* 1. 

Hands-On Lab

Building Applications in Silverlight 4

Module 7: Event Administrator Dashboard with Out of Browser, Toasts and Native Integration

**Contents**

[Introduction 3](#_Toc256194225)

[Exercise 1: Adding an Out of Browser Application 4](#_Toc256194226)

[Add a New Application 4](#_Toc256194227)

[Configure the New Project 5](#_Toc256194228)

[Configure for Out-of-Browser 6](#_Toc256194229)

[Setting up for Debugging OOB 8](#_Toc256194230)

[Check for Updates 10](#_Toc256194231)

[Wire the UI to the Domain Service 11](#_Toc256194232)

[Loading the Dashboard 14](#_Toc256194233)

[Adding ViewModels 17](#_Toc256194234)

[Binding the ViewModels to the Views 19](#_Toc256194235)

[Exercise 2: “Toast” Notifications 20](#_Toc256194236)

[Creating the Toast Content 20](#_Toc256194237)

[Pop up the Toast 21](#_Toc256194238)

[Check for Notifications 23](#_Toc256194239)

[Exercise 3: Window Handling 26](#_Toc256194240)

[Managing the Windows 26](#_Toc256194241)

[Handling Focus 26](#_Toc256194242)

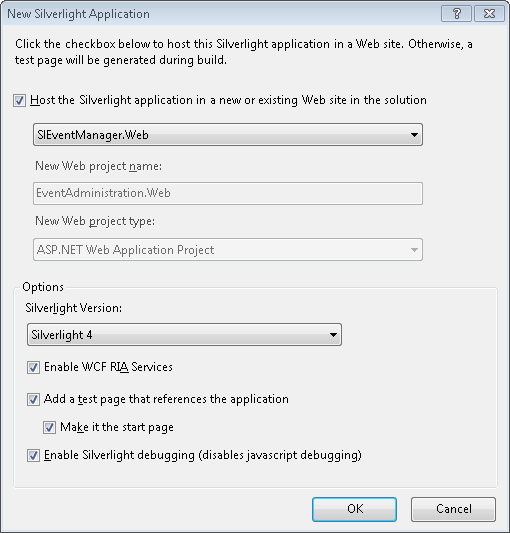
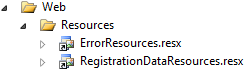
Introduction

* 1. In this lab, you will build a new Silverlight project that adds an additional administrative capability to the event manager application. It will provide administrators with a view of which attendees have currently registered for events, but who have not yet been sent emails confirming their registration.
  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.

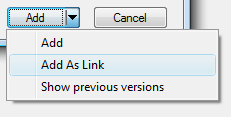
Exercise 1: Adding an Out of Browser Application

* 1. In this part of the lab, you will add a second Silverlight application to the solution. The event administration application needs to run with elevated privileges to be able to use certain Silverlight features, which in turn means it needs to run out of browser. We want to leave the end-user-facing part of the application as it is, and while it’s possible to write a single Silverlight application that can run both inside and outside the browser (and you can show different UIs in each if necessary), there’s no real benefit to doing that in this case because the two parts of the application do quite different things. There’s no point in making the user download a larger .xap than necessary, so splitting the application in two makes sense here.

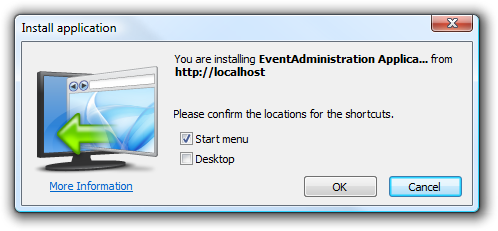
Add a New Application

* 1. Open the SlEventManager solution in Visual Studio 2010.
  2. Add a new Silverlight Application project to the solution called EventAdministration.
  3. Host it in the existing SlEventManager.Web solution
  4. Enable use of WCF RIA Services
  5. Allow Visual Studio to generate a new test page (setting it as the new start page), with debugging enabled.
     1. 
  6. Figure 1
  7. Adding a New Silverlight Application
     1. If you build the solution right now, you’ll get compilation errors. This is because the original projects were built with the Silverlight Business Application template, which relies on building certain files into both client and server. (In general, linking multiple Silverlight projects to a single web project doesn’t cause problems. The problems here are due to how this particular template uses RIA Services.) If you look at the original SlEventManager project and expand its Web\Resources folder, you’ll see some linked files
     2. 
  8. Figure 2
  9. Linked Files
     1. The little arrow overlay icon indicates that these files (and also the Designer.cs files you’ll see if you expand the icons) are references to files that live elsewhere, rather than belonging to this project. If you select one of them and look at the Properties panel, you’ll see that the FullPath property refers to a folder in the SlEventManager.Web project—it has a Resources folder that contains these files—the icons you see in the SlEventManager project are just links pointing to those files. The types generated as a result of the RIA Services link depend on these resources being present in both projects. (This enables messages such as validation errors to be shared across the client and server code.)
     2. There’s an unfortunate problem we need to work around here: resource sharing between the server and client works with the Silverlight Business Application template presumes a common namespace—in our case, both the original projects shared the name SlEventManager, with the web project just adding .Web on the end. By linking to the shared resource files in a Web subfolder from the client SlEventManager project, the resources end up in the SlEventManager.Web namespace in both cases. But our new project has a default namespace of EventAdministration, which means that if we tried to link the files in in the same way, the resources would end up in a different namespace: EventAdministration.Web. This will break the shared code. The simplest solution to this is to modify the project default namespace.

