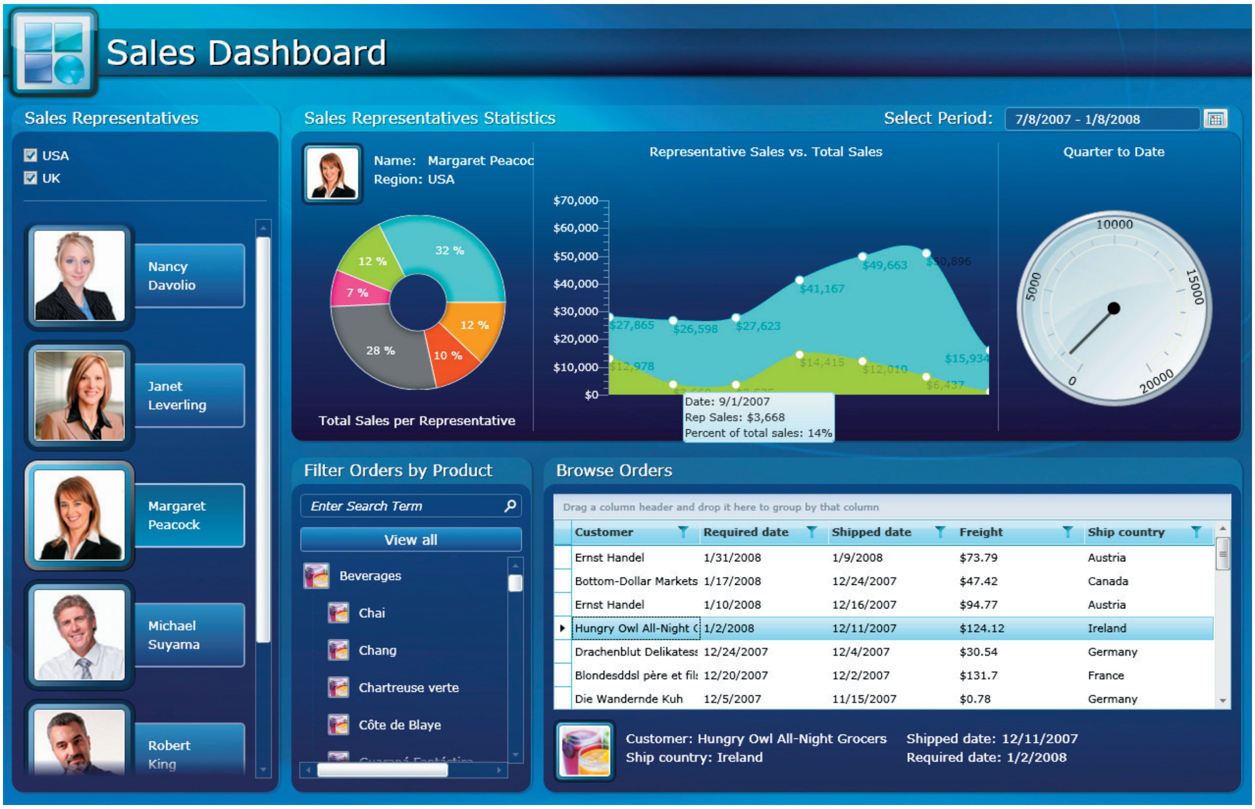
telerik
deliver more than expected

Telerik Sales Dashboard: an Extensible Cross-Platform Application

Contents



Select Period: 7/8/2007 - 1/8/2008

Name: Margaret Peacoc
Region: USA

Date: 9/1/2007
Rep Sales: $3,668
Percent of total sales: 14%

Customer	Required date	Shipped date	Freight	Ship country
Ernst Handel	1/31/2008	1/9/2008	$73.79	Austria
Bottom-Dollar Markets	1/17/2008	12/24/2007	$47.42	Canada
Ernst Handel	1/10/2008	12/16/2007	$94.77	Austria
Hungry Owl All-Night (	1/2/2008	12/11/2007	$124.12	Ireland
Drachenblut Delikates:	12/24/2007	12/4/2007	$30.54	Germany
Blondesddsl père et fil:	12/20/2007	12/2/2007	$131.7	France
Die Wandernde Kuh	12/5/2007	11/15/2007	$0.78	Germany

Customer: Hungry Owl All-Night Grocers
Ship country: Ireland
Shipped date: 12/11/2007
Required date: 1/2/2008

Introduction

Architecting an enterprise-quality application is not a trivial task. The development lifecycle of such an undertaking involves leveraging a set of well-defined and time-tested practices if code quality, maintainability and extensibility are of any importance. Over time, a great variety of design patterns and best practices have emerged to efficiently guide software developers through common application scenarios, making it more possible than ever to predictably deliver powerful solutions.

