* 1. 

Hands-On Lab

Building Applications in Silverlight 4

Module 5: Schedule Planner with DataGrid, Grouping, Right Mouse Click

**Contents**

[Introduction 3](#_Toc256193974)

[Exercise 1: Showing Talks Grouped by Track 4](#_Toc256193975)

[Create the Scheduler Planner View and ViewModel 4](#_Toc256193976)

[Create a Navigation Helper EventArg 5](#_Toc256193977)

[Add Commanding 6](#_Toc256193978)

[Create the Buttons and Bind them to Commands 6](#_Toc256193979)

[Load the Bound Collections 7](#_Toc256193980)

[Exercise 2: Showing the User’s Session Schedule 12](#_Toc256193981)

[Overview 12](#_Toc256193982)

[Show User’s Talks 12](#_Toc256193983)

[Exercise 3: Adding Sessions to the Schedule 17](#_Toc256193984)

[Add Sessions to a Schedule 17](#_Toc256193985)

[Adding Fluid UI Features (Optional) 19](#_Toc256193986)

[Exercise 4: Removing Sessions from the Schedule 22](#_Toc256193987)

[Create a Context Menu 22](#_Toc256193988)

[Add Right Click Support 23](#_Toc256193989)

[Closing the Popup Menu 24](#_Toc256193990)

[Add Support for Removing a Session 24](#_Toc256193991)

Introduction

* 1. In this lab, you will enable end users to plan their schedule for an event for which they have registered.
  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: Showing Talks Grouped by Track

* 1. In this part of the lab, you will create the page through which users will manage their schedule for an event, and incorporate it into the application’s navigation structure. You will then add a data grid showing the available talks, grouped by track using Silverlight 4’s grouped collection view source.

Create the Scheduler Planner View and ViewModel

* 1. Open the SlEventManager solution in Visual Studio 2010.
  2. In the SlEventManager project’s Views folder, add a new Silverlight Page called SchedulePlanner.
  3. In the project’s ViewModels folder, add a corresponding view model class called SchedulePlannerViewModel.
  4. Make this class derive from the ViewModelBase class you wrote in an earlier lab.
     1. The schedule planner page will be used to plan a specific event. As with the event editor page in the earlier lab, we’ll use a query string parameter in the URL to indicate the event ID. But in that earlier lab, we weren’t yet using view models, so this will work a little differently. The view will need to indicate to the view model which event ID is in the URL.
  5. Add the following method to the SchedulePlannerViewModel:
     1. C#
     2. public void Load(int eventId)
     3. {
     4. // Will be filled in later in the lab
     5. }
  6. In the SchedulePlanner.xaml.cs code behind, add and initialize a field to hold the view model:
     1. C#
     2. SchedulePlannerViewModel \_viewModel = new SchedulePlannerViewModel();
  7. Add a using directive for the SlEventManager.ViewModels namespace
  8. In the constructor, after the call to InitializeComponent, put the view model in the DataContext:
     1. C#
     2. this.DataContext = \_viewModel;
  9. Find the OnNavigatedTo method that Visual Studio created in your page’s code behind
  10. Implement the OnNavigatedTo method as shown here:
      1. C#
      2. protected override void OnNavigatedTo(NavigationEventArgs e)
      3. {
      4. string eventId = NavigationContext.QueryString["EventID"];
      5. \_viewModel.Load(int.Parse(eventId));
      6. }
      7. We need to make it possible for the user to navigate to this page. In an earlier lab, you wrote code to do something similar where administrators can navigate to the EditEvent page. However, you wrote that before adding a view model for the Home page. This time we’re going to see how the view model can make navigation decisions. This is slightly challenging because ultimately, navigation operations happen in the view, because the relevant NavigationService and NavigationContext properties belong to the view. We could make the view pass the NavigationService into the view model, but that would make the view model dependent on a view being available in order to run; since one of the main reasons for having a view model in the first place is so that it can be tested in isolation from a real view, this would defeat the purpose. So we’ll take a different approach.
      8. The view model will raise events to tell its hosting view when it wants navigation to occur. This allows the decision-making process to reside in the view model, but for the mechanics of executing navigation to live in the view. By using an event, we minimize the amount that the view model demands of the view. Just as the view model offers properties that it assumes that the view will bind to, it can offer events that it assumes the view will handle.
      9. We’ll need a custom class derived from EventArgs to hold the event data, because this event will need to indicate the ID to be put in the target page.

