
                    Product storeProduct = GetProduct(storeProductElem);

                    //find entry in the changeset to compare original values
                    ChangeSetEntry entry =
```

```
                        (from e in ChangeSet.ChangeSetEntries
                         where e.Entity == p
                         select e).First();

                    //list of conflicting fields
                    List<String> conflictMembers = new List<String>();

                    if (storeProduct.ProductName !=
((Product)entry.OriginalEntity).ProductName)
                        conflictMembers.Add("ProductName");
                    if (storeProduct.CategoryID !=
((Product)entry.OriginalEntity).CategoryID)
                        conflictMembers.Add("CategoryID");
                    if (storeProduct.QuantityPerUnit !=
((Product)entry.OriginalEntity).QuantityPerUnit)
                        conflictMembers.Add("QuantityPerUnit");
                    if (storeProduct.UnitPrice !=
((Product)entry.OriginalEntity).UnitPrice)
                        conflictMembers.Add("UnitPrice");
                    if (storeProduct.UnitsInStock !=
((Product)entry.OriginalEntity).UnitsInStock)
                        conflictMembers.Add("UnitsInStock");
                    if (storeProduct.ReorderLevel !=
((Product)entry.OriginalEntity).ReorderLevel)
                        conflictMembers.Add("ReorderLevel");
                    if (storeProduct.Discontinued !=
((Product)entry.OriginalEntity).Discontinued)
                        conflictMembers.Add("Discontinued");

                    //set conflict members
                    entry.ConflictMembers = conflictMembers;
                    entry.StoreEntity = storeProduct;

                    if (conflictMembers.Count < 1)
                    {
                        //update the xml _db
                        storeProductElem.ReplaceWith(GetProductElem(p));
                    }
                }
            }
            catch (Exception ex)
            {
                throw new Exception("Error updating Product " + p.ProductID.ToString() +
":" + p.ProductName, ex);
            }
        }

        public void DeleteProduct(Product p)
        {
            try
            {
                XElement storeProductElem =
                    (from c in _db.Descendants("Product")
                     where c.Element("ProductID").Value == p.ProductID.ToString()
                     select c).FirstOrDefault();

                if (storeProductElem == null)
                {
```

```
                    //product does not exist.  Need to indicate item has already been
deleted
                    ChangeSetEntry entry =
                        (from e in ChangeSet.ChangeSetEntries
                         where e.Entity == p
                         select e).First();
                    entry.IsDeleteConflict = true;
                }
                else
                {
                    //remove it
                    storeProductElem.Remove();
                }
            }
            catch (Exception ex)
            {
                throw new Exception("Error deleting Product " + p.ProductID.ToString() +
":" + p.ProductName, ex);
            }
        }

        #endregion

        #region HelperFunctions
        private XElement GetProductElem(Product p)
        {
            XElement productElem = new XElement("Product",
                new XElement("ProductID", new object[] { new XAttribute("DataType",
"Guid"), p.ProductID }),
                new XElement("ProductName", new object[] { new XAttribute("DataType",
"String"), p.ProductName }),
                new XElement("CategoryID", new object[] { new XAttribute("DataType",
"Guid"), p.CategoryID }),
                new XElement("QuantityPerUnit", new object[] { new XAttribute("DataType",
"String"), p.QuantityPerUnit }),
                new XElement("UnitPrice", new object[] { new XAttribute("DataType",
"Decimal"), p.UnitPrice }),
                new XElement("UnitsInStock", new object[] { new XAttribute("DataType",
"Int32"), p.UnitsInStock }),
                new XElement("ReorderLevel", new object[] { new XAttribute("DataType",
"Int32?"), p.ReorderLevel }),
                new XElement("Discontinued", new object[] { new XAttribute("DataType",
"Boolean"), p.Discontinued })
                );
            return productElem;
        }

        private Product GetProduct(XElement pElem)
        {
            Product p = new Product();

            p.ProductID = new Guid(pElem.Element("ProductID").Value);
            p.ProductName = pElem.Element("ProductName").Value;
            p.CategoryID = new Guid(pElem.Element("CategoryID").Value);
            p.QuantityPerUnit = pElem.Element("QuantityPerUnit").Value;
            p.UnitPrice = decimal.Parse(pElem.Element("UnitPrice").Value);
            p.UnitsInStock = Int32.Parse(pElem.Element("UnitsInStock").Value);
            if (!String.IsNullOrWhiteSpace(pElem.Element("ReorderLevel").Value))
```