Of course, there is no such thing as “the universal pattern” that is the Holy Grail of software development. Each design pattern, practice or guidance is best suited to a specific set of scenarios, and no design pattern should be blindly followed by forcing an application to fit to that pattern just for the sake of it. If your project requirements do not perfectly fit a pattern, be brave and experiment – it is the purpose of a pattern to make your application work and not to prevent you from meeting your requirements.

The first time you decide to actually build an application using a best practices approach, though, you will probably find either obscure high-level architectural descriptions of patterns or incredibly complex reference implementations, which due to their complexity are practically useless as learning tools.

To address this difficulty and make it easier for everyone to learn enterprise-grade best practices, we developed the Telerik Sales Dashboard application which is freely available with full source code on our website. The purpose of this paper is to provide a clear overview and walkthrough of the way the Telerik Sales Dashboard application was built, the technologies that were used, as well as offering some insight on why important design decisions were made.

The Sales Dashboard Requirements

Telerik Sales Dashboard is an application for monitoring the performance of sales representatives within a company. Managers are able to carefully monitor and evaluate the sales operations of their company in real time using statistics calculated on the fly, detailed information about individual sales, as well as tools for filtering data by various parameters. The application can be accessed via both a web browser, using Silverlight, and on the desktop, using WPF. Multiple paths to the application maximize the potential usage scenarios and support in-house as well as field workers who are outside the corporate intranet. The web and desktop user interfaces are also identical so that users have a consistent experience across different platforms, eliminating any need to re-learn the application when transitioning between environments. Finally, the application is easily extensible and new functionality can be plugged-in to the already functioning product with minimal overhead, making the application more testable and in turn more stable.

Choice of Technologies

The Sales Dashboard is implemented entirely on the Microsoft stack of technologies. The RadControls for Silverlight and RadControls for WPF UI component suites are used to implement the user interface because they provide all the required controls, have excellent performance and usability, and practically cut in half the development time dedicated to the presentation layer.

On the back-end, Telerik OpenAccess ORM is used to provide data persistence and to ensure data is properly moved between the application and the database. In addition, OpenAccess saves critical development time by generating most of the data access code and greatly simplifying the process of querying data in the database with robust LINQ support.

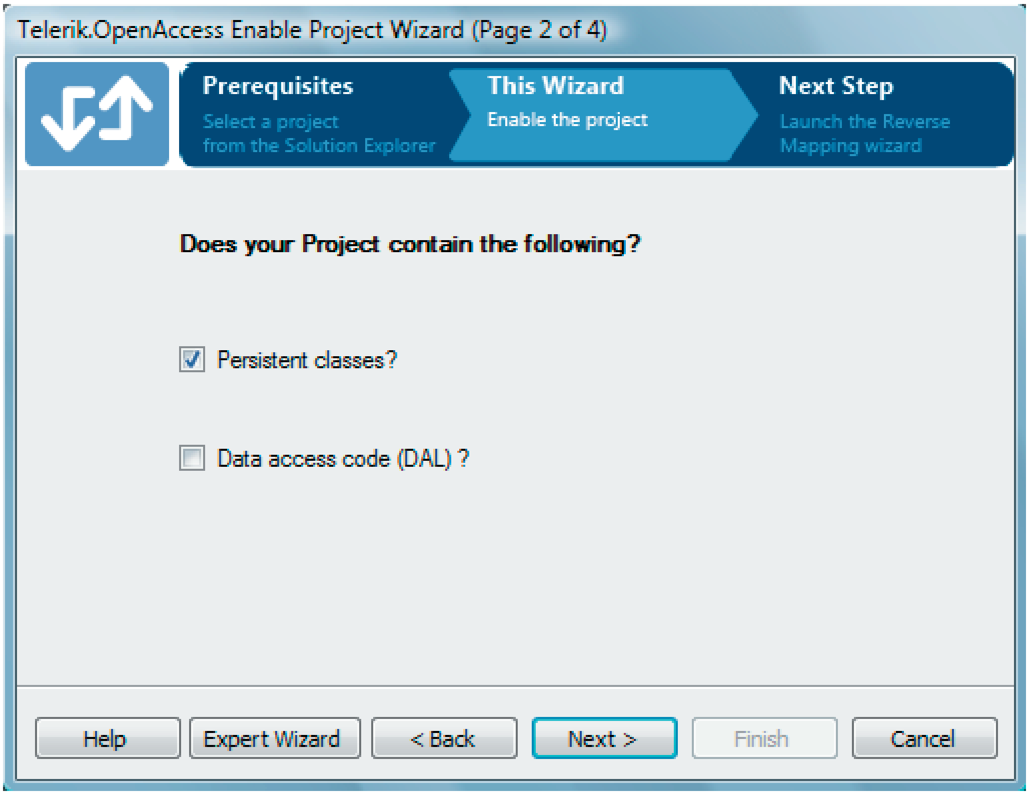
To compose the application, we used the second iteration of Prism - Microsoft’s Composite Application Guidance for WPF and Silverlight, which provides guidelines based on industry best practices for building extensible and modular applications. As part of the Prism guidance, the Telerik Sales Dashboard application is built using the Model-View-ViewModel (MVVM), Dependency Injection, and Inversion of Control (IoC) patterns.

