* 1. 

Hands-On Lab

Building Applications in Silverlight 4

Module 8: Advanced OOB and MEF

**Contents**

[Introduction 3](#_Toc256194266)

[Exercise 1: Sending Email 4](#_Toc256194267)

[Exercise 2: Custom Window Chrome 7](#_Toc256194268)

[Configuring the Application for Custom Chrome 7](#_Toc256194269)

[Handling Move, Resize and Close Manually 8](#_Toc256194270)

[Exercise 3: Window Closing Event 11](#_Toc256194271)

[Exercise 4: MEF 13](#_Toc256194272)

[Sending Email via MEF 13](#_Toc256194273)

[Multiple Email Providers 15](#_Toc256194274)

Introduction

* 1. In this lab, you will explore some more out-of-browser options. You will continue with the additional administrative capability we added to the event manager application in the previous lab by adding the ability to send emails. You will then look at additional out-of-browser features added in Silverlight 4. Finally you will use MEF, the Managed Execution Framework to support runtime pluggability.
  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.

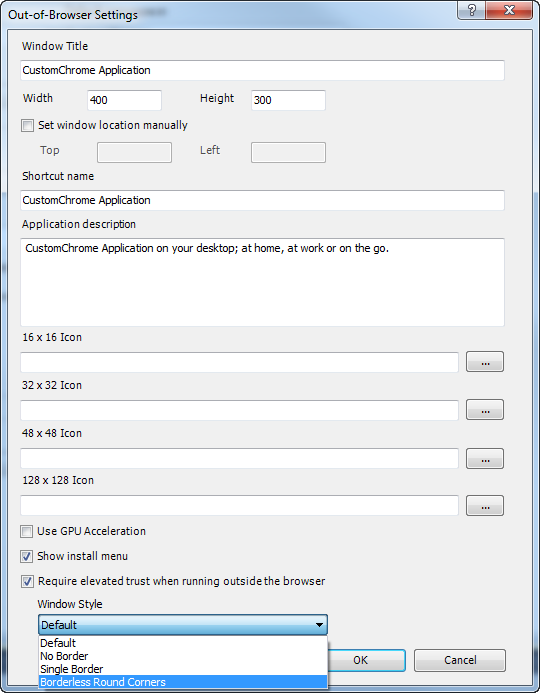
Exercise 1: Sending Email

* 1. The main goal of this administration UI is to provide a way to acknowledgement event registration requests. We need to send email to attendees confirming their registration. So now that we can display which users have outstanding registrations, we need to add the code to email them.
  2. We will be using Outlook to create email, controlling it with the COM automation interop feature added in Silverlight 4. Normal Silverlight applications cannot do this. They need to run with elevated privilege, so the first thing we need to do is modify the application to request elevation.
  3. To be clear, this does not mean administrative privileges. It simply means that Silverlight imposes fewer restrictions of its own; all the usual Windows security mechanisms will still apply.
  4. Elevation is a choice made at installation time, so first, we need to uninstall the application so that we can reinstall it with elevation.
  5. Run the application.
  6. Right-click on the UI, and choose the option to remove the application.
  7. Close the application and browser. The application is now uninstalled.
  8. Open the Properties page for the EventAdministration project go to the Silverlight tab.
  9. Click the Out-of-Browser Settings… button.
  10. At the bottom of the dialog that opens, check the Require elevated trust when running outside the browser checkbox:
      1. 
  11. Figure 1
  12. Requiring Elevated Trust
      1. If you try to run the application again, Visual Studio will get confused because we configured it for out-of-browser debugging, but the application is no longer installed for out-of-browser operation. So just once, we need to launch the web application on its own.
  13. In the Solution Explorer, right click on the SlEventManager.Web project and choose Debug → Start new instance. This debugs just the web application.
  14. Make sure the EventAdministrationTestPage.html page is showing, and install the application.
  15. This time you will get a different dialog, asking you if you are happy to let the application run elevated. Accept this.
  16. Close the web browser. The application is now installed for elevated out-of-browser operation, and we can go back to debugging it like before for the remainder of this lab.
  17. The view model will need to know which item is selected in order to send an email. So add a suitable property to OobUiViewModel:
      1. C#
      2. public UnacknowledgedRegistrationViewModel SelectedItem { get; set; }
      3. We can use a simple auto property here because the view model will never change the selection of its own accord, and so we don’t need to raise change notifications. Nor do we need code in the set accessor. We don’t need to take action when an event is selected. We just need to know which item is selected when the administrator decides to send an email.
  18. Add a using declaration for the System.Runtime.InteropServices.Automation namespace to the view models’ file.
  19. Add the following code:
      1. C#
      2. public void OnAcknowledge()
      3. {
      4. if (SelectedItem != null)
      5. {
      6. dynamic outlook =
      7. AutomationFactory.CreateObject("Outlook.Application");
      8. dynamic mailItem = outlook.CreateItem(0);
      9. mailItem.To = SelectedItem.UserEmail;
      10. mailItem.Subject = "Registration Confirmation";
      11. mailItem.HTMLBody = "Thank you for registering for " +
      12. SelectedItem.EventTitle;
      13. mailItem.Display();
      14. }
      15. }
      16. This uses Outlook’s automation API to create and display an email with the subject, to, and body prepopulated. The administrator can review the email before sending it.
  20. The code won’t compile yet. Support for the new dynamic keyword depends on a library that is not referenced by default. Add a reference to Microsoft.CSharp, which is found in the Silverlight SDK, at either
      1. 32 bit
         1. C:\Program Files\Microsoft SDKs\Silverlight\v4.0\Libraries\Client\
      2. 64-bit
         1. C:\Program Files (x86)\Microsoft SDKs\Silverlight\v4.0\Libraries\Client\
  21. In OobUi.xaml, add a button with a caption of “Acknowledge Registration…”.
  22. Add a Click handler for this button, and in it, call the view model’s OnAcknowledge method.
  23. In OobUi.xaml, add this attribute to the data grid:
      1. C#
      2. SelectedItem="{Binding Path=SelectedItem, Mode=TwoWay}"
  24. Run the application.
  25. Log in.
  26. Click Get to fetch the items.
  27. Select an item and click the button to acknowledge the registration.
  28. Outlook should start. (If you’re running on a machine for which you’ve not configured email, this might not get you very far. Outlook may start to show its installation user interface. But as long as it’s configured correctly, you should see a new email window.) The new email window’s To, Subject and body should be prepopulated.

