Chapter No. 4
"Building a Basic Dashboard"
In this package, you will find:

A Biography of the authors of the book
A preview chapter from the book, Chapter NO.4 "Building a Basic Dashboard"
A synopsis of the book’s content
Information on where to buy this book

About the Authors

Todd Snyder has been a Software Developer/Architect for over 17 years. During that time, he has spent several years as a Consultant, providing technical guidance and leadership for the development of Enterprise Class Systems on the Microsoft Platform. At Infragistics, he is a principal consultant who focuses on the design and construction of RIA and N-tier based applications. Todd is the co-leader for the New Jersey .NET user group (http://www.njdotnet.net/) and is a frequent speaker at trade shows, code camps, and Firestarters.

Joel Eden, Ph.D. has been working in the area of user experience and design methods for over 10 years. Currently a Senior Interaction Designer working on UX Tools at Infragistics, he previously spent 3 years in the Infragistics UX Services group, consulting for external clients. Prior to Infragistics, he worked at multiple design agencies in the Philadelphia area, as well as working at Lockheed Martin's Advanced Technology Labs. He holds a B.S. in Computer Science, and a PhD in Information Science, both from Drexel University.

I would like to thank my wife Kathleen and my daughters Abigail and Genevieve, for giving me reasons to design a better future.

For More Information:
www.packtpub.com/microsoft-silverlight-5-for-enterprise-dashboards/book
Jeff Smith has been a Visual Designer for 6 years. During that time he has been an Art Director at various agencies and studied Special Effects and Animation at NYU. A convert from Flash and Flex, he has been working with .NET technologies for the past 2 years, specializing in WPF and Silverlight. At Infragistics, he is an UX Visual Designer who focuses on the design, implementation, and User Experience. You can view some of his work at http://www.thinksinkstudio.com.

I would like to thank my friends, family, girlfriend, and colleagues for providing support and inspiration.

Matt Duffield is the Architect at emGovPower, a leading software provider for managing city and county governments. With over 15 years of working in IT, he enjoys building rich user-centric applications. Matt is very active in the community, speaking at user groups and code camps. He is an INETA speaker member and has been a Microsoft MVP in Client Application Development. He started working in .NET as soon as it was available to the development community and was also an early adopter of both WPF and Silverlight. He believes good business intelligence (BI) architecture is the formula for a successful software package that looks beyond just data and provides insightful information. His blog can be found at http://mattduffield.wordpress.com. You can follow him on Twitter at @mattduffield. Matt currently lives in Charlotte, North Carolina with his wife and two sons.

First of all, I would like to thank my family and colleagues who have been instrumental in helping me work on this book. It is their encouragement and enthusiasm that has in turn been my drive for writing. I have always loved working in Silverlight, and I look forward to many more years doing the same. I would also like to thank the team at Packt Publishing for allowing me to become a part of this book and having the patience and wisdom to help me during the writing process. It has been a wonderful experience, and I look forward to doing it again in the future.

Microsoft Silverlight 5: Building Rich Enterprise Dashboards

Welcome, to the exciting world of building Rich Enterprise Dashboards with Silverlight. Throughout this book, you will learn how to harness the power of the Silverlight 5.0 platform to build amazing Dashboards. This book goes beyond just showing the nuts and bolts of learning Silverlight and showcases how to create amazing dashboards that offer the best user experience and visual design that will impress your end users and peers.

What This Book Covers

Chapter 1, The Silverlight Technology, is an overview of the Silverlight 5.0 Platform and teaches you how to build Silverlight applications.

Chapter 2, Overview of Dashboards and Silverlight, answers why you should use Silverlight 5.0 to build dashboards and compares the use of Silverlight and HTML for building dashboards.

Chapter 3, Silverlight Design and Development Tools, is an overview of the different tools you will need to build a dashboard applications.

Chapter 4, Building a Basic Dashboard, is a walkthrough of how to build a simple Silverlight dashboard application.