Create a Navigation Helper EventArg

* 1. Add a class to the Helpers folder of your project and name it NavigateToEntityEventArgs.
  2. Implement it as follows:
     1. C#
     2. public class NavigateToEntityEventArgs : EventArgs
     3. {
     4. public NavigateToEntityEventArgs(int id)
     5. {
     6. EntityID = id;
     7. }
     8. public int EntityID { get; private set; }
     9. }
  3. Open the HomeViewModel class (This is the one that will decide when to navigate to the schedule planner)
  4. Add the **SlEventManager.Helpers** namespace reference to the using statements.
  5. Add the following code to the **HomeViewModel** class:
     1. C#
     2. public event EventHandler<NavigateToEntityEventArgs> NavigateToSchedulePlanner;

Add Commanding

* 1. Provide a command for the button in the HomeViewModel:
     1. C#
     2. private readonly RelayCommand \_plannerCommand;
     3. public ICommand PlannerCommand { get { return \_plannerCommand; } }
  2. Add this method to see whether the user has currently selected an event, and if they have, to raise the navigation event:
     1. C#
     2. private void OnPlanner()
     3. {
     4. if (SelectedEvent != null && NavigateToSchedulePlanner != null)
     5. {
     6. NavigateToSchedulePlanner(this,
     7. new NavigateToEntityEventArgs(SelectedEvent.EventID));
     8. }
     9. }
  3. In the **UpdateRegistrationButtons** method, add the following code to make sure the command is enabled and disabled at the appropriate time:
     1. C#
     2. \_plannerCommand.IsEnabled = \_unregisterCommand.IsEnabled;
  4. In the HomeViewModel constructor, add the following code before the call to **UpdateForUserRole** to create the command object and wire it up to this handler:
     1. C#
     2. \_plannerCommand = new RelayCommand(OnPlanner);
     3. Our home page’s view model now contains the code needed to decide when and where to navigate, but our view needs to do two things: we need a link button to invoke the command, and we need code to handle the navigation event.

Create the Buttons and Bind them to Commands

* 1. In Home.xaml, find the StackPanel that contains the attendee-specific buttons (the User Picture hyperlink, and the Register and Unregister buttons)
  2. Add this XAML inside that panel:
     1. XAML
     2. <HyperlinkButton Command="{Binding Path=PlannerCommand}"
     3. Content="Planner" />
  3. In the Home.xaml.cs code behind go into the constructor and add a handler for the \_viewModel.NavigateToSchedulePlanner event.
  4. Add the following using statement:
     1. C#
     2. using SlEventManager.Helpers;
  5. In this handler, navigate to the entity requested by the view model with the following code:
     1. C#
     2. void \_viewModel\_NavigateToSchedulePlanner(object sender,
     3. NavigateToEntityEventArgs e)
     4. {
     5. NavigationService.Navigate(new Uri("/SchedulePlanner?EventID=" + e.EntityID,
     6. UriKind.Relative));
     7. }
  6. Run the application.
  7. Log in with username of ian and password of P@ssw0rd, and check the box to remain logged in.
  8. In the event list, select an event for which you are registered.
  9. Click the Planner link.
     1. The application should navigate to the new page (which is currently empty). The URL should contain an EventID= item in the query string indicating the ID of the event you selected.
  10. Hit the **Back** button on the browser.
  11. Choose a different event (registering for it if necessary).
  12. Click Planner again and you should see a different event ID.
  13. To verify that the relevant ID is getting through to where we need it, put a breakpoint on the ScheduleEventPlannerViewModel class’s Load method.
  14. Run the application in the debugger.
  15. Verify that this method’s eventId contains the appropriate value whenever you navigate to the planner page for an event.
      1. Now that the navigation is wired up, we need our SchedulePlannerViewModel to provide the view with a list of available talks to populate the grid.

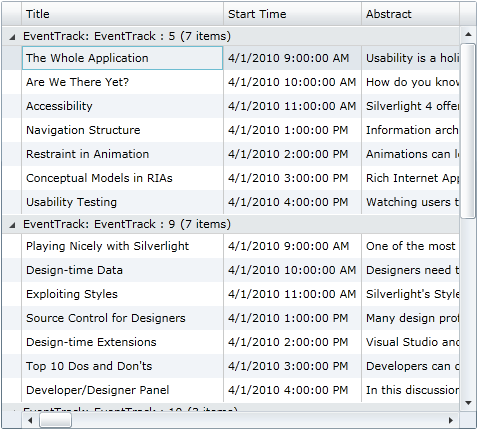
Load the Bound Collections

* 1. Add the following using declarations to the SchedulePlannerViewModel:
     1. C#
     2. using System.Collections.Generic;
     3. using System.Linq;
     4. using SlEventManager.Web;
     5. using SlEventManager.Web.Services;

using System.ServiceModel.DomainServices.Client;