The Implementation

Since Silverlight provides a subset of the functionality found in WPF, we start by implementing the core functionality of the application with a Silverlight user interface. This makes it much easier and quicker to later develop a WPF version of the presentation layer than it would be if we start with WPF.

First, we create the web project (*Telerik.SalesDashboard.Web*) that contains a WCF web service (*SalesDashboardService*) that serves data processed by our business rules to the presentation layer. Additionally, the web project hosts the Silverlight version of the user interface.

Then we add the Northwind database to the web project and create a project that will hold our data persistent classes (*Telerik.SalesDashboard.Data*). We enable OpenAccess ORM for both projects and specify that the *Telerik.SalesDashboard.Data* project will hold persistent classes, while the web project will contain data access code (*Fig 1*).



Telerik.OpenAccess Enable Project Wizard (Page 2 of 4)
Prerequisites
Select a project from the Solution Explorer
This Wizard
Enable the project
Next Step
Launch the Reverse Mapping wizard
Help
Expert Wizard
< Back
Next >
Finish
Cancel

Does your Project contain the following?

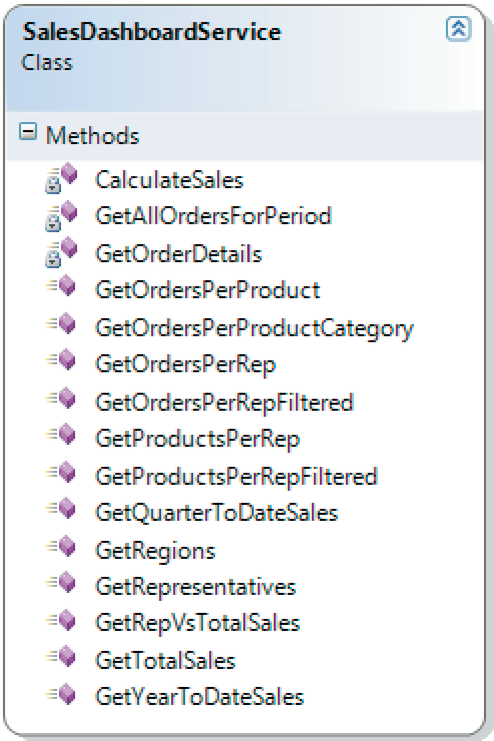
☑ Persistent classes?

☐ Data access code (DAL) ?

Fig. 1: Telerik OpenAccess ORM Enable Project Wizard in Visual Studio

Using OpenAccess's wizards we reverse-map the schema in our Northwind database to create the persistent classes that will be later consumed in the application. The ORM is capable of figuring out how to map the schema to classes, but it also allows you to very precisely fine-tune the mapping as needed.

Next, we implement the business logic in the WCF service layer. There are a total of 12 methods that the service offers publicly and 3 additional internal methods (*Fig 2*).