Chapter 5, Dashboard Types and User Needs, outlines the three main types of dashboards: Strategic, Operational, and Analytical. By the end of this chapter, you will understand how to choose the right dashboard for your project depending upon your needs.

Chapter 6, Designing for Insight, throws light on designing concepts of bringing the end-user's attention to certain information. Using these concepts, we will design a dashboard that supports the scenario we have already created.

Chapter 7, Designing your Dashboard, covers the basics of styling a Silverlight dashboard with Microsoft Expression Blend. Through this chapter you will understand the key styling terminology, learn how to reference resources, and edit basic control templates.

Chapter 8, Building an End-to-End Solution, is an overview of building a real-world Silverlight dashboard application that uses the Microsoft SQL Server Adventure Works sample database. Through this chapter you will explore how to use the N-Tier application model to build out the AdventureWorks dashboard application.

For More Information:
www.packtpub.com/microsoft-silverlight-5-for-enterprise-dashboards/book
Chapter 9, *Data Access Strategies*, outlines the different data access strategies and technologies you can use when building your Silverlight dashboard application.

Chapter 10, *Building Dashboards in SharePoint and Silverlight*, covers how to build a SharePoint web part for hosting a Silverlight dashboard, and how to use the SharePoint Client Object Model to access data hosted in SharePoint.
Building a Basic Dashboard

In the previous chapters, you were introduced to the technology and tools that are required to build a Silverlight dashboard application. We will now build upon that knowledge as we walk through the process of creating a dashboard application, how to use the Model View View-Model (MVVM) pattern, and how to customize the user interface to better showcase the Key Performance Indicators (KPI) you want to display in your dashboard.

In this chapter, we will cover the following topics:

- Creating a new Silverlight project
- What is the MVVM pattern
- Building a dashboard application
- Customizing the UI to better showcase Key Performance Indicators (KPI)

Creating a Silverlight Dashboard Application

Now that you understand the technology and tools that make Silverlight tick, it's time to build your first Silverlight dashboard. We will walk through how to create a new Silverlight project and set up the dashboard user interface.

For More Information:
www.packtpub.com/microsoft-silverlight-5-for-enterprise-dashboards/book
To create a new Silverlight dashboard:

2. Select File | New Project.
3. Expand the C# node.
4. Under the Silverlight project list, select Silverlight Navigation Application.

If you do not see the Silverlight project types listed, most likely you are missing the Silverlight 5.0 Tools for Visual Studio 2010. You can download the tools from the main Silverlight site: http://www.silverlight.net/getstarted/.

For More Information:
www.packtpub.com/microsoft-silverlight-5-for-enterprise-dashboards/book
After you enter the project/solution name and select a location to store your project, the New Silverlight Application dialog will appear. This dialog allows you to define the host application and version you want to use to build your Silverlight Dashboard application.

After you click Ok, you should compile and run the newly created Silverlight project. This will ensure that the project was properly set up before we move onto the next step. For our first Silverlight Dashboard, we will be using a Grid and Chart to display data stored in an XML file that we will download from the host web project. In order to use these controls, we need to make sure the necessary references are added to our Silverlight project. Open the solution explorer in Visual Studio, and expand the references node under the Silverlight project. Right-click on the reference node, and select Add Reference. When the Add Reference dialog appears, add the following references:

- System.Windows.Controls.Data

If you do not see the reference for Data Visualization, then you need to download the Silverlight 4.0 toolkit from codeplex: http://silverlight.codeplex.com/

What is the MVVM pattern

In order to display data in our dashboard application, we need to take a moment and explore how data binding works in XAML and how it relates to the Model View View-Model (MVVM) pattern.

One of the key aspects of Silverlight and XAML is its rich data binding support. In XAML, you use the binding keyword to set up one or two-way data binding to the datacontext of the view. To define the datacontext for a view, you can databind it in XAML to a static reference or in the view’s code behind, and set it to an instance of object, e.g., a View-Model. In addition to data binding the data displayed in a textbox and listbox, you can set up buttons to bind to commands and use behaviors to extend the data binding capabilities of XAML.

