* 1. 

Hands-On Lab

Building Applications in Silverlight 4

Module 3: User Registration with Authentication, Validation, MVVM, Implicit Styles and RichTextBox

**Contents**

[Introduction 3](#_Toc256193900)

[Exercise 1: Introducing MVVM and Customizing the UI by UI Role 4](#_Toc256193901)

[Setup Authentication 4](#_Toc256193902)

[Create Registration Buttons 7](#_Toc256193903)

[Create the ViewModel 8](#_Toc256193904)

[Exercise 2: Commands and the ViewModel 11](#_Toc256193905)

[Commanding 11](#_Toc256193906)

[Add Authentication Based Custom Domain Service Methods 13](#_Toc256193907)

[Implement the Bindings 15](#_Toc256193908)

[Exercise 3: Server-Side Access Control 18](#_Toc256193909)

[Exercise 4: Validation 19](#_Toc256193910)

[Exercise 5: RichTextBox 20](#_Toc256193911)

[Whitespace Handling 22](#_Toc256193912)

[Nested Formatting 22](#_Toc256193913)

[Read-only RichTextBox 23](#_Toc256193914)

[Programming the RichTextBox 23](#_Toc256193915)

[Exercise 6: Implicit Styling 26](#_Toc256193916)

Introduction

* 1. In this lab, you will modify the example application to customize data and behavior based on the logged in user, using the Model-View-ViewModel pattern. You will also implement server-side access control, and explore how WCF RIA Services data annotations work with data forms to provide end to end validation in Silverlight applications.
  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.
  3. In the lab for the previous module, your Silverlight application used a simple approach that works for very basic applications, but which can run into problems as your program grows. You were binding controls directly to data sources that came back from WCF RIA Services, and putting all the code into the code behind for your XAML pages. This is problematic for a number of reasons. Embedding application behavior into the code behind makes that behavior hard to test in isolation. You cannot write unit tests for that code without somehow loading the XAML. Binding directly to objects returned by a service raises problems when your application needs to show state that doesn’t correspond directly to properties those objects already have. You certainly wouldn’t want to go adding new items to the object model returned by a domain service simply to support user interface status such as knowing whether the user has dismissed a notification in the UI yet, but it’s entirely reasonable (a good idea in fact) to use data binding to work with that sort of state.
  4. Non-trivial Silverlight applications typically add an extra layer between the XAML and the underlying data model to provide flexibility and testability. You will use this approach in this part of the lab. You will customize the user interface based on whether the user is logged in, and if so, which role their account belongs to.
  5. Obviously, concepts such as account and role are as much server-side concerns as client-side ones, so these ideas belong in the world of the domain model. In fact the Silverlight Business Application template adds some services to support these features to any new application. The logic for deciding which specific UI elements should be available to which users is clearly specific to the client-side code, and that’s the logic that will go into the extra layer. We call this layer the ViewModel, because it sits between the View (the XAML with its codebehind) and the Model (the client-side wrappers generated by WCF RIA Services for our domain service, in this case).

Exercise 1: Introducing MVVM and Customizing the UI by UI Role

* 1. This exercise will add a ViewModel to the application’s home screen (view) while introducing MVVM practices. Authentication features will be modified to check against the sample database’s user and roles tables.

Setup Authentication

* 1. Open the SlEventManager solution in Visual Studio 2010 (copy the Starting Point solution).
     1. This lab uses the aspnet\_ tables and ASP.NET authentication. You can see the tables in the application’s database by following the steps below.

Expand the App\_Data folder and double click SlEventManager.mdf. This will show the Server Explorer panel, and will expand the Data Connections entry for your project’s database. Expand the Tables item for this database, and you’ll see various tables whose names begin with aspnet\_.