SalesDashboardService
Class

Methods
- CalculateSales
- GetAllOrdersForPeriod
- GetOrderDetails
- GetOrdersPerProduct
- GetOrdersPerProductCategory
- GetOrdersPerRep
- GetOrdersPerRepFiltered
- GetProductsPerRep
- GetProductsPerRepFiltered
- GetQuarterToDateSales
- GetRegions
- GetRepresentatives
- GetRepVsTotalSales
- GetTotalSales
- GetYearToDateSales

Fig. 2: Methods defined in Sales Dashboard WCF service

Now that the web project is ready, we start implementing the Silverlight part of the application. First we create the "shell" project (*Telerik.SalesDashboard.Shell*) that serves as the main application project in which individual modules" will be plugged. It contains a helper class called "Bootstrapper" that registers the modules that will be available in the application, and a user control defining the main user interface (*Shell.xaml*). The *Shell.xaml* file defines special placeholders, called ***regions*** in Prism, which determine where modules registered for a particular region will be rendered. Our design requires each module to be displayed in a visual container common for all modules, so we create a special user control called ***DashboardControl*** to handle this task. Here is what a region definition in *Shell.xaml* looks like:

```
<common:DashboardControl Regions:RegionManager.RegionName="RepresentativesRegion"
                         Title="Sales Representatives"
                         Grid.Row="1"
                         Grid.RowSpan="2"/>
```

In parallel to the "shell" project, we also create the *Telerik.SalesDashboard.Common* project that holds all resources common to the application modules. It includes custom UI controls, value converters used in XAML, images, interfaces, and custom events used for communication between modules.

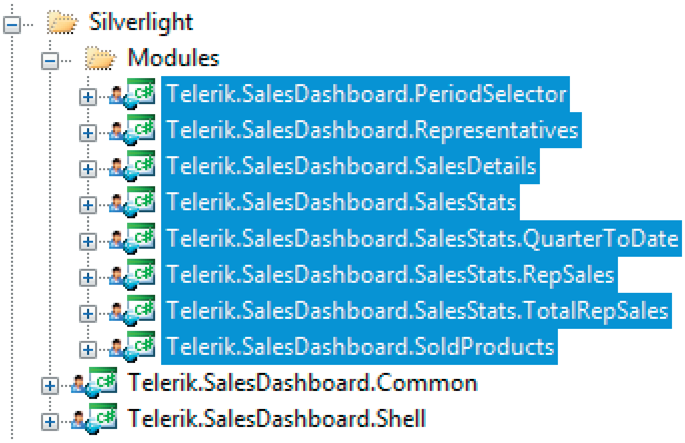
We must make a number of important architectural decisions in this project to maximize code maintainability. First, in order to keep our projects and modules loosely coupled, we will use interfaces for the persistent data classes and the web service providing data to the application. This frees our application from being dependent on the implementation of the persistent classes and web service, making it very easy to swap their implementations. Note that the shell project does not have a direct reference to the classes in the *Telerik.SalesDashboard.Data* project. So, to make the persistent classes implement our interfaces, we tweak the automatically generated partial classes (created when the service reference to the *SalesDashboardService* was added) to implement our interfaces (*see the ModelWrappers.cs file*). Another decision worth highlighting, again with the purpose of keeping the modules unaware of specific implementations, is a manual proxy that we create to the automatically generated service client. This helps the application preserve its loosely coupled architecture since our custom proxy implements an interface known to the modules.

Implementing the Modules

After the Shell and Common projects are ready, we start implementing the eight modules required for the application (*Fig 3*). Each module has the following:

1. A “viewmodel” file, such as *RepresentativesViewModel.cs*, that contains the module logic
2. A module definition file, such as *RepresentativesModule.cs*, that defines a module and registers it with a particular region in the main application container
3. One or more module views, such as *RepresentativesView.xaml*, that contain any required visual elements.

Additionally, a module may contain classes needed for converting data coming from the model to a format more suitable for use in the UI. As dictated by the Prism guidance, in the Sales Dashboard we strive to keep the code-behind files of the module views void of any application logic. Instead, we use data binding and commands extensively throughout the application, with only a few exceptions that are noted below in the module implementation details. For communication, the modules use Prism events to which they subscribe or publish data and thus each module is unaware of the rest.