Here is the XAML for data binding a TextBox and ListBox. You set the Mode=TwoWay so that any changes the user makes are automatically propagated to the View-Model. To data bind a ListBox, you set its ItemsSource to a collection type.

```xml
<TextBox Text="{Binding CustomerName, Mode=TwoWay}" />
<ListBox ItemsSource="{Binding CustomerCollection}" />
```
To set up command binding, set the **Command** property for a **Button** to an **ICommand** exposed by your View-Model. You set the **CommandParameter** property to pass a parameter value to a command:

```xml
<Button Content="Submit" Command="{Binding SaveCustomer}"
        CommandParameter="Save" />
```

**MVVM** pattern is based on the separation of concern principles of the Model View Controller (MVC) pattern. To use the MVVM pattern, you need to define three separate loosely coupled components to handle the use cases of your application. The **View** (XAML and Code Behind) should focus on displaying data to users; the **Model** represents the business behaviour and attributes of the application, and the **View-Model** takes on several responsibilities, including exposing the properties (Data and Command) you will bind to in XAML, responding to user actions, tracking UI state (e.g., Selected Item), and abstracting the interworkings of the Model from the view.

When building an application using the MVVM pattern, it's always a good idea to start out with a base view model class. This class will contain the common functionality all View-Models need for notifying their views when one of their attributes changes. This is done by implementing the **INotifyPropertyChanged** interface. When data changes in your View-Model, the **PropertyChanged** event notifies bound dependencies of the change so that they might update accordingly.

```csharp
using System.ComponentModel;
namespace Chapter4.ViewModels
{
    public class BaseViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
        protected void SendChangedNotification(string propertyName)
        {
            if (thisPropertyChanged != null)
            {
```
Building a Basic Dashboard

```csharp
private string _viewName = string.Empty;
public string ViewName
{
    get { return this._viewName; }
    set
    {
        if (this._viewName != value)
        {
            this._viewName = value;
            this.SendChangedNotification("ViewName");
        }
    }
}
```

Another handy class to create is the delegate command. This makes it easy to build commands that are bindable from XAML. The delegate command implements the ICommand interface and uses .NET delegates to notify a view model when a command has been invoked. To use a command, you need to expose it as a property off over your view model and set up the binding for a button or hyperlink to the command.

Microsoft and other third-party developers offer frameworks that include helper classes similar to the delegate command that make it easier to build MVVM-based applications.

- Microsoft Prism: http://compositewpf.codeplex.com/
- MVVM Light: http://mvvmlight.codeplex.com/
- Caliburn: http://caliburn.codeplex.com/

For More Information:
www.packtpub.com/microsoft-silverlight-5-for-enterprise-dashboards/book
Building the Dashboard
We will now be using the MVVM pattern to build out our Dashboard application. The main use case of the dashboard will be to display a list of products and allow you to filter the monthly sales data for the selected product.

We will be creating the following items:

- XML for storing the product sales data
- Product and Sales KPI model classes used to store the instance data displayed by the Dashboard

For More Information:
www.packtpub.com/microsoft-silverlight-5-for-enterprise-dashboards/book
Building a Basic Dashboard

- Product Sales View Model that will expose the list of products and sales data for the selected product
- XAML View contains a Grid for displaying the list of products and Chart for display the selected product sales data

Setting up the Data Source

To keep things simple, we will be loading the data source for the dashboard from an XML file (Products.xml), which is stored in the ClientBin folder, along with the application's XAP file.

