* 1. 

Hands-On Lab

Building Applications in Silverlight 4

Module 6: Printing the Schedule

**Contents**

[Introduction 3](#_Toc256194028)

[Exercise 1: Printing the Schedule on One Page 4](#_Toc256194029)

[Create the Printing ViewModel and View 4](#_Toc256194030)

[Hook up the Print Button 7](#_Toc256194031)

[Set the Printable Content 7](#_Toc256194032)

[Exercise 2: Printing a Schedule across Multiple Pages 10](#_Toc256194033)

[Pagination Concerns 10](#_Toc256194034)

[Add Logic for Pagination 10](#_Toc256194035)

[Page Numbering 14](#_Toc256194036)

Introduction

* 1. In this lab, you will enable end users to print out the planned schedule for an event that they built using the schedule planner you added in the previous lab.
  2. You can start from where you left off with the previous lab, if you completed it. But if you did not get to the end, or if it you would just prefer to work from a fresh starting point, make a copy of the solution in the StartingPoint folder for this lab—it contains the completed solution from the previous lab.

Exercise 1: Printing the Schedule on One Page

* 1. In this part of the lab, you will add the ability to print an attendee’s schedule for an event. In this first part, the output will all be on a single page. The structure of the code and XAML will be very similar in some respects to what you created in the earlier lab for the schedule planner. But there are a few differences in the detail for printing that warrant building a separate view rather than attempting to print using the existing components.