* + 1. Right click on aspnet\_Users and select Show Table Data, and it will show the contents of the table. You should see two rows, representing two user accounts (ian and administrator) that are present in the database provided for this example project.
    2. These aspnet\_ tables are standard ones recognized by various ASP.NET features for working with users, role-based security, and profiles. It’s common for these tables to end up in an auto-generated SQL Express database called ASPNETDB.MDF—Visual Studio can arrange for this database to be generated for you. (And in fact it will hide it by default, so it’s not always obvious that it’s done it.) But we’re not going to use a generated database. We’re going to use the tables already present in the SlEventManager.mdf database, for two reasons. First, the database is preloaded with some user accounts and roles that you’ll be using in this lab. Second, keeping everything in one database is simpler for deployment reasons.
    3. The one downside is that we need to configure the ASP.NET web application to use our database. By default it will go ahead and generate one for you. In fact, if you’ve already been playing with the login controls that the Silverlight Business Application template adds to new applications, it’s possible that this has already happened. Select the App\_Data folder and then click the Show All Files button at the top of Solution Explorer, and if you see an ASPNETDB.MDF, that means the automatic generation has already happened. Not that this is a problem. We’ll switch to the example database now.
  1. Open the SlEventManager.Web project’s web.config file.
  2. Find the <system.web> element.
  3. Delete the <roleManager enabled="true" /> element.
  4. Delete the **<profile>** element and all its children.
  5. Inside the <system.web> element, add the following:
     1. XML
     2. <membership defaultProvider="SlEventManagerMembershipProvider" userIsOnlineTimeWindow="15">
     3. <providers>
     4. <clear />
     5. <add name="SlEventManagerMembershipProvider"
     6. type="System.Web.Security.SqlMembershipProvider, System.Web, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
     7. connectionStringName="SlEventManagerDb" applicationName="/SlEventManager" enablePasswordRetrieval="false"
     8. enablePasswordReset="true" requiresQuestionAndAnswer="true" requiresUniqueEmail="true"
     9. passwordFormat="Hashed" />
     10. </providers>
     11. </membership>
     12. <roleManager enabled="true" defaultProvider="SlEventManagerRoleProvider">
     13. <providers>
     14. <clear />
     15. <add name="SlEventManagerRoleProvider"
     16. type="System.Web.Security.SqlRoleProvider, System.Web, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
     17. connectionStringName="SlEventManagerDb" applicationName="/SlEventManager" />
     18. </providers>
     19. </roleManager>
     20. <profile enabled="true" defaultProvider="SlEventManagerProfileProvider">
     21. <providers>
     22. <clear />
     23. <add name="SlEventManagerProfileProvider" connectionStringName="SlEventManagerDb"
     24. applicationName="/SlEventManager"
     25. type="System.Web.Profile.SqlProfileProvider, System.Web, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
     26. </providers>
     27. <properties>
     28. <add name="FriendlyName" />
     29. </properties>
     30. </profile>
     31. The editorial process has split some of the lines to fit them on the page. If you copy and paste this in you should be fine, but don’t try to reproduce the line breaks in the middle of the type attributes.
     32. This explicitly configures Membership, Role, and Profile providers. Silverlight Business Applications use all three of these. (The Membership feature is the one that deals with user logins, by the way.)
     33. This explicit configuration lets us do two things. First, it lets us specify an applicationName; this scopes the user accounts, roles, and profiles to our application, and it’s considered good practice always to set this. Second, it lets us specify the database connection string to be used for each of these features, which is how ASP.NET will know to use our database. All three are using the SlEventManagerDb connection string.
  6. Find the <connectionStrings> section of the web.config. Add another entry for the authentication database:
     1. XML
     2. <add name="SlEventManagerDb"
     3. connectionString="Data Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\SlEventManager.mdf;Integrated Security=True;User Instance=True;MultipleActiveResultSets=True"
     4. providerName="System.Data.SqlClient" />
     5. The connectionString should not have any line breaks in your web.config.
  7. Run the application.
  8. Click on the login link at the top right of the page. A login UI will appear.
  9. Log in with a username of administrator, and a password of P@ssw0rd (that’s a number zero after the w).
     1. The login should succeed, verifying that your application is successfully using the aspnet\_Users table (and associated tables such as aspnet\_Membership) to for managing logins. (Feel free to verify that it’s not simply allowing *any* credentials, by trying to log in with a non-existent account, or the wrong password. The login should fail when you try that.)
  10. Close the application.
  11. Back in Visual Studio, inspect the data in the aspnet\_Roles table.
      1. This contains two rows, defining two roles: Event Administrators and Registered Users. (In the database supplied, the administrator account belongs to both roles, while the ian account belongs only to the second. Role membership is defined in the aspnet\_UsersInRoles table.) The Registered Users role is one you’ll typically get by default in this sort of application, because the Silverlight Business Application template includes code to generate it.
      2. The AddUser method in the SlEventManager.Web project’s UserRegistrationService.cs file uses the ASP.NET Roles class to create the Registered Users role if it doesn’t already exist, and automatically adds newly-created users to that role.
  12. Run the application
  13. If the application still shows you as being logged in, log out.
  14. Use the login UI to create a new user account (all the code for registering as a new user is built into the template).
      1. Once you’ve done this you should see new entries in the aspnet\_Users, aspnet\_Membership, and aspnet\_UsersInRoles tables, representing your new account, and its membership of the Registered Users role.