Each product listed in the XML file contains the attributes that will be displayed by the grid and sub elements for storing the last six months of sales data for each product.

```xml
<Product>
  <ProductID>414324-423</ProductID>
  <ProductName>Tunnbröd</ProductName>
  <Supplier>PB Knäckebröd AB</Supplier>
  <Category>Grains/Cereals</Category>
  <QuantityPerUnit>12 - 250 g pkgs.</QuantityPerUnit>
  <UnitPrice>9.00</UnitPrice>
  <UnitsInStock>61</UnitsInStock>
  <UnitsOnOrder>10</UnitsOnOrder>
  <SalesData>
    <SalesMonth="1"Amount="2500" />
    <SalesMonth="2"Amount="4000" />
    <SalesMonth="3"Amount="3200" />
    <SalesMonth="4"Amount="2500" />
    <SalesMonth="5"Amount="1200" />
    <SalesMonth="6"Amount="1800" />
  </SalesData>
</Product>
```

To retrieve and load the data from the ClientBin folder, we will be using the WebClientDownloadStringAsync method to retrieve the XML file. Once the XML file is retrieved, we will be using XLINQ to load the data stored in XML into the product and sales model classes for our application.

```csharp
WebClient client = new WebClient();
client.DownloadStringCompleted +=
    newDownloadStringCompletedEventHandler (client_DownloadStringCompleted);
client.DownloadStringAsync(newUri("products.xml", UriKind.Relative));
```

For More Information:
www.packtpub.com/microsoft-silverlight-5-for-enterprise-dashboards/book
Defining the View-Model and Model classes

The main components of our application will be our View Model and Model classes. We will be creating two model classes: **Product** and **SaleKPI**, and a single View Model **ProductSalesViewModel**.

The product class is fairly simple and contains the properties that will be displayed by the grid, and a child collection for monthly sales data.

```csharp
using System.Collections.Generic;
namespace Chapter4.Model
{
    public class Product
    {
        public string ProductId { get; set; }
        public string ProductName { get; set; }
        public string Category { get; set; }
        public string QuantityPerUnit { get; set; }
        public decimal UnitPrice { get; set; }
        public IList<SaleKPI> SalesData { get; set; }
    }
}
```

The **SaleKPI** class will contain the two properties: **Month** and **Amount** that will be displayed by the Chart Control.

```csharp
namespace Chapter4.Model
{
    public class SaleKPI
    {
        public string Month { get; set; }
        public decimal Amount { get; set; }
    }
}
```

The **ProductSalesViewModel** will be responsible for exposing the properties. The dashboard's XAML view will be bind too, tracking the selected product so it can filter the sales data displayed, and loading the model with the XML data retrieved from the **products.xml** file.

When a user selects a product (row in the grid) the view model updates the **SalesData** property to display the sales data for the selected product.

```csharp
using System;
using System.Collections.Generic;
using System.IO;
```
using System.Linq;
using System.Net;
using System.Xml.Linq;
using Chapter4.Model;
namespace Chapter4.ViewModels
{
    public class ProductSalesViewModel : BaseViewModel
    {
        public ProductSalesViewModel()
        {
            LoadData();
        }
        private IList<Product> _products;
        public IList<Product> Products
        {
            get { return this._products; }
            set
            {
                if (this._products != value)
                {
                    this._products = value;
                    this.SendChangedNotification("Products");
                }
            }
        }
        private Product _selectedProduct;
        public Product SelectedProduct
        {
            get { return this._selectedProduct; }
            set
            {
                if (this._selectedProduct != value)
                {
                    this._selectedProduct = value;
                    this.SendChangedNotification("SelectedProduct");
                    if (this._selectedProduct != null)
                    {
                        this.SalesData = this._selectedProduct.SalesData;
                    }
                }
            }
        }
        private IList<SaleKPI> _salesData;
    }
}
When the View-Model is created, it sends an asynchronous request to retrieve the products.xml file from the ClientBin folder. When the asynchronous request returns, the View-Model uses XLINQ to load the data into the Product and SaleKPI model classes.