* 1. Add this property:
     1. C#
     2. private IList<Talk> \_allTalks;
     3. public IList<Talk> AllTalks
     4. {
     5. get { return \_allTalks; }
     6. set
     7. {
     8. if (value != \_allTalks)
     9. {
     10. \_allTalks = value;
     11. OnPropertyChanged("AllTalks");
     12. }
     13. }
     14. }
  2. Add the EventTitle property:
     1. C#
     2. private string \_eventTitle;
     3. public string EventTitle
     4. {
     5. get { return \_eventTitle; }
     6. set
     7. {
     8. if (value != \_eventTitle)
     9. {
     10. \_eventTitle = value;
     11. OnPropertyChanged("EventTitle");
     12. }
     13. }
     14. }
  3. Add the following field to the class so we can talk to our service and populate this and the other properties our view model:
     1. C#
     2. private EventManagerDomainContext \_context;
     3. The EventManagerDomainContext is the WCF RIA Services domain context for our class, remember. This enables us to use our service code.
     4. This page needs a complete list of all the tracks and talks associated with an event. We already added a suitable query to the domain service to support the EditEvent page, so our view model can just reuse that same operation.
  4. Go to the (currently empty) Load method and implement it like this:
     1. C#
     2. public void Load(int eventId)
     3. {
     4. \_context = new EventManagerDomainContext();
     5. var eventQuery = from ev in \_context.GetEventsWithTracksAndTalksQuery()
     6. where ev.EventID == eventId
     7. select ev;
     8. \_context.Load(eventQuery, (loadOp) =>
     9. {
     10. var talkQuery = from ev in loadOp.Entities
     11. from track in ev.EventTracks
     12. from talk in track.Talks
     13. select talk;
     14. AllTalks = talkQuery.ToList();
     15. EventTitle = loadOp.Entities.Select(ev => ev.EventTitle).FirstOrDefault();
     16. }, null);
     17. }
     18. This builds a LINQ query to filter the events down to just the event whose schedule is being planned, and then calls the domain context’s Load method to fetch the data. As with all networking operations in Silverlight, this runs asynchronously, so we use a lambda to provide a completion callback. The object representing the operation (called loadOp here) returns the query results in its Entities property—because the query focuses on events, this will be a collection of events (containing just the one event here), so we need another LINQ query to extract all the talk objects.
     19. These two LINQ queries operate quite differently. The first one exploits WCF RIA Service’s ‘composable’ query feature: when a domain service operation returns an IQueryable<T>, a client can perform certain LINQ queries on it in a way that effectively causes the query itself to be sent to, and executed on the server. So the where clause in this code is handled on the server, even though this is client-side code. But the second query here is using normal LINQ to Objects—it runs on the client side, after the server has returned us all the tracks and talks for the event. This is exactly what we want in both cases: in the first case, we only want the server to return us data for a single event, but in the second case, we’re just restructuring the data to suit our view and so it makes sense for that work to happen on the client, because it has enabled us to reuse a domain operation we wrote earlier.
  5. Now that our view model is providing data, open the SchedulePlanner.xaml file.
  6. In the root page element, add the following XAML namespace declarations, which we’ll need to use the data grid:
     1. XAML
     2. xmlns:data="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Data"
     3. xmlns:scm="clr-namespace:System.ComponentModel;assembly=System.Windows"
  7. In the same element, add the following to the XAML to bind the view model to the page:
     1. XAML
     2. DataContext="{Binding}"
  8. As the first child of the root element (directly before the Grid) add the following:
     1. XAML
     2. <navigation:Page.Resources>
     3. <CollectionViewSource x:Key="eventEventTalksByTrackViewSource"
     4. Source="{Binding Path=AllTalks}">
     5. <CollectionViewSource.GroupDescriptions>
     6. <PropertyGroupDescription PropertyName="EventTrack" />
     7. </CollectionViewSource.GroupDescriptions>
     8. </CollectionViewSource>
     9. </navigation:Page.Resources>
     10. This defines a data source that uses your view model’s AllTalks property, but which groups the talks by their EventTrack property.
  9. To display this set of talks, and also the event title, add the following inside the Grid:
     1. XAML
     2. <Grid.ColumnDefinitions>
     3. <ColumnDefinition Width="1\*" />
     4. <ColumnDefinition Width="36" />
     5. <ColumnDefinition Width="1\*" />
     6. </Grid.ColumnDefinitions>
     7. <TextBlock
     8. x:Name="eventTitleTextBlock"
     9. Text="{Binding Path=EventTitle}"
     10. VerticalAlignment="Top"
     11. />
     12. <data:DataGrid
     13. x:Name="eventTrackList"
     14. Margin="12,56,9,12"
     15. ItemsSource="{Binding Source={StaticResource eventEventTalksByTrackViewSource}}"
     16. >
     17. <data:DataGrid.Columns>
     18. <data:DataGridTextColumn x:Name="talkTitleColumn"
     19. Binding="{Binding Path=TalkTitle}"
     20. Header="Title" />
     21. <data:DataGridTextColumn x:Name="talkTimeColumn"
     22. Binding="{Binding Path=TalkStartTime}"
     23. Header="Start Time" />
     24. <data:DataGridTextColumn x:Name="talkAbstractColumn"
     25. Binding="{Binding Path=TalkAbstract}"
     26. Header="Abstract" />
     27. </data:DataGrid.Columns>
     28. </data:DataGrid>
  10. Run the application.
  11. Go to the event scheduler page.