Exercise 2: Custom Window Chrome

* 1. In this part of the lab, you will create a Silverlight application that disables the standard window borders when running out of browser, enabling you to provide a more distinctive-looking window.

Configuring the Application for Custom Chrome

* 1. Open Visual Studio 2010.
  2. In Visual Studio 2010, press Ctrl+Shift+N, or use the File→New→Project... menu item.
  3. Select Visual C#→Silverlight in the treeview on the left.
  4. In the center, select the Silverlight Application template.
  5. Name the project CustomChrome.
  6. Ensure the “Create directory for solution” checkbox is checked.
  7. In the New Silverlight Application dialog, let Visual Studio create a new web project to host the application.
  8. Double click on the CustomChrome project’s Properties node in the Solution Explorer.
  9. Select the Silverlight tab.
  10. Check the “Enable running application out of the browser” checkbox.
  11. Click the Out-of-Browser Settings… button.
  12. Check the “Require elevated trust when running outside the browser” checkbox.
      1. You can only disable the standard window borders in an elevated trust application.
  13. Set a fixed window Width and Height of 400 and 300 respectively.
      1. You might expect the settings for configuring the window border style to live in this dialog. However, the current preview version of the Silverlight 4 tools does not in fact offer any UI for controlling this.
  14. Click OK to close the Out-of Browser Settings dialog.
  15. Build the project.
  16. Right click the CustomChrome project and open its property window.
  17. Click the Out of Browser button.
  18. Select the Window Style to be **Borderless Rounded Corners**, as shown in the figure below.
      1. 
      2. Figure 2
      3. Customizing the Window Chrome
  19. Run the application. Install it for out-of-browser operation. When the OOB version appears, it will be a blank white rectangle with no content and no borders. (It might not even be that easy to see, depending on what else is on your screen.) You won’t be able to move, resize or close the window in the usual way yet. Get rid of it by right clicking, and choosing to remove the application.