Silverlight
  Modules
    Telerik.SalesDashboard.PeriodSelector
    Telerik.SalesDashboard.Representatives
    Telerik.SalesDashboard.SalesDetails
    Telerik.SalesDashboard.SalesStats
    Telerik.SalesDashboard.SalesStats.QuarterToDate
    Telerik.SalesDashboard.SalesStats.RepSales
    Telerik.SalesDashboard.SalesStats.TotalRepSales
    Telerik.SalesDashboard.SoldProducts
  Telerik.SalesDashboard.Common
  Telerik.SalesDashboard.Shell

Fig. 3: Projects in Visual Studio for each of the Sales Dashboard’s 8 modules

Module Implementation Details

Some modules in the Sales Dashboard require additional code to meet the application requirements. In this section, we will briefly highlight some of the more important decisions we must make during the development of the modules.

PeriodSelector Module

This module handles the task of displaying a RadCalendar in a popup that can be used to select a range of dates for filtering data displayed in other modules. To close the popup containing the RadCalendar control when the user clicks outside of it, we subscribe to the *MouseLeftButtonUp* event of the *RootVisual* element of the application. We do not use the *MouseLeftButtonDown* event because it is used and handled internally by most controls and thus it is inappropriate for our purpose.

Also, we need to bind several properties of UI elements in this module to properties defined in the code-behind of the view, which are then in turn bound to properties in the viewmodel. This code-behind binding is necessary because the project requirements are not achievable with the XAML bindings available in Silverlight. This *is* a deviation from the Prism guidance, but, as already established, patterns should not be followed blindly if they block application requirements.

Representatives Module

This module displays a list of sales representatives along with a checkbox filtering UI for filtering the list of reps to specific sales regions. To enable the filtering, we create a helper class *RegionViewModel* that is used for displaying the checkboxes with the region names. It has an *IsSelected* property and *IsSelectedChanged* event that allow us to know when a region is selected or deselected. When the module gets the region names from the Sales Dashboard service layer, it constructs *RegionViewModel* objects that hold the region names. Finally, the module listens for the *IsSelectedChanged* event of the *RegionViewModels*, and when the event occurs, the list of representatives is updated accordingly.

SalesDetails Module

In this module, the *Sale* and *GridDataSource* classes are created to make raw data more suitable for displaying in the UI.

The data displayed in the module is updated by four different events, fired by the *Representatives*, *PeriodSelector*, and *SoldProducts* modules.

SalesStats Module

The *SalesStats* module acts primarily as a logical container that wraps the modules responsible for displaying sales statistics. It contains the Prism regions in which the *QuarterToDate*, *RepSales*, and *TotalRepSales* modules are rendered.

RepSales Module

This module uses a RadChart to display a sales rep's sales compared against total company sales for a given period. To display the comparison, we create the *ChartDataPoint* class to "flatten" the two series (Rep Sales and Company Sales) to a single data source for display in the RadChart. We also use a trick in our XAML binding to display extra values in the RadChart data point tooltips:

```
<chart:SeriesMapping Label=”Rep Sales”>
    <chart:SeriesMapping.SeriesDefinition>
         <chart:SplineAreaSeriesDefinition ShowItemToolTips=”True”
                                    ToolTipFormat=”Date: #LEGENDLABEL{0}&#x0a;Rep
Sales: #Y{C0} &#x0a;Percent of total sales: #BUBBLESIZE{G2}%”
                                   DefaultFormat=”C0” />
         </chart:SeriesMapping.SeriesDefinition>
         <chart:SeriesMapping.ItemMappings>
              <chart:ItemMapping FieldName=”Date” DataPointMember=”XValue” />
         <chart:ItemMapping FieldName=”RepValue” DataPointMember=”YValue” />
          <chart:ItemMapping FieldName=”DateString” DataPointMember=”LegendLabel” />
          <chart:ItemMapping FieldName=”Percentage” DataPointMember=”BubbleSize” />
         </chart:SeriesMapping.ItemMappings>
</chart:SeriesMapping>
```

Because we want to display the date, rep sales, *and* percentage of total sales in each data point's tooltip (three unique values), we map the *BubbleSize* property of RadChart to the "Percentage" value in our source object. We can do this because the *BubbleSize* property is not used by RadChart when rendering a SplineArea chart. And by mapping this "extra" value to an unused property, we can declaratively bind our value to RadChart for use in the tooltip.

Finally, we also use the special formatting options provided by RadChart to display useful information in the tooltips that appears when users move the mouse pointer over a data point in the chart (see the *ToolTipFormat* property).