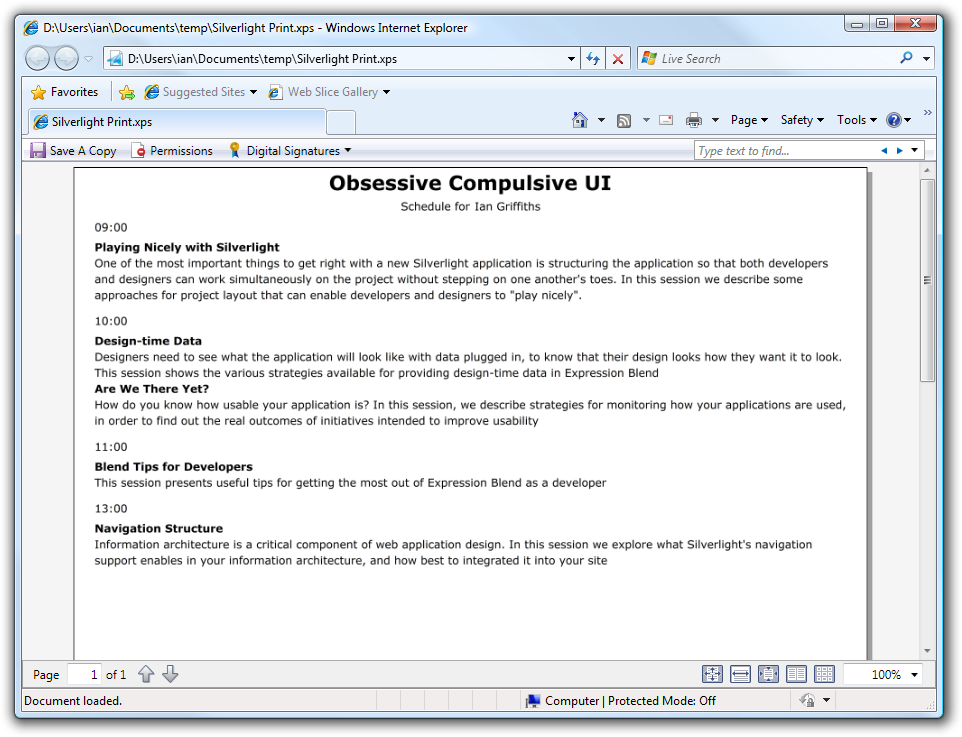
Create the Printing ViewModel and View

* 1. Open the SlEventManager solution in Visual Studio 2010.
  2. In the SlEventManager project’s ViewModels folder add a new class called SchedulePrintViewModel.
     1. This will act as the source data for the XAML file that will ultimately define the layout for the printed document.
  3. Add the following using directive:
     1. C#
     2. using System.ComponentModel;
  4. Implement **SchedulePrintViewModel** class like this:
     1. C#
     2. public class SchedulePrintViewModel
     3. {
     4. public string EventTitle { get; set; }
     5. public string AttendeeName { get; set; }
     6. public ICollectionView Talks { get; set; }
     7. public double PrintWidth { get; set; }
     8. }
     9. The PrintWidth property will hold the width of the printable area. We’ll need this to impose a layout constraint, as you’ll see.
  5. In the SlEventManager project’s Views folder, add a new Silverlight User Control called SchedulePrintView.
  6. Add the following Grid to the SchedulePrintView:
     1. XAML
     2. <Grid>
     3. <Grid.RowDefinitions>
     4. <RowDefinition Height="Auto" />
     5. <RowDefinition Height="Auto" />
     6. <RowDefinition Height="\*" />
     7. </Grid.RowDefinitions>
     8. <TextBlock
     9. FontSize="20"
     10. FontWeight="Bold"
     11. HorizontalAlignment="Center"
     12. Text="{Binding Path=EventTitle}"
     13. />
     14. <StackPanel
     15. Grid.Row="1"
     16. Orientation="Horizontal"
     17. HorizontalAlignment="Center"
     18. >
     19. <TextBlock Text="Schedule for " />
     20. <TextBlock Text="{Binding Path=AttendeeName}" />
     21. </StackPanel>
     22. <Viewbox
     23. VerticalAlignment="Top"
     24. StretchDirection="DownOnly"
     25. Grid.Row="2"
     26. >
     27. <ItemsControl
     28. Width="{Binding Path=PrintWidth}"
     29. ItemsSource="{Binding Path=Talks.Groups}"
     30. >
     31. <ItemsControl.ItemTemplate>
     32. <DataTemplate>
     33. <Grid Margin="20,5">
     34. <Grid.RowDefinitions>
     35. <RowDefinition Height="20" />
     36. <RowDefinition Height="Auto" />
     37. </Grid.RowDefinitions>
     38. <TextBlock Text="{Binding Path=Name, StringFormat=HH:mm}" />
     39. <ItemsControl
     40. Grid.Row="1"
     41. ItemsSource="{Binding Path=Items}"
     42. >
     43. <ItemsControl.ItemTemplate>
     44. <DataTemplate>
     45. <Grid>
     46. <Grid.RowDefinitions>
     47. <RowDefinition Height="Auto" />
     48. <RowDefinition MaxHeight="40" />
     49. </Grid.RowDefinitions>
     50. <TextBlock
     51. Text="{Binding Path=TalkTitle}"
     52. FontWeight="Bold"
     53. />
     54. <TextBlock
     55. Grid.Row="1"
     56. Text="{Binding Path=TalkAbstract}"
     57. TextWrapping="Wrap"
     58. />
     59. </Grid>
     60. </DataTemplate>
     61. </ItemsControl.ItemTemplate>
     62. </ItemsControl>
     63. </Grid>
     64. </DataTemplate>
     65. </ItemsControl.ItemTemplate>
     66. </ItemsControl>
     67. </Viewbox>
     68. </Grid>
     69. This is similar to the markup you created for the schedule planner, but with a few subtle differences. We can’t use tooltips in printed output because tooltips rely on mouse interaction, something that doesn’t work too well on paper. On screen, we use tooltips to show the full abstract, enabling us to truncate in the usual view, but when printing, we need to show the whole abstract up front rather than trimming it. (Also, if you did the optional Fluid UI section in the previous lab, you’ll notice that we’re not using a ListBox any longer because we only added that for the entry and exit animations.)
     70. Also note that we’ve wrapped the ItemsControl in a Viewbox, a new feature in Silverlight 4. This can enlarge or shrink items to fit the available space. We’ve set the StretchDirection to DownOnly meaning this particular Viewbox will never try to enlarge the content—if it’s smaller than the space available it just won’t fill it. But if there’s too much content, it will be shrunk, to fit it on the page. (In the planner UI we were able to deal with this problem by using a ScrollViewer, but of course that’s another UI element type that’s not a great choice for printed output.)
     71. The Viewbox is the reason we needed the ViewModel to make the PrintWidth available. Notice that the outer ItemsControl’s Width is constrained to be PrintWidth. Without that, the Viewbox would allow the ItemsControl to grow as wide as it liked—the Viewbox performs unconstrained layout on its children. The result would be that all each talk abstract would occupy a single, possibly very long line—the absence of a constraint would mean the text would never be wrapped because the line of text could grow indefinitely long. The Viewbox would then shrink everything to fit. The result would be tiny text with very long talk abstract lines. By forcing the ItemsControl not to get wider than PrintWidth, word wrapping kicks in, and the Viewbox only does its job once the content becomes too tall to fit.