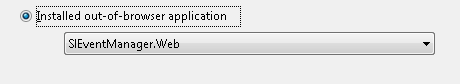
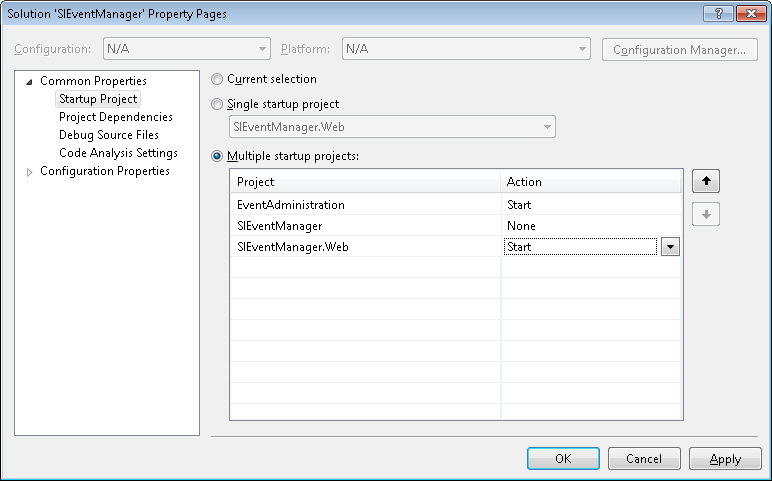
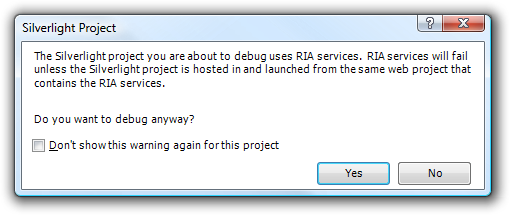
Configure the New Project

* 1. Go to the new project’s Properties page and to the Silverlight tab.
  2. Set the Default namespace to SlEventManager. (You can leave the assembly name as it is, and of course you can’t change the project name, because you already have a project called SlEventManager.)
  3. Add similar links in your new project by first going to the new EventAdministration project and creating a Web folder with a Resources subfolder.
  4. Use Add Existing Item to add *links* to the **ErrorResources**.resx and **RegistrationDataResources.resx** files in the web project. (Do **not** add links to the associated .Designer.cs files.)
     1. To add the file as a link, click on the dropdown arrow at the right of the Add button in the Add Existing Item dialog, and choose the Add As Link menu item. Otherwise, it will make a copy of the files, which makes it possible for the two projects to get out of sync.
     2. 
  5. Figure 3
  6. Add Linked Files
     1. You now need to configure these two .resx files to be associated with the generated .Designer.cs files. This isn’t quite as straightforward as adding links to those files.
  7. For each .resx file, go to the Properties panel and set the Custom Tool property to PublicResXFileCodeGenerator. When
     1. When you do that, Visual Studio will realize that it needs to link to the relevant generated file, and your .resx file in Solution Explorer should now contain the relevant generated file. The project should now build.
  8. Go into App.xaml.cs and change the EventAdministration namespace near the top to SlEventManager by choosing Refactor and Rename from the menu bar.
  9. Say yes when Visual Studio asks if you want to change the namespace across the entire project. (This will also update the MainPage.xaml and MainPage.xaml.cs files for you.)
  10. Go to the project properties page.
  11. Change the startup object from to EventAdministration.App to SlEventManager.App.