We are using XLINQ to query the returned XML that contains product data easily. XLINQ allows us to extract the elements and attributes for each product and its related sales data. We use the extract data to create a Product class and SaleKPI collection for each product element defined in the XML. To execute the defined XLINQ query, we call the ToList() command to load the product data for the dashboard.

```csharp
private void client_DownloadStringCompleted(object sender,
                                          DownloadStringCompletedEventArgs e)
{
    XDocument document = XDocument.Load(new StringReader(e.Result));
    var result = from d in document.Root.Descendants("Product")
                  select new Product
                            {
                                ProductId = d.Element("ProductID").Value,
                                ProductName = d.Element("ProductName").Value,
                                QuantityPerUnit = d.Element("QuantityPerUnit").Value,
                                Category = d.Element("Category").Value,
                                UnitPrice = decimal.Parse(d.Element("UnitPrice").Value),
                                SalesData = this.GetSalesData(d.Element("SalesData"))
                            };
    this.Products = result.ToList();
}
private IList<SaleKPI> GetSalesData(XElement element)
```
Building a Basic Dashboard

```csharp
    { var result = from d in element.Descendants("Sales")
        select new SaleKPI()
        {
            Month = d.Attribute("Month").Value,
            Amount = decimal.Parse(d.Attribute("Amount").Value),
        };
    return result.ToList();
```
<Controls:DataGrid.Columns>
<Controls:DataGridTemplateColumn Header="Product"
    SortMemberPath="ProductName">
    <DataTemplate>
        <StackPanel Orientation="Horizontal" Margin="5">
            <TextBlock Text="{Binding ProductName}" />
            <TextBlock Text=" (" />
            <TextBlock Text="{Binding ProductId}" />
            <TextBlock Text=" )" />
        </StackPanel>
    </DataTemplate>
</Controls:DataGridTemplateColumn>
<Controls:DataGridTextColumn Binding="{Binding Category}" Header="Category" />
<Controls:DataGridTextColumn Binding="{Binding QuantityPerUnit}" Header="Quantity Per Unit" />
<Controls:DataGridTextColumn Binding="{Binding UnitPrice, StringFormat='c'}" Header="Unit Price" />
</Controls:DataGrid.Columns>
</Controls:DataGrid>
<dv:Chart Title="Sales"
    Margin="5, 50, 0, 0"
    Height="300"
    VerticalAlignment="Top"
    BorderThickness="0">
    <dv:Chart.Axes>
        <dv:DateTimeAxis IntervalType="Months" Orientation="X"
            Location="Bottom">
            <dv:DateTimeAxis.AxisLabelStyle>
                <Style TargetType="dv:DateTimeAxisLabel">
                    <Setter Property="StringFormat" Value="{}{0:MMM}" />
                </Style>
            </dv:DateTimeAxis.AxisLabelStyle>
        </dv:DateTimeAxis>
        <dv:LinearAxis Orientation="Y"
            ShowGridLines="True"
            Interval="1000" Minimum="1000" Maximum="5000">
            <dv:LinearAxis.AxisLabelStyle>
                <Style TargetType="dv:AxisLabel">
                    <Setter Property="StringFormat" Value="{}{0:###, K}" />
                </Style>
            </dv:LinearAxis.AxisLabelStyle>
        </dv:LinearAxis>
    </dv:Chart.Axes>
</dv:Chart>

For More Information:
www.packtpub.com/microsoft-silverlight-5-for-enterprise-dashboards/book
Building a Basic Dashboard

```xml
<dv:LinearAxis />
</dv:Chart.Axes>
<dv:Chart.Series>
    <dv:LineSeries Title="Sales by Month"
        ItemsSource="{Binding SalesData}"
        IndependentValueBinding="{Binding Month}"
        DependentValueBinding="{Binding Amount}" />
</dv:Chart.Series>
</dv:Chart>
</StackPanel>
</ScrollViewer>
</Grid>

To set up the data binding between the dashboard's view and its view model, we can define a static reference to the view model and bind the data context of the view to the static reference, or set the data context for the view in the code behind. Either method will work for simple applications; but for more advanced applications, it is better to set up the data context in code, so you can manage the life cycle of the view model and handle sharing state between multiple view models.