Hook up the Print Button

* 1. Open SchedulePlanner.xaml.
  2. Add a new button to the UI at the top right.
  3. Set its content to Print…
  4. Name the button printButton.
  5. Add a Click event handler for **printButton**.
     1. There’s a slight challenge to getting this button to behave well in layout. In an earlier lab you added a Canvas that covers the whole UI, to make it possible to remove the context menu when the user clicks outside the menu. Because this Canvas is on top, Visual Studio will add new elements to that when you drop them onto the design surface. But because of how this layout resizes, the button would be better off being in the Grid. You can either edit the XAML manually to put it where you need it, or you could use Blend, which has more sophisticated support for working with multiple overlapping layers of content, and deeply nested containers. Or you could temporarily remove the Canvas while you position the button. Or you could just live with the fact that it won’t appear quite where you want when you resize the UI.
  6. In the SchedulePlanner.xaml.cs code behind file, add these using directives:
     1. C#
     2. using System.Windows.Printing;
     3. using System.Windows.Data;
     4. using System.ComponentModel;

Set the Printable Content

* 1. In the Click handler you just added, write this code to create a PrintDocument
     1. C#
     2. PrintDocument pd = new PrintDocument();
  2. Write the following code to set the PrintPage event handler
     1. C#
     2. pd.PrintPage += (s, pe) =>
     3. {
     4. };
  3. Inside the lambda, create an instance of the **SchedulePrintView**
     1. C#
     2. SchedulePrintView printView = new SchedulePrintView();
  4. Create an instance of the **SchedulePrintViewModel** and initialize its properties as shown in the code below.
     1. C#
     2. SchedulePrintViewModel printViewModel = new SchedulePrintViewModel
     3. {
     4. EventTitle = \_viewModel.EventTitle,
     5. AttendeeName = WebContext.Current.User.FriendlyName,
     6. Talks = (new CollectionViewSource
     7. {
     8. Source = \_viewModel.SubscribedTalks,
     9. GroupDescriptions =
     10. {
     11. new PropertyGroupDescription("TalkStartTime")
     12. },
     13. SortDescriptions =
     14. {
     15. new SortDescription("TalkStartTime", ListSortDirection.Ascending)
     16. }
     17. }).View,
     18. PrintWidth = pe.PrintableArea.Width
     19. };
     20. printView.DataContext = printViewModel;
     21. pe.PageVisual = printView;
  5. After the PrintPage handler, invoke the **Print** method of the **PrintDocument**
     1. C#
     2. pd.Print("Schedule");
     3. This creates a PrintDocument, and attaches a handler for its PrintPage event. That handler creates the print view and view model.
     4. We’re creating the CollectionViewSource in code here and putting its View property directly into the view model. We’re doing this because in the current preview, putting a CollectionViewSource in XAML doesn’t appear to work when printing. The grouped data does not appear to be available for binding at the point at which printing happens. (The data binding system often defers loading of data. This is usually not a problem on screen, because the screen will update as soon as the information becomes available. But with printing it can be a problem: if you don’t have the data by the time you print, it won’t make it onto the page.) By putting this in code and extracting the view source’s View property directly, we seem to get the grouped data when needed.
  6. Run the application.
  7. Log in as the user **ian** and password of **P@ssw0rd**
  8. Go to the event planner for the first event.
  9. Click the Print… button. You should see the print dialog appear.
  10. Select a printer.
      1. You might want to use the Microsoft XPS Document Writer because that way, if things aren’t perfect first time, you won’t waste paper printing things out for real.
  11. Print the document, and if you printed to an XPS file, find it and open it. You should see the schedule printed out.
      1. 
  12. Figure 1
  13. Printing the Content