Configure for Out-of-Browser

* + 1. Our administration application won’t be able to run at all in-browser, so the next thing we need to do is to add some UI that detects when the user has tried to open the application in the browser. If the application’s not already installed, we can offer to install it for out-of-browser use, but otherwise, we just need to direct the user to run it as a normal app
  1. Add two user controls to the project, one called WebUiNotInstalled and one called WebUiInstalled.
  2. Add the following content to WebUiInstalled:
     1. XAML
     2. <TextBlock TextWrapping="Wrap">
     3. This application cannot run inside the web browser. It is
     4. already installed on your computer, so please run it in the usual way.
     5. </TextBlock>
  3. For the WebUiNotInstalled, we need to give the user the option to install the application. Add the following XAML inside the Grid:
     1. XAML
     2. <Grid.RowDefinitions>
     3. <RowDefinition />
     4. <RowDefinition Height="Auto" />
     5. </Grid.RowDefinitions>
     6. <TextBlock TextWrapping="Wrap">
     7. This application cannot run inside the web browser. To install it on
     8. your computer, please click the Install... button.
     9. </TextBlock>
     10. <Button
     11. x:Name="installButton"
     12. Grid.Row="1"
     13. Content="Install..."
     14. />
  4. Add a Click handler to the button. Inside this handler, kick off the installation of the application with the following code:
     1. C#
     2. Application.Current.Install();
  5. Next, the MainPage.xaml.cs codebehind needs to decide which of these two UIs to show. In the constructor, after the call to InitializeComponent, add this code:
     1. C#
     2. if (!Application.Current.IsRunningOutOfBrowser)
     3. {
     4. if (Application.Current.InstallState == InstallState.Installed)
     5. {
     6. LayoutRoot.Children.Add(new WebUiInstalled());
     7. }
     8. else
     9. {
     10. LayoutRoot.Children.Add(new WebUiNotInstalled());
     11. }
     12. }
  6. Open the **EventAdministration** project’s property pages, and in the Silverlight tab, check the Enable running application out of the browser button.
     1. 
  7. Figure 4
  8. Enabling Out-of-Browser
  9. In the SlEventManager.Web project in the Solution Explorer, right-click on EventAdministrationTestPage.html and select Set as Start Page.
  10. Run the application.
  11. You should see the UI that offers to install the application. Click the Install… button. You should see the usual confirmation dialog:
      1. 
  12. Figure 5
  13. Prompt to Install Out-of-Browser Application
  14. Check the **Desktop** checkbox and click OK. You’ll then see the application launch out-of-browser with an empty window. It’s empty because so far, MainPage.xaml doesn’t load any UI at all if we’re running out of browser.
  15. Close both the OOB window and the browser window.
  16. Run the application again, and this time you should see the UI that tells you the application is already installed.
  17. Close the browser.
  18. Try running from the Windows Start menu or the Desktop where you installed it. Again you should see an empty window.

Setting up for Debugging OOB

* 1. We have a problem now: running the application from within Visual Studio will always show the web UI that tells us we can’t run the application from the web. This is going to make meaningful debugging of the application difficult. Now that we’ve got it installed, we need to modify the way we launch the application for debugging in Visual Studio.
  2. Open the EventAdministration project’s property pages and go to the Debug tab.
  3. Select the Installed out-of-browser application radio button.
     1. 
  4. Figure 6
  5. Setting Up Debugging for Out-of-Browser
     1. Normally you’d then set the EventAdministration project as the startup project, and you’d be able to debug the application purely in out-of-browser mode. However, this particular application won’t work without the web server project also running—unlike some OOB apps, this one cannot usefully run offline.
  6. Right click on the solution and choose Properties.
  7. In the Common Properties → Startup Project section, choose Multiple startup projects, and configure both EventAdministration and SlEventManager.Web to run.
     1. 
  8. Figure 7
  9. Multiple Startup Projects
  10. Run the application again, and you should now find that Visual Studio launches both the web browser and the out-of-browser windows. The first time you do that, you’ll see this dialog:
      1. 
  11. Figure 7
  12. Warning Dialog
  13. This dialog appears when you debug a project with the RIA Services Link enabled. Check the checkbox and click Yes, because in fact you are running the web project as well as the Silverlight application.
      1. The dialog doesn’t take into account out-of-browser RIA Services scenarios. The debugger will attach itself to both Silverlight applications (and also to the web application).
      2. When you use out-of-browser debugging support, you don’t need to worry about deploying updates to the application each time you build. Visual Studio will always debug the most recently built version.
  14. Add another user control to the EventAdministration project, called OobUi. This will be the main out of browser user interface.
  15. Add a TextBlock in there with the text “OOB” so that this control is distinct.
  16. In the MainPage.xaml.cs, modify the code that decides which UI to show, so that it show this new OobUi control when we’re out of the browser:
      1. C#
      2. if (!Application.Current.IsRunningOutOfBrowser)
      3. {
      4. if (Application.Current.InstallState == InstallState.Installed)
      5. {
      6. LayoutRoot.Children.Add(new WebUiInstalled());
      7. }
      8. else
      9. {
      10. LayoutRoot.Children.Add(new WebUiNotInstalled());
      11. }
      12. }
      13. else
      14. {
      15. LayoutRoot.Children.Add(new OobUi());
      16. }
  17. Run the application again and confirm that you see the OobUi when you run out of browser.
      1. Because you need to run the web project and the Silverlight project, both in-browser and out-of-browser versions of your application will be running. Since the debugger attaches to both, debugging this kind of code can occasionally be surprising. If you put breakpoints in this code to follow it through, you need to remember that you’ll be debugging two processes simultaneously taking different paths through the same code.

Check for Updates

* 1. The next few steps will alert the user when an update is available for the application.
  2. Open **OobUi.xaml.cs** and add assign an event handler for the **Application.Current.CheckAndDownloadUpdatedCompleted** event inside the **OobUi** constructor.
     1. C#
     2. Application.Current.CheckAndDownloadUpdateCompleted +=
     3. new CheckAndDownloadUpdateCompletedEventHandler(
     4. Current\_CheckAndDownloadUpdateCompleted);
  3. Add the following method call to check for updates, in the **OobUi** constructor:
     1. C#
     2. Application.Current.CheckAndDownloadUpdateAsync();
  4. Add the event handler that tells the user if an update is available:
     1. C#
     2. void Current\_CheckAndDownloadUpdateCompleted(
     3. object sender, CheckAndDownloadUpdateCompletedEventArgs e)
     4. {
     5. if (e.UpdateAvailable)
     6. {
     7. MessageBox.Show("An update is available for this application. Please close the application and restart it.");
     8. }
     9. }