Create Registration Buttons

* 1. Go to Home.xaml in the SlEventManager project’s Views folder.
  2. Add the following StackPanel after the existing one:
     1. XAML
     2. <StackPanel Orientation="Horizontal" HorizontalAlignment="Right">
     3. <Button x:Name="registerForEventButton" Content="Register" />
     4. <Button x:Name="unregisterForEventButton" Content="Unregister" />
     5. </StackPanel>
     6. Right now both panels will be visible at once, but we’ll fix that with a ViewModel that will provide bindable properties to control the visibility of these two panels.
  3. Add a new class to your SlEventManager project (with Shift+Alt+C or the Add→Class… context menu in the Solution Explorer) called ViewModelBase.
     1. A ViewModel needs to notify Silverlight whenever its properties change so that and data binding expressions that connect to the ViewModel know to update the target UI element. The usual way to do this is to implement INotifyPropertyChanged. So it’s common to define a base class to provide a common implementation of this interface for all ViewModels.
  4. Add this using declaration:
     1. C#
     2. using System.ComponentModel;
  5. Implement the class as shown here:
     1. C#
     2. public class ViewModelBase : INotifyPropertyChanged
     3. {
     4. public event PropertyChangedEventHandler PropertyChanged;
     5. protected void OnPropertyChanged(string propertyName)
     6. {
     7. if (PropertyChanged != null)
     8. {
     9. PropertyChanged(this,
     10. new PropertyChangedEventArgs(propertyName));
     11. }
     12. }
     13. }