```csharp
using System.Windows.Controls;
using Chapter4.ViewModels;
namespace Chapter4
{
    public partial class Home : Page
    {
        public Home()
        {
            InitializeComponent();
            this.Loaded += new System.Windows.RoutedEventHandler(Home_Loaded);
        }
        void Home_Loaded(object sender, System.Windows.RoutedEventArgs e)
        {
            this.DataContext = new ProductSalesViewModel();
        }
    }
}
```

To set up the product grid, we need to follow the given steps:

1. Set the grid ItemSource (property exposed by the view model).
2. Define the columns (properties in the product class).
3. Set up a two-way data binding between the grid's SelectedItem and the view.

For More Information:
www.packtpub.com/microsoft-silverlight-5-for-enterprise-dashboards/book
4. **Model's SelectedProduct property.**

5. **Turn off AutoGeneratedColumns.**

6. **Set the Height of the Grid to 300 and IsReadOnly to True.**

```xml
<Controls:DataGrid Height="300"
                    IsReadOnly="True"
                    AutoGenerateColumns="False"
                    ItemsSource="{Binding Products}"
                    SelectedItem="{Binding SelectedProduct, Mode=TwoWay}"
                    >
            <Controls:DataGrid.Columns>
                <Controls:DataGridTemplateColumn Header="Product"
                                               SortMemberPath="ProductName">
                    <DataTemplate>
                        <StackPanel Orientation="Horizontal" Margin="5">
                        <TextBlock Text="{(Binding ProductName)}" />
                        <TextBlock Text="(" />
                        <TextBlock Text="{(Binding ProductId)}" />
                        <TextBlock Text=")" />
                        </StackPanel>
                    </DataTemplate>
                </Controls:DataGridTemplateColumn>
                <Controls:DataGridTextColumn Binding="{Binding Category}" Header="Category" />
                <Controls:DataGridTextColumn Binding="{Binding QuantityPerUnit}" Header="Quantity Per Unit" />
                <Controls:DataGridTextColumn Binding="{Binding UnitPrice, StringFormat='c'}" Header="Unit Price" />
            </Controls:DataGrid.Columns>
        </Controls:DataGrid>
```

To use the Chart control, you need to define its axes and series types you want to display. Because we want the dashboard to display financial data across a time period; we will be setting up a DateTimeAxis (X axis) and a Linear Y axis. Make sure to set the IntervalType for the X Axis to Months, and its location to Bottom. For the Y Axis, set its interval type to 1000, and its Minimum and Maximum values.

To bind data to the chart, we need to define a series type and set its ItemSource to a property exposed by the view model. For the dashboard, set up a line series type, set its item source to SalesData, bind the independent value to the Month property, and the dependent value to the Amount property of the SaleKPI class.

For More Information:

Building a Basic Dashboard

```xml
<dv:Chart Title="Sales"
    Margin="5, 50, 0, 0"
    Height="300"
    VerticalAlignment="Top"
    BorderThickness="0">
<dv:Chart.Axes>
    <dv:DateTimeAxis IntervalType="Months" Orientation="X" Interval="1"
        Location="Bottom">
        <StyleTargetType="dv:DateTimeAxisLabel">
            <Setter Property="StringFormat" Value="{}{0:MMM}" />
        </Style>
    </dv:DateTimeAxis>

    <dv:LinearAxis Orientation="Y" ShowGridLines="True" Interval="1000"
        Minimum="1000" Maximum="5000">
        <StyleTargetType="dv:AxisLabel">
            <Setter Property="StringFormat" Value="{}{0:###, K }" />
        </Style>
    </dv:LinearAxis>
</dv:Chart.Axes>