Exercise 2: Printing a Schedule across Multiple Pages

Although a schedule is fairly likely to fit comfortably onto a single page, and we can force it to fit uncomfortably if necessary thanks to the Viewbox, squashing things to fit on one page is not a good general-purpose solution. In practice, splitting content across multiple pages is often necessary, so we’ll modify the printing code to support this.

Pagination Concerns

* 1. Run the application
  2. Go into the schedule planner and subscribe to lots of talks.
  3. Print the talks.
     1. If you added a lot of talks, you may notice the content’s font is getting smaller when printed (there is no page 2). If you get 3 or more talks in every time slot for the whole day, you’ll see that when you print, the list of talks starts to get narrower, as the Viewbox has to shrink it down to make it fit vertically. This ensures we have enough items in the schedule to make paging important.
     2. To paginate the content, we’ll need to take control of how many items we show on screen at once. This means we need to stop relying on the ItemsControl to display the top-level list of groups.
  4. Stop the application.

Add Logic for Pagination

* 1. Add a new user control to the Views folder called ScheduleTimeSlotPrintView.
  2. Replace ScheduleTimeSlotPrintView’s Grid named **LayoutRoot** with the following content:
     1. XAML
     2. <Grid Margin="20,5">
     3. <Grid.RowDefinitions>
     4. <RowDefinition Height="20" />
     5. <RowDefinition Height="Auto" />
     6. </Grid.RowDefinitions>
     7. <TextBlock Text="{Binding Path=Name, StringFormat=HH:mm}" />
     8. <StackPanel
     9. x:Name="talkPanel"
     10. Grid.Row="1"
     11. />
     12. </Grid>
     13. Notice the StackPanel. We are going to populate that with the talks for the time group, so this fills the role that was previously taken by the inner ItemsControl. So we’re getting rid of both ItemsControls. We’re not actually going to split individual time slots across page boundaries. All the talks from a given slot will be on the same page. However, we still can’t use the ItemsControl to generate the individual talks within a slot, albeit for a slightly different reason.
     14. We’re avoiding ItemsControl at the top level so that we can paginate, but we’re avoiding a nested ItemsControl here simply because it doesn’t work. A databound ItemsControl defers the generation of its child items. While the information will be available in time to print, it won’t be available at the point at which we try to work out how many items will fit on a page. If the information for the nested items control isn’t available at that point, we can’t know how big it’s going to be. If we ask the ItemsControl how big it wants to be at that point, it will return a preferred height of 0. Since our pagination calculations need to know how big this control will be, we can’t use ItemsControl. Instead we’ll have to populate the panel manually.
  3. Add the following using statements to the ScheduleTimeSlotPrintView.xaml.cs codebehind:
     1. C#
     2. using SlEventManager.Web;

using System.Windows.Data;