Create the ViewModel

* 1. Add a new folder to the SlEventManager project called ViewModels.
     1. ViewModels are typically designed either for a particular view (i.e. a particular XAML file and its code behind) or some part of the view (e.g., an individual item in a list inside a view). You will be defining a ViewModel for the Home.xaml view.
  2. Add a class called HomeViewModel in the ViewModels folder.
  3. Make this new class derive from the ViewModelBase class you created.
  4. The ViewModel needs to determine the visibility of the administrator buttons, so add the following property definition:
     1. C#
     2. private Visibility \_adminButtonsVisibility;
     3. public Visibility AdminButtonsVisibility
     4. {
     5. get { return \_adminButtonsVisibility; }
     6. set
     7. {
     8. if (\_adminButtonsVisibility != value)
     9. {
     10. \_adminButtonsVisibility = value;
     11. OnPropertyChanged("AdminButtonsVisibility");
     12. }
     13. }
     14. }
     15. This is the typical boilerplate implementation for a property that raises change notifications. This particular property’s type is Visibility—that’s because we’re going bind this to one of the StackPanels’ Visibility properties.
  5. Add another similar property called AttendeeButtonsVisibility.
     1. C#
     2. private Visibility \_attendeeButtonsVisibility;
     3. public Visibility AttendeeButtonsVisibility
     4. {
     5. get { return \_attendeeButtonsVisibility; }
     6. set
     7. {
     8. if (\_attendeeButtonsVisibility != value)
     9. {
     10. \_attendeeButtonsVisibility = value;
     11. OnPropertyChanged("AttendeeButtonsVisibility");
     12. }
     13. }
     14. }
  6. Add the following code to have the ViewModel set these properties :
     1. C#
     2. private void UpdateForUserRole()
     3. {
     4. bool isLoggedIn = WebContext.Current.User.IsAuthenticated;
     5. bool isAdmin = isLoggedIn &&
     6. WebContext.Current.User.IsInRole("Event Administrators");
     7. AdminButtonsVisibility = isAdmin ?
     8. Visibility.Visible : Visibility.Collapsed;
     9. AttendeeButtonsVisibility = (isLoggedIn && !isAdmin) ?
     10. Visibility.Visible : Visibility.Collapsed;
     11. }
     12. This uses the WebContext class, which is provided by WCF RIA Services. This has built-in support for the concept of user authentication and roles. This relies on the AuthenticationService in the SlWebManager.Web project’s Services folder.
  7. We need to call the UpdateForUserRole method when the UI first loads, but also whenever the user logs in or out. So add the following constructor to your ViewModel:
     1. C#
     2. public HomeViewModel()
     3. {
     4. WebContext.Current.Authentication.LoggedIn += (s, e) => UpdateForUserRole();
     5. WebContext.Current.Authentication.LoggedOut += (s, e) => UpdateForUserRole();
     6. UpdateForUserRole();
     7. }
     8. The WebContext.Current.Authentication object offers these events to notify you when the user logs in or out, enabling us to know when we should update the ViewModel’s state.
  8. Create an instance of this ViewModel and bind the view to it by adding the field declaration and initializer in the Home.xaml.cs codebehind:
     1. C#
     2. HomeViewModel \_viewModel = new HomeViewModel();
  9. Add this using statement:
     1. C#
     2. using SlEventManager.ViewModels;
  10. In the view’s constructor, assign this into the DataContext for the view:
      1. C#
      2. this.DataContext = \_viewModel;
  11. In **Home.xaml**, go to the original **StackPanel**, the one containing the buttons for editing and creating events.
  12. Add the following attribute to the 2 StackPanels to cause the panel’s visibility to be controlled by the ViewModel.
      1. C#
      2. Visibility="{Binding Path=AdminButtonsVisibility}"
  13. Add a similar attribute to the other StackPanel, binding its Visibility to the AttendeeButtonsVisibility property.
  14. Run the application.
      1. Initially you will most likely not be logged in (unless you checked the checkbox to remain logged in earlier.) When not logged on, you will see no buttons at all. But if you log in as a normal user (e.g. ian, password P@ssw0rd), you should see the Register and Unregister buttons appear. If you log back out these will disappear again. If you log in as administrator (P@ssw0rd) you will see the buttons for editing and creating events.