TotalRepSales Module

This module uses a RadChart to display total sales for a selected Sales Rep compared to other company sales representatives. In this module, we need to create the *ChartDataItem* class in order to transform our source data to a form more convenient for binding to our RadChart UI.

SoldProducts Module

In this module, as well as in the *PeriodSelector* module, we need to deviate a bit from the Prism guidance in order to accommodate our project requirements. The module displays a list of all products sold by the company with a textbox that can be used to filter the list. To support this scenario, we need to create two properties in the code-behind of the module view that act as a proxy between the UI elements and properties in the module viewmodel. This is needed because the binding mechanism available in Silverlight will not work in the module's scenario.

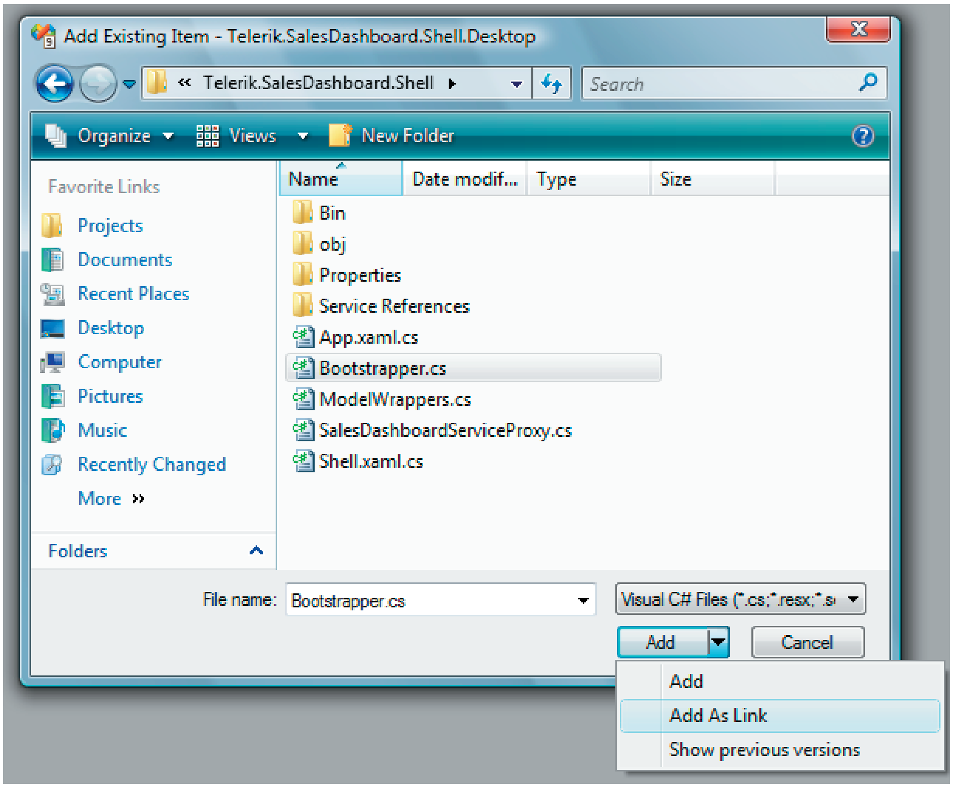
```
Binding b = new Binding();
b.Mode = BindingMode.TwoWay;
b.Path = new PropertyPath(“FilterText”);
this.SetBinding(SoldProductsView.FilterTextProperty, b);
```

```
searchTextBox.TextChanged += (s, e) =>
{
    if (searchTextBox.Text != searchTextBox.DefaultText)
    {
        this.FilterText = searchTextBox.Text;
    }
};
```

Building the WPF UI

After all modules are built targeting Silverlight, it is time to build the WPF version of the presentation layer. And thanks to our use of the RadControls, this process requires surprisingly little work. The process primarily involves configuration changes and almost no work on the application logic and UI, except for handling a few minor "discrepancies" that exist between Microsoft's various XAML platforms.

First we create a project structure mirroring the Silverlight projects, naming each project by appending ".Desktop" to the project name from the Silverlight implementation. Then, one by one, we must go through every project and "add as links" all corresponding files from the original Silverlight projects to the new WPF projects (Fig 4). This process creates references to the files in our Silverlight projects for use in our WPF projects, and it means any changes made to the original file are automatically applied to both WPF and Silverlight versions of the Sales Dashboard.