Wire the UI to the Domain Service

* 1. Now that our application is choosing the right UI to show based on how it is launched, we need to start wiring the OobUi into our domain service. To begin with, we need to log in, so we need the user to provide credentials. You might be tempted to copy across the login UI code from the SlEventManager project. However, doing so requires copying substantial amounts of infrastructure code over, so it’s easier just to build our own login UI.
  2. Add a new Silverlight Child Window to the EventAdministration project, called LoginWindow.
  3. Replace the content of the window’s Grid with this
     1. XAML
     2. <Grid.RowDefinitions>
     3. <RowDefinition Height="Auto" />
     4. <RowDefinition Height="Auto" />
     5. <RowDefinition Height="Auto" />
     6. <RowDefinition Height="Auto" />
     7. </Grid.RowDefinitions>
     8. <Grid.ColumnDefinitions>
     9. <ColumnDefinition Width="Auto" />
     10. <ColumnDefinition Width="200" />
     11. </Grid.ColumnDefinitions>
     12. <Border
     13. x:Name="invalidCredentials"
     14. Visibility="Collapsed"
     15. Grid.ColumnSpan="2"
     16. Padding="3"
     17. Margin="3"
     18. HorizontalAlignment="Center"
     19. Background="Red">
     20. <TextBlock
     21. Foreground="White"
     22. FontWeight="Bold"
     23. Text="Either the user name or password is incorrect"
     24. />
     25. </Border>
     26. <TextBlock
     27. Text="User name:"
     28. Grid.Row="1"
     29. Margin="3"
     30. VerticalAlignment="Center"
     31. />
     32. <TextBox
     33. x:Name="userNameText"
     34. Grid.Row="1"
     35. Grid.Column="1"
     36. Margin="3"
     37. />
     38. <TextBlock
     39. Text="Password:"
     40. Grid.Row="2"
     41. Margin="3"
     42. VerticalAlignment="Center"
     43. />
     44. <PasswordBox
     45. x:Name="passwordText"
     46. Grid.Column="2"
     47. Grid.Row="2"
     48. Margin="3"
     49. />
     50. <StackPanel
     51. Grid.Row="3"
     52. Grid.ColumnSpan="2"
     53. Orientation="Horizontal"
     54. HorizontalAlignment="Right"
     55. >
     56. <Button
     57. x:Name="OkButton"
     58. Content="OK"
     59. MinWidth="75"
     60. Margin="10"
     61. Click="OKButton\_Click"
     62. />
     63. <Button
     64. x:Name="CancelButton"
     65. Content="Cancel"
     66. MinWidth="75"
     67. Margin="10"
     68. Click="CancelButton\_Click"
     69. />
     70. </StackPanel>
  4. Delete the Width and Height properties of the **LoginWindow** and set the **Title** to **Login**.
  5. In the **LoginWindow.xaml.cs** codebehind, add this namespace declaration:
     1. C#

using System.ServiceModel.DomainServices.Client.ApplicationServices;

* 1. Then replace the body of the OKButton\_Click handler method with this:
     1. C#
     2. WebContext.Current.Authentication.Login(
     3. new LoginParameters(userNameText.Text, passwordText.Password),
     4. loginOperation =>
     5. {
     6. if (loginOperation.LoginSuccess)
     7. {
     8. this.DialogResult = true;
     9. }
     10. else
     11. {
     12. invalidCredentials.Visibility = Visibility.Visible;
     13. }
     14. }, null);
     15. This attempts to log the user in, and if it fails, shows an error, and if it succeeds, closes the login child window. For this to work, that WCF RIA Services need to know how it’s supposed to be authenticating.
  2. In the App.xaml for the OOB project, add the following after the closing Application.Resources tag (but before the closing Application tag) so
     1. XAML
     2. <Application.ApplicationLifetimeObjects>
     3. <app:WebContext>
     4. <app:WebContext.Authentication>
     5. <appsvc:FormsAuthentication/>
     6. </app:WebContext.Authentication>
     7. </app:WebContext>
     8. </Application.ApplicationLifetimeObjects>
  3. You’ll need to add these namespace declarations to the root element:
     1. XAML
     2. xmlns:appsvc="clr-namespace:System.ServiceModel.DomainServices.Client.ApplicationServices;assembly=System.ServiceModel.DomainServices.Client.Web"
     3. xmlns:app="clr-namespace:SlEventManager"
  4. Go to OobUi.xaml and add a button with Content="Login" and add a Click handler. Add this code in the handler
     1. C#
     2. LoginWindow w = new LoginWindow();
     3. w.Show();
  5. Run the application.
  6. Click the Login button. The login UI should appear much as it does in the web-hosted Silverlight application from earlier labs. Try logging in with invalid credentials. The UI should show a message saying “The username or password is incorrect”. (This might be quite slow first time you try it.)
  7. Try logging in with correct credentials. (Use “administrator” and “P@ssw0rd”.) The login should succeed, verifying that the credentials are being correctly validated.
     1. There are two additional steps you could do if you’d like to improve the user experience. Neither of these is necessary for the rest of the lab, but would make the application better.
     2. You could add code to change the login button to a logout one once you’ve logged in—you can log out by calling the WebContext.Current.AuthenticationService.Logout method.
     3. You could add a BusyIndicator to the login control, and make it visible while waiting for a response from the service, so that the user knows something is happening. The BusyIndicator control is in the Silverlight toolkit. (In the November 2009 toolkit, it’s in the ‘experimental’ band, so you’ll need to add the relevant pieces to the project yourself—you won’t find it in the Visual Studio Toolbox.)