Exercise 2: Commands and the ViewModel

* 1. One of the goals of a ViewModel is to minimize the amount of code in the view, in order to facilitate unit testing. The first part of this lab illustrated how data binding makes it possible for a ViewModel to update properties of elements in the view. But what about when we need things to flow in the other direction? How does the ViewModel discover user activity such as button clicks?
  2. One solution is to have ordinary event handlers in the view’s code behind, and have those call into methods on the ViewModel. For example, the ViewModel could define an OnRegisterButtonClicked method to be called by the view. While this doesn’t quite meet the ideal of avoiding all code in the view, that’s not a huge problem, because the only code required in the view is a single method call through to the ViewModel. However, Silverlight 4 introduces a feature that was previously only available on the desktop with WPF, which can help us do better: commands.
  3. Commands are an abstraction representing an operation that can be invoked through some user action such as a button click. You can data bind a Button’s Command property to a command object exposed by a ViewModel, which avoids the need for a Click handler in the code behind. The button is able to invoke the command directly. But it adds an additional benefit: the command abstraction (represented by the ICommand interface) also supports the idea that a command may be enabled or disabled from time to time. This means the ViewModel also has a way of controlling whether the button associated with a command is enabled or disabled. (Of course, we could have done that by offering a bool property on the ViewModel and binding a button’s IsEnabled property to that. But commands let us kill two birds with one stone: by binding just the one property: Button.Command, we can have the ViewModel both handle button clicks and set the enabled state of the button.
  4. In this exercise, we’ll use this technique for the Register and Unregister buttons you added in the previous part.

Commanding

* 1. Add a new class to the SlEventManager project called RelayCommand, with the following implementation:
     1. While Silverlight defines the ICommand interface, it doesn’t provide any implementations. So the first thing we need to do is provide our own. The most flexible approach is to write one class that simply relays the command invocation onto a delegate. That way we can use one command implementation to implement any number of commands.
     2. C#
     3. public class RelayCommand : ICommand
     4. {
     5. private Action \_handler;
     6. public RelayCommand(Action handler)
     7. {
     8. \_handler = handler;
     9. }
     10. private bool \_isEnabled;
     11. public bool IsEnabled
     12. {
     13. get { return \_isEnabled; }
     14. set
     15. {
     16. if (value != \_isEnabled)
     17. {
     18. \_isEnabled = value;
     19. if (CanExecuteChanged != null)
     20. {
     21. CanExecuteChanged(this, EventArgs.Empty);
     22. }
     23. }
     24. }
     25. }
     26. public bool CanExecute(object parameter)
     27. {
     28. return IsEnabled;
     29. }
     30. public event EventHandler CanExecuteChanged;
     31. public void Execute(object parameter)
     32. {
     33. \_handler();
     34. }
     35. }
     36. There are other common implementations of this general idea you can find on the Internet that are also called RelayCommand. The variations are in details such as whether they use the command parameter. This example doesn’t need it a command parameter, so we just ignore it. Another variation is whether the CanExecute status is handled with a property or a callback. But the basic principle is the same.
  2. Add two methods OnRegister and OnUnregister to the HomeViewModel.cs ViewModel class. These will be used as handlers for the commands:
     1. C#
     2. private void OnRegister()
     3. {
     4. }
     5. private void OnUnregister()
     6. {
     7. }
  3. Add the following property and field definitions to the HomeViewModel.cs ViewModel class:
     1. C#
     2. private readonly RelayCommand \_registerCommand;
     3. public ICommand RegisterCommand { get { return \_registerCommand; } }
     4. private readonly RelayCommand \_unregisterCommand;
     5. public ICommand UnregisterCommand { get { return \_unregisterCommand; } }
     6. While these properties will be used for data binding, they never change, so there’s no need to for set accessors nor any need to raise change notifications.
  4. In the constructor, add the following code to initialize these command fields with RelayCommand objects referring to the two handlers. Put these at the *start* of the constructor before the call to UpdateUserForRole.
     1. C#
     2. \_registerCommand = new RelayCommand(OnRegister);
     3. \_unregisterCommand = new RelayCommand(OnUnregister);
  5. Add a property to track the current selection.
     1. **W**e need to write the logic that will determine whether these commands should be enabled. This requires two things: we need to know events for which the user is already registered, and we need to know which event has currently been selected in the grid.
     2. C#
     3. private Event \_selectedEvent;
     4. public Event SelectedEvent
     5. {
     6. get { return \_selectedEvent; }
     7. set
     8. {
     9. \_selectedEvent = value;
     10. UpdateRegistrationButtons();
     11. }
     12. }
     13. private void UpdateRegistrationButtons()
     14. {
     15. }
  6. Add the following using directive:
     1. C#
     2. using SlEventManager.Web;