<dv:Chart.Series>
    <dv:LineSeries Title="Sales by Month"
        ItemsSource="{BindingSalesData}"
        IndependentValueBinding="{Binding Month}"
        DependentValueBinding="{Binding Amount}" />
</dv:Chart.Series>
</dv:Chart>
```

At this point, you should have a fully working dashboard. Compile and run the application. When it displayed, you should see the data displayed by the chart control change to reflect the sales data for the selected product.

For More Information:

www.packtpub.com/microsoft-silverlight-5-for-enterprise-dashboards/book
Customizing the UI

When building a dashboard style application, one of the important things you need to keep in mind is defining a great user experience. There are several ways to accomplish this: make sure the information, especially the Key Performance Indicators (KPIs) displayed, matches the needs of the user. The visual design should be clean and simple, focusing on highlights KPI and other important data. It's important too to make sure the data displayed is organized correctly, and the necessary formatting is done.

Throughout the rest of the book, we will focus on how to build a Silverlight Dashboard application that offers the best user experiences, uses a professional clean looking visual design, and displays data from multiple different data sources.

Before moving on, let's look at a couple simple ways we can improve the user experience of the Dashboard by customizing the look and feel and applying formatting to the grid and chart controls.

To customize the look and feel of the grid, we are going to use a template column to combine two columns into one and apply formatting to the unit price column. In the following XAML, we have added a new template column named Product. Inside of the data template for the column, we have added a set of Text Block controls that allows us to format the new column ProductName (ProductId). If you want your template column to support sorting, make sure to set the SortMemberPath to a validated property of the Product class.

For More Information:
www.packtpub.com/microsoft-silverlight-5-for-enterprise-dashboards/book
Building a Basic Dashboard

```xml
<Controls:DataGridTemplateColumn Header="Product"
    SortMemberPath="ProductName">
    <Controls:DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal" Margin="5">
                <TextBlock Text="{Binding ProductName}" />
                <TextBlock Text="(" />
                <TextBlock Text="{Binding ProductId}" />
                <TextBlock Text=")" />
            </StackPanel>
        </DataTemplate>
    </Controls:DataGridTemplateColumn.CellTemplate>
</Controls:DataGridTemplateColumn>

To apply a currency format to the unit price column, add the StringFormat property to the binding string, and set the format to 'c'. This will make sure the column's data is displayed as currency.

```xml
<Controls:DataGridTextColumn Binding="{Binding UnitPrice, StringFormat='c'}" Header="Unit Price" />
``` Xml

To format the chart control, we can apply formatting to the DateTime axis to display the name of the month and reformat the Y Axis to display the numeric value using the defined format.

```xml
<dv:Chart.Axes>
    <dv:DateTimeAxisIntervalType="Months" Orientation="X" Interval="1" Location="Bottom">
        <dv:DateTimeAxis.AxisLabelStyle>
            <StyleTargetType="dv:DateTimeAxisLabel">
                <Setter Property="StringFormat" Value="{}{0:MMM}"/>
            </Style>
        </dv:DateTimeAxis.AxisLabelStyle>
    </dv:DateTimeAxis>
    <dv:LinearAxis Orientation="Y" ShowGridLines="True" Interval="1000"
        Minimum="1000" Maximum="5000">
        <dv:LinearAxis.AxisLabelStyle>
            <StyleTargetType="dv:AxisLabel">
                <Setter Property="StringFormat" Value="{}{0:###, K}" />
            </Style>
        </dv:LinearAxis.AxisLabelStyle>
    </dv:LinearAxis>
</dv:Chart.Axes>
```

For More Information:
www.packtpub.com/microsoft-silverlight-5-for-enterprise-dashboards/book
Summary

In this chapter, we walked through how to create a Silverlight dashboard application. We walked through how XAML data binding works and how to use the Model View View-Model (MVVM) pattern, and how to customize the Silverlight grid and chart controls to better showcase the Key Performance Indicators (KPI) of a Dashboard.

In the next chapter, we will give a broad sweep across the major types of dashboards: strategic, operational, and analytical. We will cover how to choose between the different types to meet the needs of the dashboard application users.

For More Information:
www.packtpub.com/microsoft-silverlight-5-for-enterprise-dashboards/book
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.