```
            {
                p.ReorderLevel = Int32.Parse(pElem.Element("ReorderLevel").Value);
            }
            p.Discontinued = Boolean.Parse(pElem.Element("Discontinued").Value);

            return p;
        }

        private ProductCategory GetProductCategory(XElement catElem)
        {
            ProductCategory pc = new ProductCategory();
            pc.CategoryID = new Guid(catElem.Element("CategoryID").Value);
            pc.CategoryName = catElem.Element("CategoryName").Value;
            pc.Description = catElem.Element("Description").Value;

            return pc;
        }

        private XElement GetProductCategoryElem(ProductCategory pc)
        {
            XElement catElem = new XElement("ProductCategory",
               new XElement("CategoryID", new object[] { new XAttribute("DataType",
"Guid"), pc.CategoryID }),
               new XElement("CategoryName", new object[] { new XAttribute("DataType",
"String"), pc.CategoryName }),
               new XElement("Description", new object[] { new XAttribute("DataType",
"String"), pc.Description })
               );
            return catElem;
        }
        #endregion

    }

    public class ProductEntitiesComparer : Comparer<object>
    {
        public override int Compare(object x, object y)
        {
            if (x is Product && y is ProductCategory)
                return 1;
            else if (x is ProductCategory && y is Product)
                return -1;
            else
                return 0;
        }
    }

}
```

The code above not only has service connection related code but also two queries, one to get product by ID and another to get product by category. These queries can be used in a LightSwitch application as well.

The next two class files are the representation of the entities themselves: Product.cs and ProductCategory.cs.

Create these two class files in the Server project and add the following code respectively:

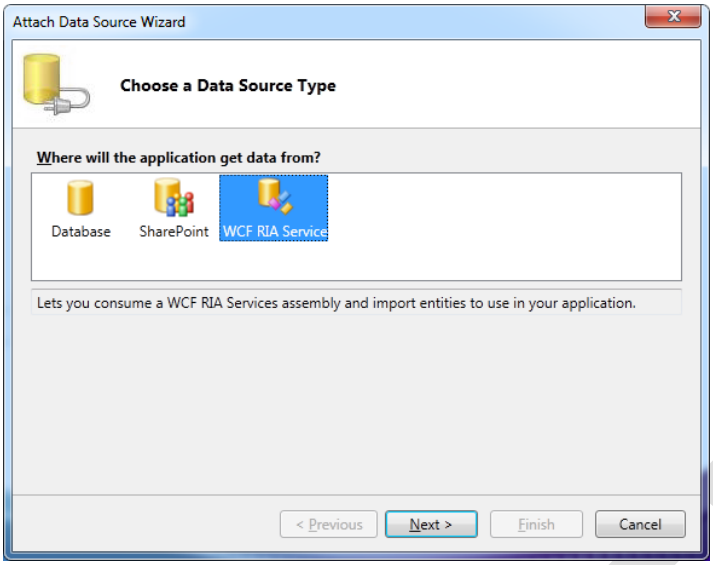
Product.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;


namespace Blank
{
    public class Product
    {
        [Key()]
        [ReadOnly(true)]
        [Display(Name = "Product ID")]
        [ScaffoldColumn(false)]
        public Guid ProductID { get; set; }

        [Required()]
        [Display(Name = "Product Name")]
        public string ProductName { get; set; }

        [Required()]
        [Display(Name = "Category ID")]
        public Guid CategoryID { get; set; }

        [Display(Name = "Quantity Per Unit")]
        public string QuantityPerUnit { get; set; }

        [Range(0, double.MaxValue, ErrorMessage = "The specified price must be greater
than zero.")]
        [Display(Name = "Unit Price")]
        public Decimal UnitPrice { get; set; }

        [Display(Name = "Units In Stock")]
        [Range(0, Int32.MaxValue, ErrorMessage = "Cannot have a negative quantity of
products.")]
        public Int32 UnitsInStock { get; set; }

        [Display(Name = "Reorder Level")]
        public Nullable<Int32> ReorderLevel { get; set; }

        [Display(Name = "Discontinued")]
        public Boolean Discontinued { get; set; }

        [Association("Product_Category", "CategoryID", "CategoryID", IsForeignKey =
true)]
        [Display(Name = "Category")]
        public ProductCategory Category { get; set; }
    }
}
```