Add Authentication Based Custom Domain Service Methods

* 1. Add the following using directive in the EventManagerDomainService class in the SlEventManager.Web project:
     1. C#
     2. using System.Web.Security;
  2. Add this method, which will retrieve the IDs of the events for which the current user is registered.
     1. This uses the ASP.NET Membership class to discover the current user, so there’s no need for any parameters.
     2. C#
     3. [Invoke]
     4. public IEnumerable<int> FetchEventsForWhichCurrentUserIsRegistered()
     5. {
     6. MembershipUser mu = Membership.GetUser();
     7. if (mu == null)
     8. {
     9. return new int[0];
     10. }
     11. var q = from attendeeEvent in this.ObjectContext.AttendeeEvents
     12. where attendeeEvent.Attendee.AspNetUserId == (Guid) mu.ProviderUserKey
     13. select attendeeEvent.EventID;
     14. return q;
     15. }
     16. The *[Invoke]* attribute tells WCF RIA Services that this method does not attempt to return any entities. By default, when a domain service method returns an *IEnumerable<T>*, RIA Services presumes that the method intends to act as a query over some domain entities. That would cause it to report an error here, because this method enumerates ints, which are not valid entities. Declaring that this is simply an invocation-style operation avoids the error.
  3. A the following methods to handle registering and unregistering the current user for events:
     1. C#
     2. [Invoke]
     3. public void RegisterCurrentUserForEvent(int eventId)
     4. {
     5. Attendee attendee = GetOrCreateAttendeeForCurrentUser();
     6. if (!attendee.AttendeeEvents.Any(ev => ev.EventID == eventId))
     7. {
     8. attendee.AttendeeEvents.Add(new AttendeeEvent { EventID = eventId });
     9. }
     10. this.ObjectContext.SaveChanges();
     11. }
     12. [Invoke]
     13. public void UnregisterCurrentUserForEvent(int eventId)
     14. {
     15. Attendee attendee = GetOrCreateAttendeeForCurrentUser();
     16. AttendeeEvent av = attendee.AttendeeEvents.SingleOrDefault(
     17. x => x.EventID == eventId);
     18. if (av != null)
     19. {
     20. attendee.AttendeeEvents.Remove(av);
     21. }
     22. this.ObjectContext.SaveChanges();
     23. }
     24. private Attendee GetOrCreateAttendeeForCurrentUser()
     25. {
     26. MembershipUser mu = Membership.GetUser();
     27. if (mu == null)
     28. {
     29. throw new InvalidOperationException("User not logged in");
     30. }
     31. Attendee at = this.ObjectContext.Attendees.FirstOrDefault(
     32. x => x.AspNetUserId == (Guid) mu.ProviderUserKey);
     33. if (at == null)
     34. {
     35. at = new Attendee
     36. {
     37. AspNetUserId = (Guid) mu.ProviderUserKey
     38. };
     39. this.ObjectContext.AddToAttendees(at);
     40. }
     41. return at;
     42. }