File name: Bootstrapper.cs

Visual C# Files (*.cs;*.resx;*.s

Fig. 4: Add As Link option in the Add Existing Item dialog in Visual Studio

Shell Project Changes Required for WPF

Microsoft technically ships three different XAML platforms: WPF, XBAP, and Silverlight. Each largely overlaps, but there are "discrepancies" that must be handled to enable an application to target the unique environments. For the Telerik Sales Dashboard, only the documented changes below are required to enable the application to run in all three environments.

The primary modifications that are needed are in the *Shell* project. The XAML code can be practically 100% reused, with the exception of changing the root element in the *Shell.xaml* file from *UserControl* to *Window*. This is because the default root visual element in Silverlight is different from WPF, so we must handle this Microsoft discrepancy for our application support both environments.

To support XAML Browser Application (XBAP) deployments, we also create a dedicated *Shell_XBAP.xaml* file. This enables the WPF version of Sales Dashboard to be run from the browser. The only change in the XBAP shell file is, again, to change the root element to *Page*. In the constructor of the XBAP shell we do the following:

```
Bootstrapper bootStrapper = new Bootstrapper();
bootStrapper.ShellPage = this;
bootStrapper.Run();
```

This code is making use of a special property we add to the "bootstrapper" that enables the application to run as an XBAP. Here are the modifications that we make to the *Bootstrapper.cs* file to enable the Telerik Sales Dashboard to run as a Silverlight, regular WPF, and XBAP application:

```
protected override DependencyObject CreateShell()
{

      this.Container.RegisterType<ISalesDashboardServiceProxy,  SalesDashboardServiceProxy>(new
ContainerControlledLifetimeManager());
#if XBAP
            return ShellPage;
#else
            Shell shell = Container.Resolve<Shell>();
#if SILVERLIGHT
            Application.Current.RootVisual = shell;
#else
            shell.Show();
#endif
            return shell;
#endif

        }

#if XBAP
        private DependencyObject shellPage;
        public DependencyObject ShellPage
        {
            get
            {
                return shellPage;
            }
           set
            {
                shellPage = value;
            }
        }
```

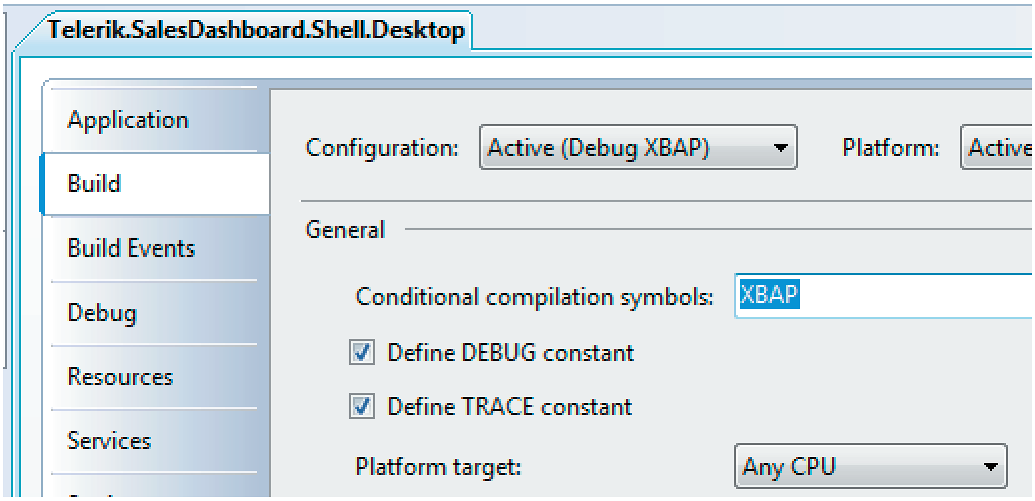
```
#endif
```

The modifications take care of properly initializing the application depending on whether we want to run it as a desktop application or in a browser. The code makes use of conditional preprocessor directives to determine if specific code blocks should be included in the build. This is a common technique for reusing code to target multiple environments and a language feature in both C# and VB.