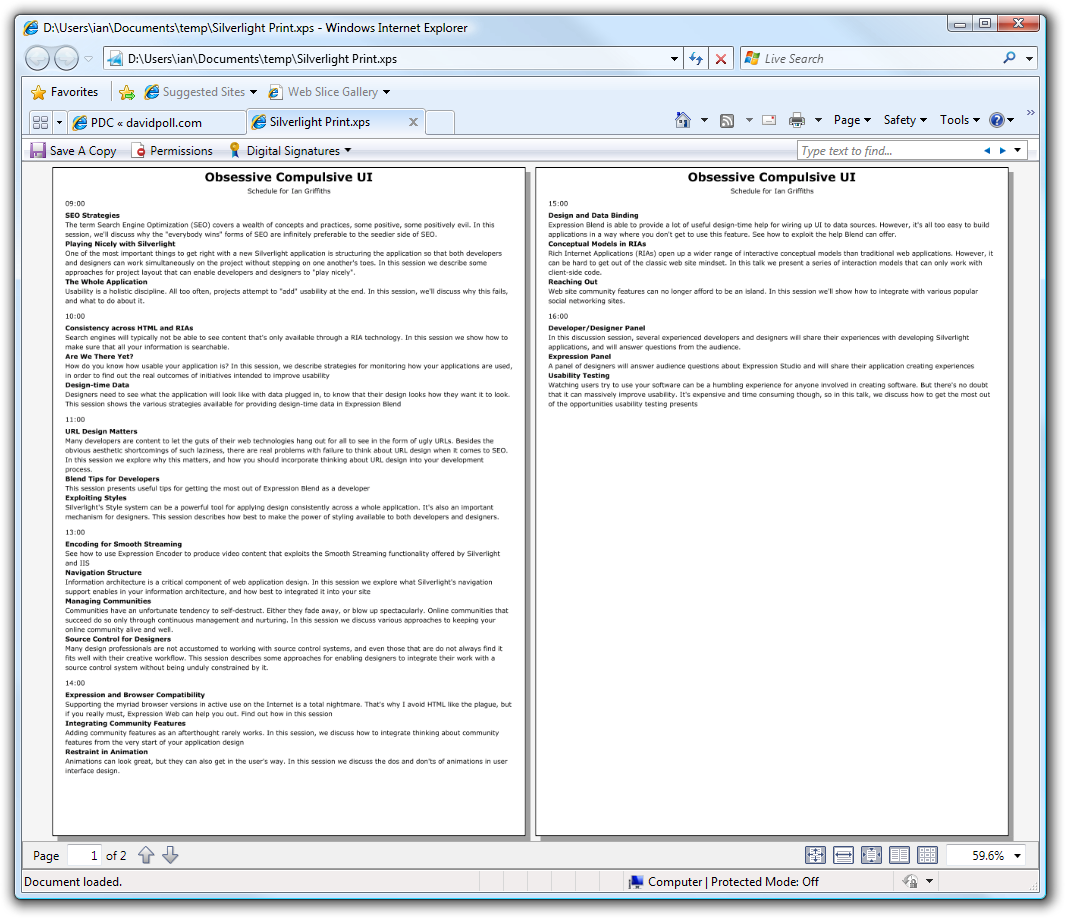
* 1. Add the following to the ScheduleTimeSlotPrintView.xaml.cs codebehind:
     1. C#
     2. public void SetData(CollectionViewGroup cvg)
     3. {
     4. this.DataContext = cvg;
     5. foreach (Talk item in cvg.Items)
     6. {
     7. talkPanel.Children.Add(new TextBlock
     8. {
     9. Text = item.TalkTitle,
     10. FontWeight = FontWeights.Bold
     11. });
     12. talkPanel.Children.Add(new TextBlock
     13. {
     14. Text = item.TalkAbstract,
     15. TextWrapping = TextWrapping.Wrap
     16. });
     17. }
     18. }
     19. For each talk this will generate a title and an abstract, just like the template we had previously been using did.
  2. Open SchedulePrintView.xaml, find the Viewbox.
  3. Delete the **Viewbox** and its contents. Replace it with the following **Border**:
     1. XAML
     2. <Border
     3. x:Name="timeSlotContainer"
     4. Grid.Row="2"
     5. />
  4. Add the following using statements to the SchedulePrintView.xaml.cs codebehind:
     1. C#

using System.Windows.Data;