If it doesn’t work and you see a blank grid, make sure you are selecting an event that has tracks. The “Obsessive Compulsive UI” event has a track.

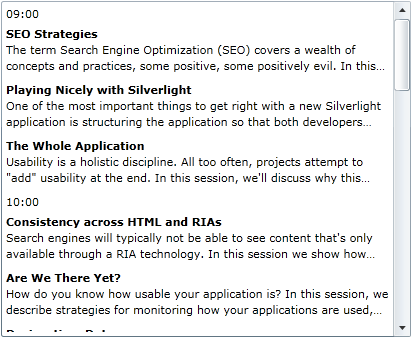
* 1. You should see a grid, showing all the talks grouped by track:
     1. 
     2. Figure 1
     3. Talks and Tracks

Exercise 2: Showing the User’s Session Schedule

Overview

* 1. As well as showing the available talks, the session planner needs to show which talks the user has selected. We’ll use grouping again, but this time we’ll group chosen talks by time slot, so the user can see when they’ve picked multiple talks that clash. (We’re not going to treat that as an error—the user may not yet have decided which talks to go to in certain slots, with the intention of keeping their options open until the day of the event.) We’ll use an ordinary basic items control. The data grid is a good way to pack a lot of information onto the screen, but it’s probably a bit too dense here, and the clarity of a simpler UI component will make it easier for users to see their schedule at a glance.
  2. The second list we’re about to add displays talks, but unlike the grid we added in the previous part, this list will only show those talks the user has added to their schedule. Our view model doesn’t yet provide that information. It will need some help from the service because of the way we’re authenticating users. Our application relies on ASP.NET’s authentication mechanisms under the covers, as wrapped by WCF RIA Services. The AuthenticationService generated by the project template handles this, and makes a User object available on the client side to represent the logged in user. However, that’s part of the set of tables in the database managed by ASP.NET. In order to accommodate user-specific data required by our application, we have our own Account entity.
  3. The client doesn’t have a way to work out which Account corresponds to the logged in user. And that’s probably as it should be. We don’t really want the client-side code to be able to ask for any old Account, and you would probably remove the generated domain service operations that allow the client to manipulate the Account objects directly. The code in earlier labs that manipulated user-specific information (registering and unregistering for events) let the server work out what accounts to use, and it will be the same here. So we’ll need to add a domain service operation to retrieve the set of talks the user has added to their schedule for a given event. This will be very similar to the operation you added in an earlier lab to get the set of events for which the user is registered.