Implement the Bindings

* 1. Add the following using declarations to HomeViewModel.cs so the ViewModel can use these methods and remember which events the current user belongs to.
     1. C#
     2. using System.Collections.Generic;
     3. using SlEventManager.Web.Services;
  2. Add the following field to the ViewModel:
     1. C#
     2. private HashSet<int> \_currentUserRegisteredEventIds;
  3. Implement the UpdateRegistrationButton method that we added earlier.
     1. This updates the button command status based on the current set of event ids:
     2. C#
     3. private void UpdateRegistrationButtons()
     4. {
     5. \_registerCommand.IsEnabled = \_currentUserRegisteredEventIds != null &&
     6. SelectedEvent != null &&
     7. !\_currentUserRegisteredEventIds.Contains(SelectedEvent.EventID);
     8. \_unregisterCommand.IsEnabled = \_currentUserRegisteredEventIds != null &&
     9. SelectedEvent != null &&
     10. \_currentUserRegisteredEventIds.Contains(SelectedEvent.EventID);
     11. }
  4. Add the following code at the end of the **UpdateForUserRole** method to populate the set of registered events:
     1. C#
     2. if (isLoggedIn)
     3. {
     4. var ctx = new EventManagerDomainContext();
     5. ctx.FetchEventsForWhichCurrentUserIsRegistered((op) =>
     6. {
     7. if (!op.HasError)
     8. {
     9. var items = op.Value;
     10. \_currentUserRegisteredEventIds = new HashSet<int>(items);
     11. UpdateRegistrationButtons();
     12. }
     13. }, null);
     14. }
     15. else
     16. {
     17. \_currentUserRegisteredEventIds = null;
     18. UpdateRegistrationButtons();
     19. }
  5. Implement the button command handlers we previously added empty methods for:
     1. C#
     2. private void OnRegister()
     3. {
     4. if (SelectedEvent != null)
     5. {
     6. var ctx = new EventManagerDomainContext();
     7. ctx.RegisterCurrentUserForEvent(SelectedEvent.EventID, (op) =>
     8. {
     9. UpdateForUserRole();
     10. }, null);
     11. }
     12. }
     13. private void OnUnregister()
     14. {
     15. if (SelectedEvent != null)
     16. {
     17. var ctx = new EventManagerDomainContext();
     18. ctx.UnregisterCurrentUserForEvent(SelectedEvent.EventID, (op) =>
     19. {
     20. UpdateForUserRole();
     21. }, null);
     22. }
     23. }
  6. Wire the view up to the commands. By binding the Command properties of the two buttons to the command objects exposed by the ViewModel:
     1. XAML
     2. <Button x:Name="registerForEventButton" Content="Register"
     3. Command="{Binding Path=RegisterCommand}" />
     4. <Button x:Name="unregisterForEventButton" Content="Unregister"
     5. Command="{Binding Path=UnregisterCommand}" />
  7. Bind the SelectedItem property of the data grid by adding the following attribute to the data grid:
     1. XAML
     2. SelectedItem="{Binding Path=SelectedEvent, Mode=TwoWay}"
  8. Run the application.
  9. Log in as a non-admin user (e.g., ian, P@ssw0rd).
     1. In the example database, the ian user is registered for the third event, so when you select that event the Register button should become disabled and the Unregister button should be enabled. For all other events, it should be the other way around. You should be able to us the buttons to register and unregister the user for events in the list.

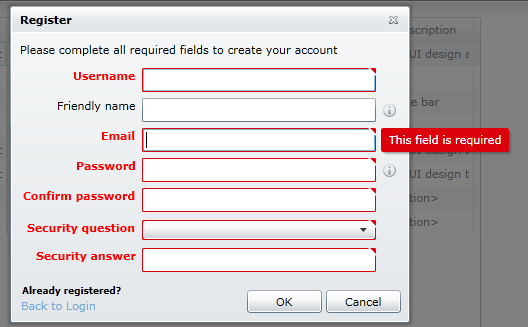
Exercise 3: Server-Side Access Control

* 1. Changing the available UI elements on the client side based on the user’s identity is a useful technique, but of course it’s not a secure one. We mustn’t rely on the client-side to enforce role-based secure access. So in this part, you’ll add security measures on the server side to restrict certain domain service facilities based on the role membership of the user. That way, even if a malicious user subverts the client-side code to gain access to features that they’re not authorized to use, their attempts will be foiled on the server side.
  2. Run the application and log in as administrator (P@ssw0rd).
  3. Select an event and click the button to edit it.
  4. Copy the URL to the clipboard.
  5. Edit the event, save the changes and return to the home page, verifying that the data changed.
  6. Log out of the application.