Loading the Dashboard

* 1. Now that we can authenticate with our server, let’s add some code to show the information this administrator dashboard needs to display. We need an extra service operation to retrieve a list of attendee registrations that we’ve not yet acknowledged.
  2. Add the following method to the EventManagerDomainService in the SlEventManager.Web project’s Services folder:
     1. XAML
     2. [RequiresRole("Event Administrators")]
     3. public IQueryable<AttendeeEvent> GetUnacknowledgedAttendeeEventsWithEvents()
     4. {
     5. return from attendeeEvent in this.ObjectContext.AttendeeEvents.Include("Event")
     6. where !attendeeEvent.IsAcknowledged
     7. select attendeeEvent;
     8. }
  3. In the EventManagerDomainService.metadata.cs file, find the AttendeeEventMetadata class and add an [Include] attribute to the Event field.
     1. This service operation will enable the client to discover when attendees have registered for an event and their registration has not yet been acknowledged. However, the information about the user’s name and email lives elsewhere: we’ve been using the ASP.NET Membership and Profile handling for that. So we need to add more to the service to make that accessible to the client.
  4. Add a new class to the web project’s Models folder called UserDisplayDetails. Add this using declaration:
     1. C#
     2. using System.ComponentModel.DataAnnotations;
  5. implement the class as follows:
     1. C#
     2. public class UserDisplayDetails
     3. {
     4. [Key]
     5. public int AttendeeID { get; set; }
     6. public string FriendlyName { get; set; }
     7. public string Email { get; set; }
     8. }
     9. The Key attribute is required when returning custom types from a WCF RIA Service. It is used to determine the logical identity of an object. (Multiple service calls may end up returning the same object, and a key makes it possible to reconcile these back to the same instance on the client.) Without this, we would get an error if we tried to return objects of this type.
  6. Add the following method to EventManagerDomainService, which returns the display details for the requested attendee IDs:
     1. C#
     2. [RequiresRole("Event Administrators")]
     3. public IEnumerable<UserDisplayDetails> GetUserDisplayDetails(int[] attendeeIds)
     4. {
     5. var dbAttendeeQuery = from attendee in this.ObjectContext.Attendees
     6. where attendeeIds.Contains(attendee.AttendeeID)
     7. select new
     8. {
     9. attendee.AttendeeID,
     10. attendee.AspNetUserId
     11. };
     12. var allUsers = Membership.GetAllUsers().OfType<MembershipUser>();
     13. var results = from attendeeInfo in dbAttendeeQuery.ToList()
     14. let aspNetUser = allUsers.FirstOrDefault(mu =>
     15. ((Guid) mu.ProviderUserKey) == attendeeInfo.AspNetUserId)
     16. where aspNetUser != null
     17. let prof = ProfileBase.Create(aspNetUser.UserName)
     18. where prof != null
     19. select new UserDisplayDetails
     20. {
     21. AttendeeID = attendeeInfo.AttendeeID,
     22. Email = aspNetUser.Email,
     23. FriendlyName = prof.GetPropertyValue("FriendlyName") as string
     24. };
     25. return results;
     26. }
  7. You’ll need to add this using directives to the file for this to compile:
     1. C#
     2. using System.Web.Profile;
     3. using SlEventManager.Web.Models;
  8. Add a DataGrid to the OobUI.xaml file called unacknowledgedEvents.
  9. Set its AutoGenerateColumns property to True.
  10. Add another button labeled “Get”, and add a Click handler.
  11. In the code behind, add this namespace directive:
      1. C#
      2. using SlEventManager.Web.Services;
  12. Then in the “**Get”** button’s Click handler, add the following code to fetch the list of unacknowledged event registrations, and then to fetch the name and email details for all the attendees in question:
      1. C#
      2. EventManagerDomainContext ctx = new EventManagerDomainContext();
      3. ctx.Load(ctx.GetUnacknowledgedAttendeeEventsWithEventsQuery(), loadUnackOp =>
      4. {
      5. int[] attendeeIds = (from atev in loadUnackOp.Entities
      6. select atev.AttendeeID).Distinct().ToArray();
      7. ctx.Load(ctx.GetUserDisplayDetailsQuery(attendeeIds), loadDetailsOp =>
      8. {
      9. var all = loadDetailsOp.Entities.ToArray();
      10. unacknowledgedEvents.ItemsSource = all;
      11. }, null);
      12. }, null);
  13. Run the application.
  14. Log in as “administrator” (password “P@ssw0rd”).
  15. Click the Get button, and the grid should be populated with the name and email of any attendee registered for an event who has not yet received an acknowledgement email. (This will probably be just the “ian” user, unless you’ve registered more users in your application.)
  16. Close the application.