Show User’s Talks

* 1. In the SlEventManager.Web application’s Services folder, open the EventManagerDomainService.cs file.
  2. Add the following operation:
     1. C#
     2. [Invoke]
     3. public IEnumerable<int> FetchTalksUserHasInScheduleForEvent(int eventID)
     4. {
     5. MembershipUser mu = Membership.GetUser();
     6. if (mu == null)
     7. {
     8. return new int[0];
     9. }
     10. var q = from atSchTalk in this.ObjectContext.AttendeeScheduleTalks
     11. where atSchTalk.Attendee.AspNetUserId == (Guid) mu.ProviderUserKey
     12. select atSchTalk.TalkID;
     13. return q;
     14. }
     15. This method just returns the IDs of the talks. That’s all we need here. Our view model already fetched a complete set of Talk entities to populate the grid of available talks, so there’s no sense in writing an operation that returns entities here because we’d end up sending duplicate copies. (That would work: the client-side domain context object would recognize that these were entities it had seen before. But it would be inefficient to transfer multiple copies of the talk entities across the network.)
  3. Add the following using declaration to the **SchedulePlannerViewModel**:
     1. C#
     2. using System.Collections.ObjectModel;
  4. Add the following property so **SchedulePlannerViewModel** can make the chosen talks available for data binding:
     1. C#
     2. private ObservableCollection<Talk> \_subscribedTalks;
     3. public ObservableCollection<Talk> SubscribedTalks
     4. {
     5. get { return \_subscribedTalks; }
     6. set
     7. {
     8. if (value != \_subscribedTalks)
     9. {
     10. \_subscribedTalks = value;
     11. OnPropertyChanged("SubscribedTalks");
     12. }
     13. }
     14. }
     15. We’re using an ObservableCollection here because the contents of the list will change as the user adds or removes talks. (It may seem like overkill to support both list-level notification and also property-level notification. But we need property change notification because this property is null to start with. And then we need list notifications once the list has been created.)
  5. In the Load method, inside the lambda that populates the AllTalks and EventTitle properties, add the following code:
     1. C#
     2. \_context.FetchTalksUserHasInScheduleForEvent(eventId, (talkIdLoadOp) =>
     3. {
     4. var chosenTalks = from talk in AllTalks
     5. where talkIdLoadOp.Value.Contains(talk.TalkID)
     6. select talk;
     7. SubscribedTalks = new ObservableCollection<Talk>(chosenTalks);
     8. }, null);
     9. This retrieves the list of talks in the user’s schedule, using the domain operation you just wrote, and then finds the Talk entities (as previously fetched with the GetEventsWithTracksAndTalksQuery operation) whose IDs are in the list returned.
     10. We’re using a feature added in Silverlight 4 here. The ObservableCollection<T> constructor that accepts an IEnumerable<T> was not previously available.
     11. Now our view model provides the set of subscribed talks, we need our view to show them. Open ScheduleManager.xaml and find the CollectionViewSource you added earlier. That first source shows all the talks, so we need a second source to show just the user’s chosen talks.
  6. Add the following XAML after the first **CollectionViewSource** inside **SchedulePlanner.xaml**:
     1. XAML
     2. <CollectionViewSource x:Key="eventEventTalksByStartTimeViewSource"
     3. Source="{Binding Path=SubscribedTalks}">
     4. <CollectionViewSource.GroupDescriptions>
     5. <PropertyGroupDescription PropertyName="TalkStartTime" />
     6. </CollectionViewSource.GroupDescriptions>
     7. <CollectionViewSource.SortDescriptions>
     8. <scm:SortDescription PropertyName="TalkStartTime" />
     9. </CollectionViewSource.SortDescriptions>
     10. </CollectionViewSource>
  7. Further down in the XAML after the DataGrid, add this XAML:
     1. XAML
     2. <ScrollViewer
     3. Grid.Column="2"
     4. Margin="15,62,12,6"
     5. HorizontalScrollBarVisibility="Disabled"
     6. VerticalScrollBarVisibility="Auto"
     7. >
     8. <ItemsControl
     9. VerticalAlignment="Top"
     10. ItemsSource="{Binding Path=Groups,
     11. Source={StaticResource eventEventTalksByStartTimeViewSource}}"
     12. >
     13. <ItemsControl.ItemTemplate>
     14. <DataTemplate>
     15. <Grid>
     16. <Grid.RowDefinitions>
     17. <RowDefinition Height="20" />
     18. <RowDefinition Height="Auto" />
     19. </Grid.RowDefinitions>
     20. <TextBlock Text="{Binding Path=Name, StringFormat=HH:mm}" />
     21. <ItemsControl
     22. Grid.Row="1"
     23. ItemsSource="{Binding Path=Items}">
     24. <ItemsControl.ItemTemplate>
     25. <DataTemplate>
     26. <Grid
     27. >
     28. <Grid.RowDefinitions>
     29. <RowDefinition Height="Auto" />
     30. <RowDefinition MaxHeight="40" />
     31. </Grid.RowDefinitions>
     32. <TextBlock
     33. Text="{Binding Path=TalkTitle}"
     34. FontWeight="Bold"
     35. />
     36. <TextBlock
     37. Grid.Row="1"
     38. Text="{Binding Path=TalkAbstract}"
     39. TextWrapping="Wrap"
     40. TextTrimming="WordEllipsis"
     41. MaxHeight="40"
     42. >
     43. <ToolTipService.ToolTip>
     44. <TextBlock Text="{Binding Path=TalkAbstract}"
     45. Width="200" TextWrapping="Wrap" />
     46. </ToolTipService.ToolTip>
     47. </TextBlock>
     48. </Grid>
     49. </DataTemplate>
     50. </ItemsControl.ItemTemplate>
     51. </ItemsControl>
     52. </Grid>
     53. </DataTemplate>
     54. </ItemsControl.ItemTemplate>
     55. </ItemsControl>
     56. </ScrollViewer>
     57. The ItemsControl base class does not provide direct support for grouping (and neither, therefore, do the ListBox or ComboBox which derive from it directly). This is not a problem in this example, because we can simply bind the ItemsControl directly to the Groups property provided by any grouping CollectionViewSource. And then inside this we have a nested ItemsControl that binds to the Items property exposed by each group object.
  8. Run the application.
  9. Log in as the user **ian** with a password of **P@ssw0rd**
  10. Go to the planner page for the first event. You should see the following:
      1. 
      2. Figure 3
      3. Planner Page Details
      4. Notice that the abstracts are being truncated to a couple of lines using a feature added in Silverlight 4: the TextTrimming property lets us crop text at a word boundary to make it fit. This XAML sets the TextTrimming property to WordEllipsis
      5. The XAML also includes a ToolTip, enabling the user to see the full description when they move the mouse over the text.