In addition, we do the following in the *App.xaml.cs* file of the *Telerik.SalesDashboard.Shell.Desktop* project to completely support XBAP deployments:

```
protected override void OnStartup(StartupEventArgs e)
{
    base.OnStartup(e);
#if !XBAP
    new Bootstrapper().Run();
    this.ShutdownMode = ShutdownMode.OnMainWindowClose;
#else
    this.StartupUri = new Uri(“Shell_XBAP.xaml”, UriKind.Relative);
#endif
}
```

The last change required to support the XBAP scenario is to add the following compilation symbol in the project properties when the “Debug XBAP” build configuration is selected (*Fig 5*):



Configuration: Active (Debug XBAP)

Platform: Active

Conditional compilation symbols: XBAP

☑ Define DEBUG constant

☑ Define TRACE constant

Platform target: Any CPU

Fig. 5: Add the XBAP conditional compilation symbol to the Telerik.SalesDashboard.Shell.Desktop project build settings.

Additional Minor Changes Required for WPF

Now that the Shell project is prepared to support multiple environments, we turn our attention to the rest of the projects in our Sales Dashboard solution. Fortunately, to fully support our cross-platform environment, only a few more minor changes are required to 5 of the projects.

Telerik.SalesDashboard.Web.Desktop

To fully support WPF, we only need to duplicate the web project (creating one additional project), call it *Telerik.SalesDashboard.Web.Desktop,* and remove the reference to the Silverlight project. The only difference between the new copy and the original is that the new one will not have a reference to the Silverlight application. This reference causes problems when the WPF project is built.

Telerik.SalesDashboard.Common.Desktop

Unlike most linked project files, you must copy locally to the WPF project all of the images because linked files will not work. Also, the *ImagePathConverter* needs the following little tweak to ensure image paths are defined correctly in Silverlight and WPF:

```
#if !SILVERLIGHT
       wpfPathPrefix = “pack://application:,,,/”;
#endif
return String.Format(“{0}Telerik.SalesDashboard.Common;component/Images/{1}.png”, wpfPathPrefix,
temp);
```

Telerik.SalesDashboard.PeriodSelector.Desktop

Due to the different behavior of the *Popup* control in WPF and Silverlight, we need to manually position it in the WPF project:

```
#if !SILVERLIGHT
    this.calendarPopup.PlacementTarget = this.btn;
    this.calendarPopup.VerticalOffset = 0;
    this.calendarPopup.HorizontalOffset = -695;
#endif
```

Also, we need to exclude from the WPF version of the Sales Dashboard the code that closes the popup when the user clicks outside of it:

```
#if SILVERLIGHT
                Application.Current.RootVisual.MouseLeftButtonUp += new MouseButtonEventHandler(R
ootVisual_MouseLeftButtonUp);
#endif
```

Telerik.SalesDashboard.SalesStats.RepSales.Dekstop and .TotalRepSales.Desktop

In Silverlight there is no default WrapPanel, so we use our own RadWrapPanel in the Silverlight project. The WrapPanel control exists in WPF, though, so we use it and do not link the Silverlight project file, instead copying it and just changing RadWrapPanel to WrapPanel.

And with those small changes, our application is ready to fully support Silverlight, WPF, and XBAP deployments.

Conclusion

Building an enterprise-quality application with Silverlight *and* WPF does not have to be hard. The Telerik Sales Dashboard demonstrates that the main development efforts are primarily concentrated on the application logic and to a lesser extent on the user interface. With Telerik OpenAccess ORM taking care of the typically error-prone process of moving data around and Prism providing a framework for composing our application with modules, we are able to focus on the business logic and build a robust, loosely-coupled, and maintainable application.

By using RadControls for WPF and Silverlight we are able to maximize our return on experience by building an application once and easily reuse nearly all of the code to target multiple XAML environments. With the exception of a few minor discrepancies in Microsoft’s XAML platforms, we are able to reuse all of our UI code and succeed in porting the application from the browser to the desktop with minimum effort. Thanks to following the Prism guidance, the application’s modular architecture is easily extensible and testable which greatly reduces future maintenance and development work.

Download the Telerik Sales Dashboard application by visiting http://www.telerik.com/salesdahboard and see for yourself how the RadControls’ unique common code base makes it possible to easily build applications for Silverlight and WPF.

Download your free trial of the RadControls for WPF and Silverlight by visiting http://www.telerik.com/wpf and http://www.telerik.com/silverlight.