Adding ViewModels

* 1. The information being displayed isn’t quite what we need. We also need to show the event for which the user is registered. We’ll add a per item view model class for this.
  2. In the EventAdministration project, add a ViewModels folder.
  3. Create a UnacknowledgedRegistrationViewModel class in the **ViewModels** folder.
  4. Add the following properties to the View Model:
     1. C#
     2. public class UnacknowledgedRegistrationViewModel
     3. {
     4. public string EventTitle { get; set; }
     5. public string UserDisplayName { get; set; }
     6. public string UserEmail { get; set; }
     7. }
  5. That will be the view model class used to represent individual items in the list. We should also add a view model for the whole OobUi.
  6. In the ViewModels folder, add an OobUiViewModel class.
  7. Copy the **ViewModelBase** class from the SlEventManager project into this project, and make it the base class of OobUiViewModel.
  8. Add the following using declarations:
     1. C#
     2. using System.Collections.Generic;
     3. using System.Linq;
     4. using SlEventManager.Web.Services;
  9. The view model will need to provide a collection to act as the data grid’s source:
     1. C#
     2. private IList<UnacknowledgedRegistrationViewModel> \_unacknowledgedRegistrations;
     3. public IList<UnacknowledgedRegistrationViewModel> UnacknowledgedRegistrations
     4. {
     5. get { return \_unacknowledgedRegistrations; }
     6. set
     7. {
     8. if (\_unacknowledgedRegistrations != value)
     9. {
     10. \_unacknowledgedRegistrations = value;
     11. OnPropertyChanged("UnacknowledgedRegistrations");
     12. }
     13. }
     14. }
  10. Add the following field, since we will need a domain context to use the service:
      1. C#
      2. private EventManagerDomainContext ctx = new EventManagerDomainContext();
  11. Add an OnGet method that will handle the click.
      1. This will be similar to the click handler you wrote earlier, but this time, rather than just showing the user names, it will generate an item for each event registration. (Individual users may register for multiple events.) And it builds an instance of the item view model for each row, providing the exact details we wish to show:
      2. C#
      3. public void OnGet()
      4. {
      5. ctx.Load(ctx.GetUnacknowledgedAttendeeEventsWithEventsQuery(), loadUnackOp =>
      6. {
      7. int[] attendeeIds = (from atev in loadUnackOp.Entities
      8. select atev.AttendeeID).Distinct().ToArray();
      9. ctx.Load(ctx.GetUserDisplayDetailsQuery(attendeeIds), loadDetailsOp =>
      10. {
      11. var attendeeDetails = loadDetailsOp.Entities.ToDictionary(d => d.AttendeeID);
      12. var registrations = from atev in loadUnackOp.Entities
      13. let details = attendeeDetails[atev.AttendeeID]
      14. select new UnacknowledgedRegistrationViewModel
      15. {
      16. EventTitle = atev.Event.EventTitle,
      17. UserDisplayName = details.FriendlyName,
      18. UserEmail = details.Email
      19. };
      20. UnacknowledgedRegistrations = registrations.ToList();
      21. }, null);
      22. }, null);
      23. }

Binding the ViewModels to the Views

* 1. In the OobUi.xaml.cs code behind, add a field declaring and constructing an instance of OobUiViewModel.
  2. Add the following using statement so you can reference the View Model.
     1. C#
     2. using SlEventManager.ViewModels;
  3. In the constructor, put a reference to the view model in the DataContext.
  4. Replace the entire Get button click handler with a call to the view model’s OnGet method.
  5. Finally, in OobUi.xaml, bind the DataGrid control’s ItemsSource to the UnacknowledgedRegistrations property of the view model:
     1. C#
     2. ItemsSource="{Binding Path=UnacknowledgedRegistrations}"
  6. Run the application, log in as administrator (P@ssw0rd), and click the Get button. You should now see a list showing event titles, names, and email addresses. There will be at least two entries because the “Ian Griffiths” user has registered for two events.

Exercise 2: “Toast” Notifications

* 1. In this next stage, we’ll arrange for the application to show “toast” style notifications: small popup windows that appear briefly at the bottom right of the screen. Our administrator dashboard can provide these each time we get any new registrations needing to be acknowledged.
  2. Silverlight 4 does not add any new feature aimed at arranging for this sort of notification to be passed from the server to the client. The focus of this part of the lab is simply the UI mechanism for displaying the notification to the end user. As for the architectural question of how the client knows when to do this, we’ll be using two distinctly lo-tech solutions. To start with we’ll just have a button to show it manually on demand, because that makes it easy to test. Then we’ll make the application poll the server. Every few seconds it will call the operation used in the previous section to fetch the list of unacknowledged items, and when new items appear, we’ll raise a notification.
  3. In general, having a client poll a server every few seconds is not ideal because it does not scale well. In this particular application, we can get away with it because only event administrators will be using this feature, so the feature only needs to scale to however many administrators we have. We probably only have a handful, or maybe even just the one.