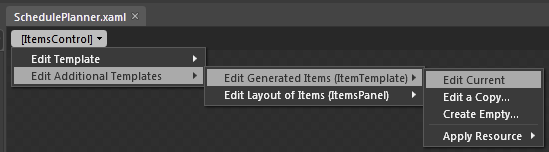
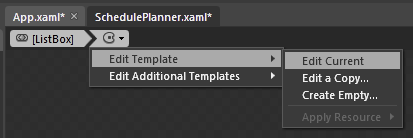
Exercise 3: Adding Sessions to the Schedule

* 1. The schedule planner won’t be much use if all it can do is show an existing plan. Users need to be able to add talks to their planned schedule. So in this part, you’ll add the code to make that happen. You’ll also see how some new visual states added in Silverlight 4 make it possible to run animations as new items are added.

Add Sessions to a Schedule

* 1. In the SchedulePlannerViewModel class, add the following command:
     1. C#
     2. private RelayCommand \_addTalkCommand;
     3. public ICommand AddTalkCommand { get { return \_addTalkCommand; } }
     4. private void OnAddTalk()
     5. {
     6. }
  2. Add the following using statement, if it is not already there:
     1. C#
     2. using System.Windows.Input;
  3. Add a constructor to create the command object and wire it into that method:
     1. C#
     2. public SchedulePlannerViewModel()
     3. {
     4. \_addTalkCommand = new RelayCommand(OnAddTalk);
     5. }
     6. The button that we’ll wire to this command will need to be enabled or disabled based on whether a talk is currently selected in the data grid, and whether that talk is already in the user’s schedule.
  4. Add a property **SelectedTalkInFullList** that can be bound to the grid’s SelectedItem property and add code to update the command status as the selection changes:
     1. C#
     2. private Talk \_selectedTalkInFullList;
     3. public Talk SelectedTalkInFullList
     4. {
     5. get { return \_selectedTalkInFullList; }
     6. set
     7. {
     8. if (value != \_selectedTalkInFullList)
     9. {
     10. \_selectedTalkInFullList = value;
     11. OnPropertyChanged("SelectedTalkInFullList");
     12. UpdateCommandStatus();
     13. }
     14. }
     15. }
     16. private void UpdateCommandStatus()
     17. {
     18. \_addTalkCommand.IsEnabled = SelectedTalkInFullList != null &&
     19. (SubscribedTalks == null ||
     20. !SubscribedTalks.Any(talk => talk.TalkID == SelectedTalkInFullList.TalkID));
     21. }
     22. Before we can implement the OnAddTalk method we need some server-side support. Again, we’re doing work that requires use of the Attendee entity representing the current user, something the client currently doesn’t have access to. While we could arrange for the client to get access to that, and have the client create new associated AttendeeScheduleTalk entities, it’s straightforward and efficient just to add a suitably specialized domain operation.
  5. Add this method to the EventDomainManagerService class in the Services folder of the SlEventManager.Web project:
     1. C#
     2. [Invoke]
     3. public void AddTalkToUserSchedule(int talkID)
     4. {
     5. Attendee attendee = GetOrCreateAttendeeForCurrentUser();
     6. AttendeeScheduleTalk ast = new AttendeeScheduleTalk { TalkID = talkID };
     7. attendee.AttendeeScheduleTalks.Add(ast);
     8. this.ObjectContext.SaveChanges();
     9. }
  6. Add the following code to the OnAddTalk method in the **SchedulePlannerViewModel** class:
     1. C#
     2. private void OnAddTalk()
     3. {
     4. // Take a copy because the update will happen asynchronously, so the
     5. // SelectedTalkInFullList property could change before we're finished.
     6. Talk talkToAdd = SelectedTalkInFullList;
     7. if (talkToAdd != null)
     8. {
     9. \_context.AddTalkToUserSchedule(talkToAdd.TalkID, addOp =>
     10. {
     11. SubscribedTalks.Add(talkToAdd);
     12. UpdateCommandStatus();
     13. }, null);
     14. }
     15. }
     16. This not only calls the new method to add the talk to the schedule on the server side, it also updates the client-side list of subscribed talks. Since that list is an observable collection, Silverlight’s data binding will notice the change and update the items control.
  7. Add the following button to SchedulePlanner.xaml after the DataGrid, so we have a way to invoke the command:
     1. XAML
     2. <Button
     3. Content="&gt;"
     4. Grid.Column="1"
     5. Margin="0,56,0,12"
     6. Command="{Binding Path=AddTalkCommand}"
     7. />
  8. In the DataGrid, add this attribute to connect its selected item to the corresponding property in the view model:
     1. XAML
     2. SelectedItem="{Binding Path=SelectedTalkInFullList, Mode=TwoWay}"
  9. Run the application.
     1. In the planner you should be able to add items to the right by selecting items on the left and clicking the button. Selecting items that are already in the schedule will cause the button to become disabled.
  10. Stop the application.