Handling Move, Resize and Close Manually

* + 1. Since we’ll be turning off the normal window borders, we’ll need to provide our own UI for moving, resizing, and closing the window.
  1. In MainPage.xaml, add the following XAML inside the root grid:
     1. XAML
     2. <Grid.RowDefinitions>
     3. <RowDefinition Height="15" />
     4. <RowDefinition Height="\*" />
     5. <RowDefinition Height="15" />
     6. </Grid.RowDefinitions>
     7. <Grid.ColumnDefinitions>
     8. <ColumnDefinition Width="15" />
     9. <ColumnDefinition Width="\*" />
     10. <ColumnDefinition Width="15" />
     11. </Grid.ColumnDefinitions>
     12. <Rectangle x:Name="topLeft"
     13. Grid.Column="0" Grid.Row="0"
     14. Fill="#ff8080ff" />
     15. <Rectangle x:Name="top"
     16. Grid.Column="1" Grid.Row="0"
     17. Fill="#ff80ffff" />
     18. <Rectangle x:Name="topRight"
     19. Grid.Column="2" Grid.Row="0"
     20. Fill="#ff8080ff" />
     21. <Rectangle x:Name="left"
     22. Grid.Column="0" Grid.Row="1"
     23. Fill="#ff80ffff" />
     24. <Rectangle x:Name="right"
     25. Grid.Column="2" Grid.Row="1"
     26. Fill="#ff80ffff" />
     27. <Rectangle x:Name="bottomLeft"
     28. Grid.Column="0" Grid.Row="2"
     29. Fill="#ff8080ff" />
     30. <Rectangle x:Name="bottom"
     31. Grid.Column="1" Grid.Row="2"
     32. Fill="#ff80ffff" />
     33. <Rectangle x:Name="bottomRight"
     34. Grid.Column="2" Grid.Row="2"
     35. Fill="#ff8080ff" />
  2. Add a MouseLeftButtonDown event handler to the topLeft rectangle. Implement it with the following code:
     1. XAML
     2. private void topLeft\_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
     3. {
     4. App.Current.MainWindow.DragResize(WindowResizeEdge.TopLeft);
     5. }
  3. Add similar handlers for all the other rectangles you just added, picking the WindowResizeEdge enumeration that corresponds to the rectangle name.
  4. Drag a button onto the XAML design surface.
  5. Set the button content to Close. Call the button closeButton.
  6. Add a Click handler for the button. Inside the Click handler, add this code:
     1. C#
     2. App.Current.MainWindow.Close();
  7. Add a MouseLeftButtonDown event handler to the root Grid element. Add the following code to that handler:
     1. C#
     2. App.Current.MainWindow.DragMove();
  8. Run the application.
  9. Right-click on the application and install it to run out of browser.
  10. Once the application has been installed and is running, try dragging the bars at the edges and corners of the window. You should be able to resize the window with them. You should be able to move the window around by dragging on the main area. And you can close it by clicking the Close button.

Exercise 3: Window Closing Event

* 1. Silverlight 4 adds the ability to discover when an OOB application’s window is being closed (this works with Out of Browser applications running in the sandbox or as a Trusted Application). The Closing event offers the opportunity to cancel this operation (unless it is being closed due to the user shutting down or logging off from the system).
  2. Open Visual Studio 2010.
  3. In Visual Studio 2010, press Ctrl+Shift+N, or use the File→New→Project... menu item.
  4. Select Visual C#→Silverlight in the treeview on the left.
  5. In the center, select the Silverlight Application template.
  6. Name the project HandleClosing.
  7. Ensure the “Create directory for solution” checkbox is checked.
  8. In the New Silverlight Application dialog, let Visual Studio create a new web project to host the application.
  9. Double click on the HandleClosing project’s Properties node in the Solution Explorer.
  10. Select the Silverlight tab.
  11. Check the “Enable running application out of the browser” checkbox.
  12. Click the Out-of-Browser Settings… button.
  13. Check the “Require elevated trust when running outside the browser” checkbox, then click OK.
      1. Only trusted applications are allowed to block the Closing event.
  14. In the MainPage.xaml file, drag a CheckBox onto the design surface. Call it canCloseCheckBox, and set its Content to Can Close.
  15. Open the MainPage.xaml.cs code behind file.
  16. In the constructor, add the following code after the call to InitializeComponent:
      1. C#
      2. Application app = Application.Current;
      3. if (app.IsRunningOutOfBrowser)
      4. {
      5. app.MainWindow.Closing += MainWindow\_Closing;
      6. }
  17. Add the following method to the code behind class:
      1. C#
      2. void MainWindow\_Closing(object sender,
      3. System.ComponentModel.ClosingEventArgs e)
      4. {
      5. if (canCloseCheckBox.IsChecked == false &&
      6. e.IsCancelable)
      7. {
      8. e.Cancel = true;
      9. }
      10. }
  18. Run the application.
  19. In the web browser, right-click to install the application to run offline. Install the application
  20. When the out-of-browser version of the application appears, try closing its window. It will ignore the request.
  21. Check the checkbox.
  22. Try closing the application. This time it should close.