Creating the Toast Content

* 1. Add a user control called NotificationContent to your EventAdministration project.
     1. We will set a notification window’s content to this user control.
  2. Set its Width and Height to 150 and 90 respectively.
  3. Remove the design-time width and height. These are hints to the design surface for elements that determine their size at runtime.
  4. Put the following content inside the Grid:
     1. XAML
     2. <Border BorderBrush="Navy" BorderThickness="1">
     3. <Border.Background>
     4. <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
     5. <GradientStop Color="#DEF" />
     6. <GradientStop Offset="1" Color="#FFF" />
     7. </LinearGradientBrush>
     8. </Border.Background>
     9. </Border>
     10. <TextBlock
     11. Text="New Subscribers"
     12. HorizontalAlignment="Center"
     13. VerticalAlignment="Top"
     14. FontWeight="Bold"
     15. FontSize="14"
     16. Margin="0,3"
     17. />
     18. <StackPanel
     19. Orientation="Horizontal"
     20. HorizontalAlignment="Center"
     21. VerticalAlignment="Center"
     22. >
     23. <TextBlock Text="{Binding Path=.}" />
     24. <TextBlock Text=" unprocessed" />
     25. </StackPanel>
  5. Note that one of the text blocks has its text data-bound to “.” so it will display whatever we put in the DataContext.
  6. In OobUi.xaml.cs, add the following:
     1. C#
     2. NotificationWindow nw = new NotificationWindow();
     3. void ShowNotification(int count)
     4. {
     5. if (nw.Visibility == System.Windows.Visibility.Visible)
     6. {
     7. nw.Close();
     8. }
     9. nw.Content = new NotificationContent();
     10. nw.Content.DataContext = count;
     11. nw.Width = nw.Content.Width + 2;
     12. nw.Height = nw.Content.Height + 2;
     13. nw.Show(5000);
     14. }
     15. This closes the notification window if it’s already visible. We need to do this because trying to show the window when it’s already open will cause an exception.
     16. Notice that this code also sets the Width and Height properties to match those of the content. NotificationWindow will not automatically size to content. (Also, be aware that there are size restrictions. If you decide you’d like to use a more outlandish design than the one offered here, you’re limited to 400 pixels wide by 100 pixels high.) We add in 2 pixels of space because the NotificationWindow provides a 1 pixel border which uses some of the space we provide. (So that’s two pixels horizontally—one on either side, and likewise vertically.)
  7. To test this, add a button to OobUi.xaml.cs. In the click handler call the ShowNotification method passing in any number.
  8. Run the application and verify that the notification window appears when the button is pressed.

Pop up the Toast

* 1. Next, we’ll put code in the view model to raise the notification when it needs to be raised.
  2. Open OobUiViewModel and add this using directive:
     1. C#
     2. using System.Windows.Threading;
  3. then add this timer code:
     1. C#
     2. private DispatcherTimer dt;
     3. private Action<int> showNotification;
     4. public void Load(Action<int> notifyCallback)
     5. {
     6. showNotification = notifyCallback;
     7. if (dt == null)
     8. {
     9. dt = new DispatcherTimer();
     10. dt.Tick += NotifyTick;
     11. dt.Interval = TimeSpan.FromSeconds(10);
     12. }
     13. dt.Start();
     14. }
     15. void NotifyTick(object sender, EventArgs e)
     16. {
     17. if (showNotification != null)
     18. {
     19. showNotification(DateTime.Now.Millisecond % 11);
     20. }
     21. }
     22. public void Unload()
     23. {
     24. dt.Stop();
     25. showNotification = null;
     26. }
     27. This sets up a timer to fire every 10 seconds, and calls a notification callback with a meaningless number. (We’ll put a real number in shortly.)
  4. In the OobUi.xaml.cs code behind, add code in the constructor to hook up a handler to the Loaded event for **OobUi**.
  5. Have the handler call the view model’s Load method, passing in the ShowNotification function to enable the view model to show the notification window:
     1. C#
     2. this.Loaded += new RoutedEventHandler(OobUi\_Loaded);
     3. ...
     4. void OobUi\_Loaded(object sender, RoutedEventArgs e)
     5. {
     6. viewModel.Load(ShowNotification);
     7. }
     8. Although we wrote an Unload method for the view model, there’s nowhere we can call that here. Silverlight doesn’t offer an Unloaded event. In fact it’s not a problem because our OobUi view remains running for as long as the application runs. If we were to change this administration application to use navigation (like the SlEventManager application), we would have to call the Unload method in the handler for navigation away from the page. (And we’d call Load in the method that navigates to the page.)
  6. Run the application. Every 10 seconds, a notification window should appear for 5 seconds.