Adding Fluid UI Features (Optional)

* 1. For this part of the lab, we’ll use the new visual states in Silverlight 4 that are sometimes referred to as ‘fluid UI’ animations, enabling animations to run when items are added to and removed from a list. This is only available on ListBoxItems, so we need to be using a ListBox.
  2. In SchedulePlanner.xaml, find the nested ItemsControl.
     1. Not the one directly inside the ScrollViewer. Find that one’s ItemTemplate and in there is another ItemsControl.
  3. Change the **ItemsControl** into a ListBox.
  4. Also, change the ItemsControl.ItemTemplate element into a ListBox.ItemTemplate element.
     1. Silverlight doesn’t require this. It is happy for you to use the base class name. But Blend gets confused if you don’t.
  5. Add a ScrollViewer.HorizontalScrollBarVisibility="Disabled" attribute to the ListBox.
  6. Open the project in Expression Blend.
  7. Select the ItemsControl that shows the select talks.
     1. If you just click on the relevant item, you will probably end up selecting the containing ScrollViewer rather than the ItemsControl inside the ScrollViewer. Use the Objects and Timeline pane to ensure you’ve selected the ItemsControl.
  8. Using the ‘breadcrumb’ menu at the top left of the editor area, edit the generated items template for the control:
     1. 
     2. Figure 4
     3. Editing the Generated Items Template
     4. Remember that this items control is showing grouped items, so the item template we’re editing right now represents a whole group. (The talks are grouped by time slot.) But we’ll want to run animations any time individual talks are added, rather than when a new group appears.
  9. Find and select the nested ListBox inside the template.
  10. From the Object menu, select Edit Additional Styles → Edit Generated Item Container (ItemContainerStyle) → Edit a Copy…
  11. Click OK in the Create Style Resource dialog. Then use the breadcrumb menus to edit the template for the item container as shown below
      1. 
      2. Figure 5
      3. Edit the Template
  12. In the Objects and Timeline pane, select the Grid inside the Template.
  13. In the Properties pane, set its Opacity to 0%.
  14. Open the States panel.
  15. With the Grid still selected, find the LayoutStates state group.
  16. Select the Loaded state.
  17. Set the opacity back up to 100%.
  18. In the Default transition row for the state group, change the duration from 0s to 0.5s.
  19. Run the application.
  20. Add a talk to the schedule. It should fade into place rather than appearing instantaneously.

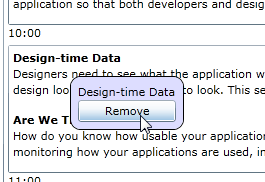
Exercise 4: Removing Sessions from the Schedule

* 1. Finally, we need to provide the user with a way to remove items from the schedule. We’ll do this through a popup UI shown by right-clicking on an item.

Create a Context Menu

* 1. Open the SchedulePlanner.xaml file in Visual Studio.
  2. Near the bottom of the page, find the closing </Grid> tag, and just before that tag, add the following:
     1. XAML
     2. <Canvas
     3. x:Name="popupContainer"
     4. Grid.ColumnSpan="3"
     5. >
     6. <Popup
     7. x:Name="menuPopup"
     8. HorizontalAlignment="Left"
     9. VerticalAlignment="Top"
     10. >
     11. <Border
     12. BorderBrush="Black"
     13. BorderThickness="1"
     14. CornerRadius="10"
     15. Background="#ddf"
     16. Padding="7"
     17. >
     18. <StackPanel>
     19. <TextBlock Text="{Binding Path=TalkTitle}" />
     20. <Button Content="Remove" />
     21. </StackPanel>
     22. </Border>
     23. </Popup>
     24. </Canvas>
     25. The Popup contains the UI elements that will make up our ad-hoc context ‘menu’. The purpose of the containing Canvas is to be able to detect mouse clicks outside of the menu in order to close the menu. (We’ll add event handlers to do that later.)
     26. Notice that we’re using a data binding expression to show the title of a talk. This XAML assumes that the popup’s data context will be set to refer to a Talk entity. We’ll do that when we show the menu.
     27. Further up the XAML file find the <ItemsControl.ItemTemplate> for the second
     28. ItemsControl. (If you did the optional part of the last exercise, you will have changed this to a <ListBox.ItemTemplate>, so find that instead.) Inside this there will be a <DataTemplate> containing a <Grid> element. This element contains all the content representing a single talk in the user’s chosen schedule. We’ll attach the right mouse button handlers to this, but first, we need to make sure that the whole of the Grid is clickable. By default, panels have no value in their Background properties, meaning that mouse input falls through to the element behind.
  3. Set the Background of the Grid inside the ItemTemplate for the innermost ItemsControl to "Transparent".