Exercise 4: MEF

* 1. The Managed Extensibility Framework (MEF) is a .NET Framework library feature designed to help build extensibility into applications. We’ll use this to enable multiple different mechanisms for sending email to be built into the application.

Sending Email via MEF

* 1. Open the SlEventManager solution. (If you haven’t completed the first part of this lab, copy the solution in the Completed\Exercise 1 folder and use that copy as your starting point.)
  2. Right click on the References for the EventAdministration project and choose Add Reference… Add references to the following two components:
     1. System.ComponentModel. Composition
     2. System.ComponentModel. Composition.Initialization
  3. The services that composable components—“parts”, as MEF calls them—provide are typically represented as interfaces. This is not strictly necessary, but if you want to have multiple implementations of a single service, defining an interface to represent the service makes sense. So add a new interface to the EventAdministration project called ISendEmail. (The current preview of the Silverlight 4 Tools for Visual Studio 2010 do not appear to include an Interface template, so you can use the Class template, and then change the class to an interface.) Define the interface as follows:
     1. C#
     2. public interface ISendEmail
     3. {
     4. void SendEmail(string recipientAddress, string eventTitle);
     5. }
  4. Add a new class to the EventAdministration project called SendEmailViaOutlook.
  5. Add using directives for the System.Runtime.InteropServices.Automation and System.ComponentModel.Composition namespaces.
  6. Implement the ISendEmail interface on the SendEmailViaOutlook class. Use the following implementation for SendEmail, which is a slightly modified version of the code from exercise 1 in this lab:
     1. C#
     2. public void SendEmail(string recipientAddress, string eventTitle)
     3. {
     4. dynamic outlook =
     5. AutomationFactory.CreateObject("Outlook.Application");
     6. dynamic mailItem = outlook.CreateItem(0);
     7. mailItem.To = recipientAddress;
     8. mailItem.Subject = "Registration Confirmation";
     9. mailItem.HTMLBody = "Thank you for registering for " + eventTitle;
     10. mailItem.Display();

1. }
   1. Add the following attribute to the class, to let MEF know that it is a provide for the ISendEmail service:
      1. C#
      2. [Export(typeof(ISendEmail))]
   2. Open the OobUiViewModel.cs file.
   3. Add a using directive for the System.ComponentModel.Composition namespace.
   4. Add the following property to the class:
      1. C#
      2. [Import]
      3. public ISendEmail SendEmail { get; set; }
   5. Find the OnAcknowledge method. This currently contains code that uses Outlook to send email. Replace this with the following code, which uses the email sender provided by MEF:
      1. C#
      2. public void OnAcknowledge()
      3. {
      4. if (SelectedItem != null && SendEmail != null)
      5. {
      6. SendEmail.SendEmail(SelectedItem.UserEmail, SelectedItem.EventTitle);
      7. }
      8. }
   6. Finally, we need to tell MEF to match the viewmodel’s imports to available exports. Do this by adding the following constructor to the OobUiViewModel class:
      1. C#
      2. public OobUiViewModel()
      3. {
      4. CompositionInitializer.SatisfyImports(this);
      5. }
   7. Ensure the application is deployed for offline use. (You can do this by running the web application first, going to the EventAdministrationTestPage.html page and installing the application if it hasn’t already been installed.) Run the application in the debugger.
   8. In the out-of-browser EventAdministration application, log in as administrator (P@ssw0rd), and click Get to get the list of unacknowledged registrations. In Visual Studio, put a breakpoint on the OnAcknowledge method, then select an item in the data grid and click Acknowledge…
   9. Step through the code, and you should be able to step into the call to SendEmail, and it will go into the SendEmailViaOutlook class you just wrote, even though the OobUiViewModel class has no direct knowledge of that class. It will attempt to create and display a new email in Outlook as it did in exercise 1 of this lab.