ProductCategory.cs

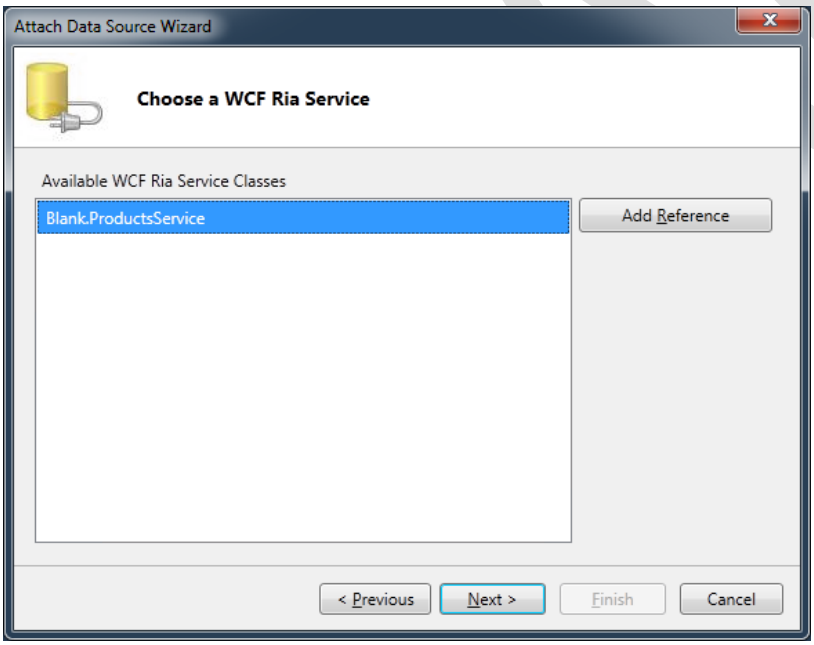
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace Blank
{
    public class ProductCategory
    {
        [ReadOnly(true)]
        [Key()]
        [Display(Name = "Category ID")]
        [ScaffoldColumn(false)]
        public Guid CategoryID { get; set; }

        [Required()]
        [Display(Name = "Category Name")]
        public string CategoryName { get; set; }