Check for Notifications

* 1. Next we need to do two things. First, we should check for new data in the timer tick, and second, we should show the notification only if something has changed.
  2. In **OobUiViewModel**, replace the body of NotifyTick with a call to OnGet.
     1. This will fetch the data again. However, we’ll now run headlong into a problem that we were previously ignoring: the code to fetch the list fails if we try to run it before logging in, and does so most ungracefully. It will take out the entire application without an error. (The default behavior for unhandled exceptions is to leave the entire application as a blank white space.) So the first thing we should do is check for errors.
  3. In each of the lambdas inside OnGet, we should exit if something went wrong, so add this code as the first line of the first lambda:
     1. C#
     2. if (loadUnackOp.HasError) { loadUnackOp.MarkErrorAsHandled(); return; }
  4. Add this code as the first line of the second lambda:
     1. C#
     2. if (loadDetailsOp.HasError)
     3. { loadDetailsOp.MarkErrorAsHandled(); return; }
     4. While this may not be world class error reporting, we are actually expecting errors here as a matter of course until we log in, so it’s not totally unreasonable to swallow them. (A more sophisticated approach would be to check to see if we’re logged in before even attempting to poll. However, it’s conceivable that our login might have expired, so we need to be prepared for failure anyway.)
  5. Next we need to check to see if anything changed when we get the data back. Just before the line of code that sets UnacknowledgedRegistrations to the results, add this code:
     1. C#
     2. if (UnacknowledgedRegistrations != null)
     3. {
     4. int[] oldAttendeeEventIds = UnacknowledgedRegistrations.Select(
     5. e => e.AttendeeEventID).Distinct().ToArray();
     6. int[] newAttendeeEventIds = loadUnackOp.Entities.Select(
     7. e => e.AttendeeEventID).Distinct().ToArray();
     8. if (newAttendeeEventIds.Any(id => !oldAttendeeEventIds.Contains(id))
     9. && showNotification != null)
     10. {
     11. showNotification(newAttendeeEventIds.Length);
     12. }
     13. }
     14. The first time we fetch results, this won’t do anything, because UnacknowledgedRegistrations will be null. But each time after that, it looks as the set of AttendeeEvent row IDs. (That table contains one row for each registration.) It doesn’t do an exact comparison, because we expect rows to go away as registrations are acknowledged. So it just looks to see if the recently fetched list contains any IDs that weren’t there the last time we fetched the list. If so, that means new registrations are present.
  6. There’s just one problem: this won’t compile, because the UnacknowledgedRegistrationViewModel class doesn’t contain an AttendeeEventID. But that’s easily fixed. Add a property of that name with type int.
  7. Then in the LINQ query that initializes the UnacknowledgedRegistrationViewModel objects (which is immediately above the code you just added in OnGet) add an extra line to the object initializer in the select clause:
     1. C#
     2. AttendeeEventID = atev.AttendeeEventID,
  8. Run the application again. Log in as the administrator user (P@ssw0rd).
     1. To test this, you’ll need to add an extra event registration. You can use the web browser that opens when you debug the application. Right now it will be on EventAdministrationTestPage.html, showing the message indicating that the event admin dashboard needs to run out of browser. But if you modify the URL in the address bar, putting SlEventManagerTestPage.html on the end, you’ll be able to run the normal end user UI in the browser at the same time as you’re debugging the administration UI out-of-browser.
  9. In the web brower, log in as the “ian” user (“P@ssw0rd”), and you should see three listed events, the third of which the user is not yet registered for.
  10. Register for this event. After a few seconds, you should see a notification popup showing that there are now 3 unacknowledged event registrations.
      1. if you set breakpoints to follow the progress of the code, the focus changes that occur between Visual Studio and your application can end up dismissing the popup. It might vanish a fraction of a second after it appears, rather than staying up for the full 5 seconds. If you run without breaking in with the debugger you should see the full behavior.

Exercise 3: Window Handling

* 1. In the previous section, our notification popup told the user that there are attendees needing attention, but didn’t offer an easy way to get back to the application. Users could reasonably expect that clicking on the window would take them to the main window. So we’ll make that happen using Silverlight 4’s new window management features.

Managing the Windows

* 1. In the NotificationContent.xaml.cs code behind’s constructor, add a handler for the MouseLeftButtonDown event. In this handler, add the following code:
     1. C#
     2. Window w = Application.Current.MainWindow;
     3. w.Activate();
  2. Run the application.
  3. Click the button that forces the notification window to pop up. (It’s easier than adding a new user or event, in order to add another registration.) While the notification popup is visible, bring some other window (e.g. Visual Studio) to the front.
  4. Click on the notification popup. You should see your Silverlight application come back to the foreground.
  5. Show the popup again, and this time, minimize the application window. If you click in the popup, it doesn’t reappear. The taskbar icon will flash to draw attention to the application, but it won’t become visible. To fix this, add the following code to the mouse down handler:
     1. C#
     2. if (w.WindowState == WindowState.Minimized)
     3. {
     4. w.WindowState = WindowState.Normal;
     5. }

Handling Focus

* 1. There’s one more case we’re not handling correctly. If the main window is already active when the notification window appears, the notification ends up stealing the focus.
  2. To fix this, back in the OobUi.xaml.cs code behind, add this line of code at the start of the ShowNotification method:
     1. C#
     2. bool mainWindowIsActive = App.Current.MainWindow.IsActive;
  3. Then at the end of the method add this:
     1. C#
     2. if (mainWindowIsActive) { App.Current.MainWindow.Activate(); }
  4. Run the application again. This time, when notification windows appear, they shouldn’t move the focus away if the main window already has the focus.