Add Right Click Support

* 1. Add two event handlers to this Grid element, for the MouseRightButtonDown and MouseRightButtonUp events.
     1. Context menus are normally opened on the right mouse button up event, but Silverlight won’t give us that event unless we marked the down event as handled, so we need both handlers.
  2. In the code behind, find the MouseRightButtonDown handler you just added.
  3. Set the MouseButtonEventArgs (‘e’) argument’s Handled property to true.
  4. Implement the MouseRightButtonUp handler as follows:
     1. C#
     2. private void Grid\_MouseRightButtonUp(object sender, MouseButtonEventArgs e)
     3. {
     4. Point mousePosition = e.GetPosition(popupContainer);
     5. Canvas.SetLeft(menuPopup, mousePosition.X);
     6. Canvas.SetTop(menuPopup, mousePosition.Y);
     7. FrameworkElement itemTemplateInstance = (FrameworkElement) sender;
     8. menuPopup.DataContext = itemTemplateInstance.DataContext;
     9. menuPopup.IsOpen = true;
     10. }
  5. Run the application.
  6. Right-click on an item in the list on the right. You will see the popup appear:
     1. 
     2. Figure 6
     3. Context Menu

Closing the Popup Menu

* + 1. The popup doesn’t go away yet when you click elsewhere. We need to handle clicks outside of the popup. For that, we need to make sure that the containing canvas is able to receive mouse clicks.
  1. Add the following code at the end of the MouseRightButtonUp handler:
     1. C#
     2. popupContainer.Background = new SolidColorBrush(Colors.Transparent);
  2. Back in the XAML, go to the popupContainer Canvas.
  3. Add a single event handler function for left and right mouse button down events, and make it do the following:
     1. C#
     2. popupContainer.Background = null;
     3. menuPopup.IsOpen = false;
  4. Run the application again. You should now be able to dismiss the menu by clicking outside of it within the Silverlight application.
     1. We need to wire our button into some behavior. For this, we’ll need an extra method on the server.

Add Support for Removing a Session

* 1. In the EventManagerDomainService, add the following method:
     1. C#
     2. [Invoke]
     3. public void RemoveTalkFromUserSchedule(int talkID)
     4. {
     5. Attendee attendee = GetOrCreateAttendeeForCurrentUser();
     6. var astQuery = from entry in attendee.AttendeeScheduleTalks
     7. where entry.TalkID == talkID
     8. select entry;
     9. AttendeeScheduleTalk ast = astQuery.SingleOrDefault();
     10. if (ast != null)
     11. {
     12. this.ObjectContext.DeleteObject(ast);
     13. this.ObjectContext.SaveChanges();
     14. }
     15. }
  2. Open the SchedulePlannerViewModel class.
     1. This time we won’t be adding a command because the menu XAML doesn’t have direct access to the view model; its data context is a Talk entity. (It would be reasonable to add a separate per-item view model for each talk in the list and make that the source instead, but for now we’re going to take a short cut.)
  3. Add a method named **RemoveTalk**:
     1. C#
     2. public void RemoveTalk(Talk talk)
     3. {
     4. \_context.RemoveTalkFromUserSchedule(talk.TalkID, removeOp =>
     5. {
     6. SubscribedTalks.Remove(talk);
     7. UpdateCommandStatus();
     8. }, null);
     9. }
  4. In the XAML, find the Button for your menu.
  5. Add a Click event handler.
  6. In the code behind add the following using directive:
     1. C#
     2. using SlEventManager.Web;
  7. Implement the **Click** handler like this:
     1. C#
     2. private void Button\_Click(object sender, RoutedEventArgs e)
     3. {
     4. Button btn = (Button) sender;
     5. Talk talk = (Talk) btn.DataContext;
     6. \_viewModel.RemoveTalk(talk);
     7. }
  8. Run the application. You should now find you can remove items from the schedule by right-clicking and then clicking the Remove button.
  9. Add a line of code inside the **Button\_Click** handler to hide the context menu. (Use the same code used in the popupContainer mouse button down handling to hide the menu.)