        [Display(Name = "Description")]
        public String Description { get; set; }

        [Display(Name = "Products")]
        [Association("Product_Category", "CategoryID", "CategoryID")]
        public ICollection<Product> Products { get; set; }
    }
}
```

Now you are ready to test your code. Create a LightSwitch Application and name it MyProducts, and then add the Blank extension (or whatever name you gave to your extension). To add the custom data source, go to Data Sources -> Add Data Source and the following dialog will be presented:

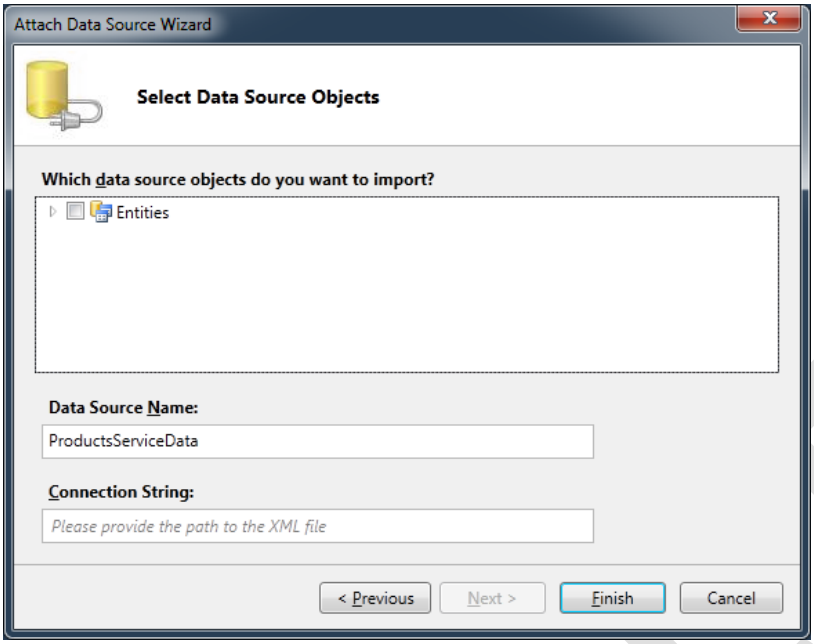


Select the WCF RIA Service and click Next.

In the next dialog, select the RIA Service and click Next as shown below:

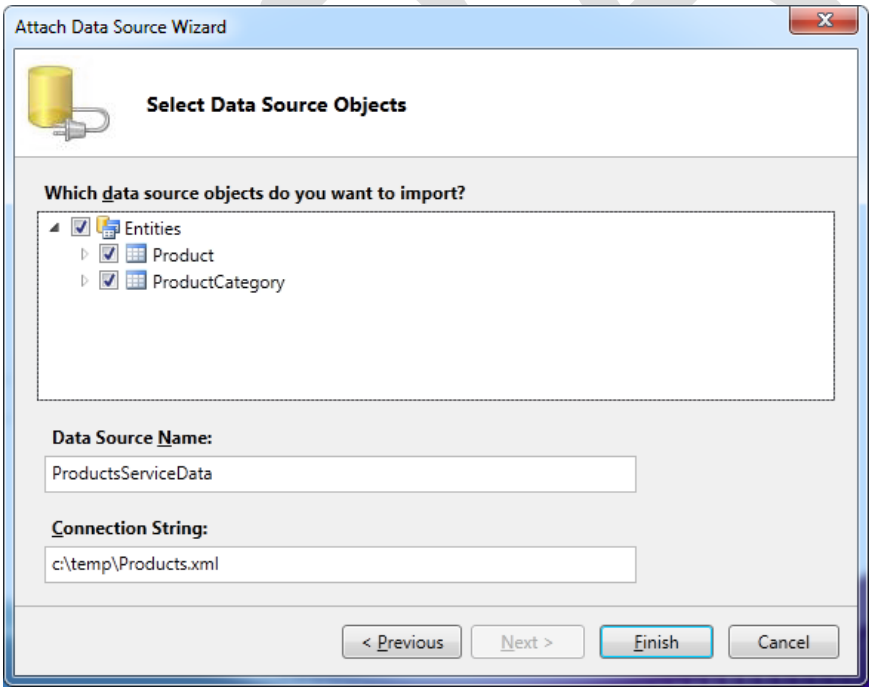


You will be presented with the following dialog:

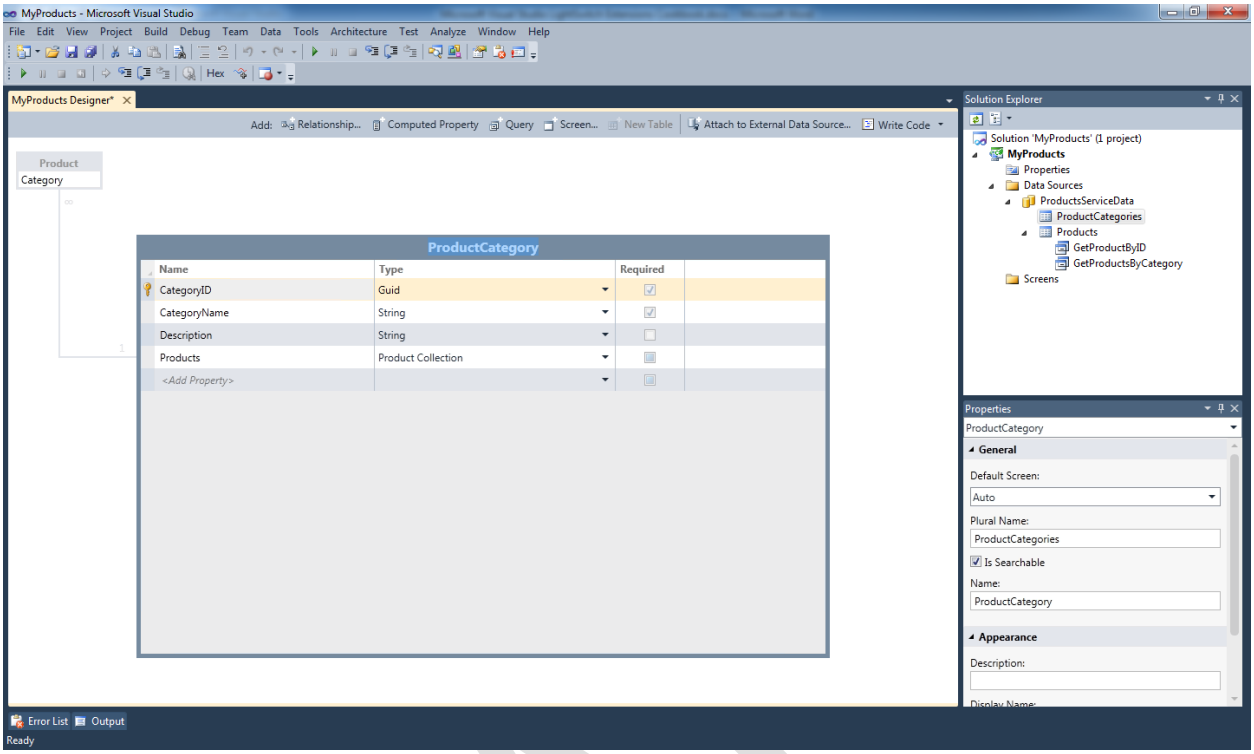


Notice the connection string text information - this was obtained by the Description attribute in the ProductsService.cs class file.

Now enter in the following information and select the Entities check box:

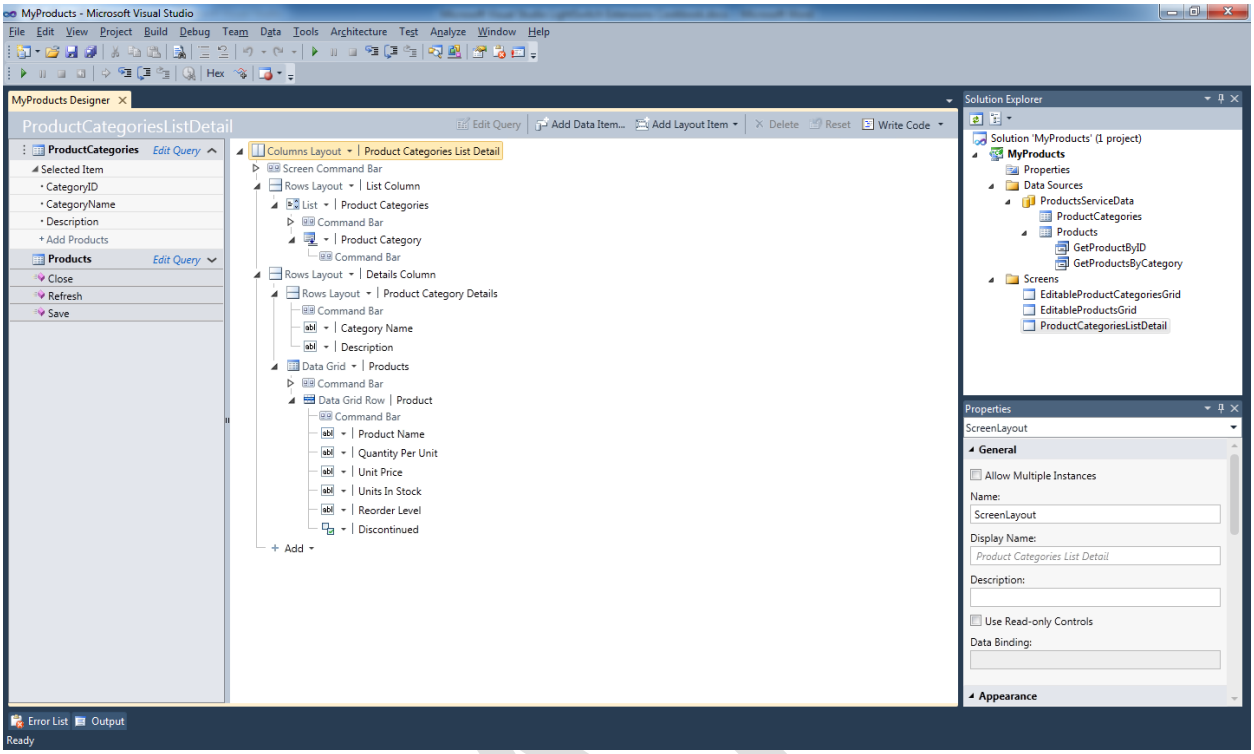


Click Finish and you should now see that your entities are present in the LightSwitch project as shown below:



At this point you can now create screens to interact these entities. You can create a grid screen for entering Products and a grid screen for Product categories.. Create a screen template based on list and details and select the productcategories as the screen data and ensure that you select the product details and product checkboxes.

The end result may look like the following:



Start adding some data and then look at the Products.xml file based on the location you specified earlier (c:\temp) and see the data that was added based on the screen input. (Note: Make sure you have access to the folder location in order to create the file.)

That is the end of this recipe for creating and using a custom data source extension.

Conclusion

In this document you have read what LightSwitch extensions are about and how they interact with the Visual Studio LightSwitch Product. You also have the recipes for creating the 6 fundamental extension types for LightSwitch.

This cookbook was developed for LightSwitch Beta 2 and will be updated for the final release of LightSwitch.

Please go to http://msdn.microsoft.com/lightswitch for more product and extensibility related information.