Multiple Email Providers

* 1. We’ve not gained much yet by using MEF. The only real benefit is that by abstracting the email sending code out, it would be slightly easier to write unit tests for the view model, because we could write a test that provided a fake IEmailSender implementation. But to demonstrate MEF’s ability to provide runtime flexibility, we’re going to extend the example to support multiple providers.
  2. We will present this in the user interface by offering multiple buttons, one for each provider. We’ll begin by defining a per-item view model for this list of buttons.
  3. In the OobUiViewModel class, Comment out the [Import] attribute on the SendEmail property. (The availability of multiple exports will cause MEF to throw an exception when it encounters a singular import.)
  4. Copy the RelayCommand.cs file from the SlEventManager project to the EventAdministration project.
  5. Add a new class called EmailSenderViewModel in the ViewModels folder, implemented as follows:
     1. C#
     2. public class EmailSenderViewModel
     3. {
     4. public string ButtonContent { get; set; }
     5. public RelayCommand SendCommand { get; set; }
     6. }
     7. The ButtonContent property will contain the text to be displayed on the button. This will need some content to distinguish the various email sender providers from one another. So our UI will need more than a list of available providers—it will need some information about each of the providers. We can use MEF’s metadata support—we can define an interface to represent the metadata we require.
  6. Add a new interface called IEmailSenderData, defined as follows:
     1. C#
     2. public interface ISendEmailData
     3. {
     4. string DisplayText { get; }
     5. }
     6. You do not normally need to implement this interface. Its purpose is merely to define the properties you need—MEF will implement it for you. We tell MEF what property values are required by annotating our exporter classes with a suitable attribute.
  7. Add the following attribute to the SendEmailViaOutlook class, to tell MEF what value we want it to provide in its implementation of ISendEmailData for this part:
     1. C#
     2. [ExportMetadata("DisplayText", "via Outlook")]
  8. In the OobUiViewModel class, add the following properties. The first uses MEF’s ImportMany attribute to be able to accept multiple providers. The second will act as a data source for an element in the UI.
     1. C#
     2. [ImportMany]
     3. public IEnumerable<Lazy<ISendEmail, ISendEmailData>> EmailSenders
     4. {
     5. set
     6. {
     7. var sendItemViewModels = from provider in value
     8. select new EmailSenderViewModel
     9. {
     10. ButtonContent = "Send email " +
     11. provider.Metadata.DisplayText,
     12. SendCommand = new RelayCommand(() =>
     13. {
     14. if (SelectedItem != null)
     15. {
     16. provider.Value.SendEmail(
     17. SelectedItem.UserEmail,
     18. SelectedItem.EventTitle);
     19. }
     20. }
     21. )
     22. {
     23. IsEnabled = true
     24. }
     25. };
     26. EmailSenderButtons = sendItemViewModels.ToList();
     27. }
     28. }
     29. public IList<EmailSenderViewModel> EmailSenderButtons { get; set; }
  9. Open OobUi.xaml.
  10. After the opening Grid tag, add the following:
      1. XAML
      2. <Grid.RowDefinitions>
      3. <RowDefinition Height="\*" />
      4. <RowDefinition Height="Auto" />
      5. </Grid.RowDefinitions>
  11. Just before the closing /Grid tag, add the following:
      1. XAML
      2. <ItemsControl
      3. Grid.Row="1"
      4. Margin="12,0,0,12"
      5. Height="23"
      6. ItemsSource="{Binding Path=EmailSenderButtons}">
      7. <ItemsControl.ItemsPanel>
      8. <ItemsPanelTemplate>
      9. <StackPanel Orientation="Horizontal" />
      10. </ItemsPanelTemplate>
      11. </ItemsControl.ItemsPanel>
      12. <ItemsControl.ItemTemplate>
      13. <DataTemplate>
      14. <Button
      15. Content="{Binding Path=ButtonContent}"
      16. Command="{Binding Path=SendCommand}"
      17. />
      18. </DataTemplate>
      19. </ItemsControl.ItemTemplate>
      20. </ItemsControl>
  12. Run the application again, and you should now see an extra button saying “Send email via Outlook”.
  13. We still only have one provider, but the UI is now ready to accept more. Add a new class to the project called SendEmailViaWebService.
  14. Add a namespace directive for the System.ComponentModel.Composition namespace.
  15. Implement the class as follows:
      1. C#
      2. [Export(typeof(ISendEmail))]
      3. [ExportMetadata("DisplayText", "via Web Service")]
      4. public class SendEmailViaWebService : ISendEmail
      5. {
      6. public void SendEmail(string recipientAddress, string eventTitle)
      7. {
      8. MessageBox.Show(string.Format("Email: {0}, {1}",
      9. recipientAddress, eventTitle));
      10. }
      11. }
      12. This doesn’t do anything of course. It’s just a stub to verify that the UI can use multiple different providers.
  16. Run the application again. This time you should see two buttons, one saying Send email via Outlook, and one saying Send email via Web Service.
  17. In the out-of-browser EventAdministration application, log in as administrator (P@ssw0rd), and click Get to get the list of unacknowledged registrations.
  18. Select an item in the data grid then click the Send email via Web Service button.
  19. A MessageBox should appear showing the email and event name, as a result of the SendEmailViaWebService handler being used.
  20. Dismiss the dialog, and verify that the Send email via Outlook button continues to use the Outlook provider.