* 1. In the SchedulePrintView.xaml.cs code behind, add this method which builds a panel containing the talks:
     1. C#
     2. public int PopulatePage(IEnumerable<object> items, int startingPoint)
     3. {
     4. double containerWidth = timeSlotContainer.ActualWidth;
     5. double containerHeight = timeSlotContainer.ActualHeight;
     6. StackPanel timeSlotPanel = new StackPanel();
     7. timeSlotPanel.Width = containerWidth;
     8. timeSlotContainer.Child = timeSlotPanel;
     9. int itemsAdded = 0;
     10. this.UpdateLayout();
     11. foreach (object item in items.Skip(startingPoint))
     12. {
     13. ScheduleTimeSlotPrintView uc = new ScheduleTimeSlotPrintView();
     14. uc.SetData((CollectionViewGroup) item);
     15. uc.DataContext = item;
     16. timeSlotPanel.Children.Add(uc);
     17. this.UpdateLayout();
     18. timeSlotPanel.Measure(new Size(containerWidth, double.PositiveInfinity));
     19. if (timeSlotPanel.DesiredSize.Height > containerHeight && itemsAdded > 0)
     20. {
     21. timeSlotPanel.Children.Remove(uc);
     22. break;
     23. }
     24. itemsAdded += 1;
     25. }
     26. return itemsAdded;
     27. }

The purpose of this code is to put as many items onto the current page as will fit. The code relies on Silverlight’s layout system to do most of the work. It creates a StackPanel (timeSlotPanel), which will contain the items to be printed for this page, and adds child items one at a time. Each time round the loop, we call this.UpdateLayout() to get Silverlight to lay the page out based on its current content.

Then we find out how much space is required by the items we’ve added so far by calling Measure on the StackPanel. It’s fairly unusual to call this directly, because Silverlight calls Measure for you on all your UI elements as part of the layout mechanism to find out how much space they would like. But even though Measure will have been called here during UpdateLayout, the layout process goes on to do further work, and by the time UpdateLayout returns, we will know how much space the StackPanel has been allocated in the final layout, whereas what we really want to know is how much it wanted: it may have been allocated less space than it desired. (And if that happens, it means we’ve tried to put more items on the page than will fit.)

* + 1. We call Measure ourselves, passing in double.PositiveInfinity as the height. This requests a vertically unconstrained layout: a mode of layout in which the element is required to work out how much space it would like in an ideal world, if there were no constraints. (Note that we constrain the width—we are essentially asking “If you were this wide, how tall would you need to be for all your contents to fit?”) If the StackPanel would like more space than is actually available, we know that adding this last item pushed us just past the point where the items will fit. We remove the last item to get back to a point where we’ve not overfilled the page, and then break out of the loop. The code returns the number of items that it added, so that the calling code knows whether it needs to print more pages, and if so, what item it should start at for the next page.
  1. In SchedulePlanner.xaml.cs, find the print button click handler.
  2. Add this variable to the start of the method (this tracks the number of items that have been printed so far)
     1. C#
     2. int totalItemsPrinted = 0;
  3. Then immediately after the line that sets the PageVisual property, add this code:
     1. C#
     2. totalItemsPrinted += printView.PopulatePage(
     3. printViewModel.Talks.Groups, totalItemsPrinted);
     4. pe.HasMorePages = totalItemsPrinted < printViewModel.Talks.Groups.Count;
     5. This calls the code you just added to populate the current page, and then works out whether it needs more pages after this one.
  4. Run the application and print the schedule to an XPS file. You should now find it prints across two pages:
     1. 
  5. Figure 2
  6. Printing Multiple Pages of Content
     1. With multipage printouts it’s often useful to add page numbers. This is relatively straightforward to achieve.

Page Numbering

* 1. Open the SchedulePrintViewModel class add the following property:
     1. C#
     2. public int PageNumber { get; set; }
  2. In SchedulePlanner.xaml.cs, go to the print click handler.
  3. After the totalItemsPrinted variable, add another variable to track the current page:
     1. C#
     2. int currentPage = 1;
  4. Find the code that constructs a SchedulePrintViewModel using the C# object initializer to set the properties.
  5. Add one more property initializer to set the PageNumber property to currentPage++.
     1. The increment operator will ensure the page number goes up each time.
  6. In SchedulePrintView.xaml, find the Grid.RowDefinitions section near the top.
  7. Add a fourth RowDefinition with a Height of Auto.
  8. Inside the **Grid**, add the following **TextBox**:
     1. XAML
     2. <TextBlock
     3. Grid.Row="3"
     4. Text="{Binding Path=PageNumber}"
     5. HorizontalAlignment="Center"
     6. />
  9. Run the application and print again. This time you should see page numbers at the bottom of the page.