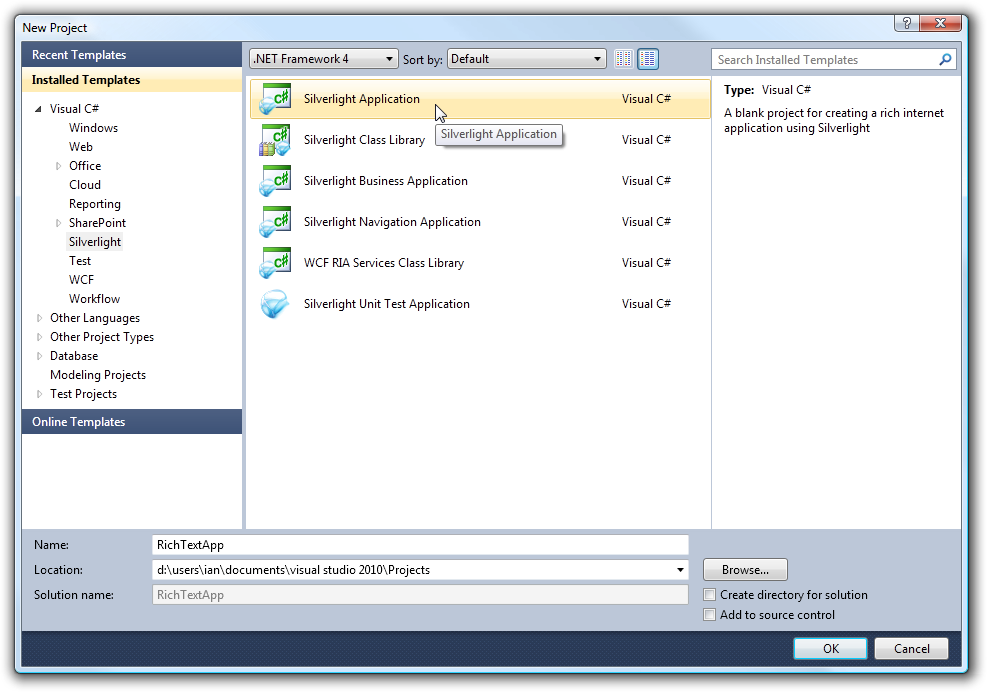
Paste the URL back in and this will take you to the editing page again.

* 1. Edit the event and save the changes again.
  2. Go back to the home page to verify that you were able to modify the event despite not being logged in.
  3. Close the application.
  4. In the SlEventManager.Web project, open the EventManagerDomainService.cs file in the Services folder.
  5. Find the UpdateEvent method.
  6. Add a [RequiresRole("Event Administrators")] attribute.
  7. Run the application again.
     1. As before, paste in the URL for the editing page while not logged in. Make a change to the event and click Save Changes. This time you should get an error. (Verify that you are still able to save changes when logged in as the administrator though.)

Exercise 4: Validation

* 1. The login UI allows new user accounts to be created. It performs some validation of data. In this final step you will add an additional validation requirement.
  2. Run the application.
  3. Ensure you’re not already logged in and then click the login button.
  4. Instead of logging in, click the Register Now link at the bottom of the dialog.
  5. Try clicking OK without filling in any fields.
     1. Most of the fields will light up red. If you click in them, you will see further details. This behavior is controlled in the RegistrationData.cs file in the Models folder of the SlEventManager.Web project.
     2. 
     3. Figure 1
     4. Registration Validation
  6. Close the browser.
  7. Find the FriendlyName property in the RegistrationData class. (Currently this field is not required.)
  8. Add a [Required] attribute to this property and run the application again.
  9. Try to register without filling any details in again. This time, the Friendly name field should turn red like the rest.

Exercise 5: RichTextBox

* 1. One way we could enhance user registration would be to enable users to supply some sort of text about themselves—if we wanted to add some sort of forum system to the site, the ability to provide a more detailed online identity would be important. We could add a simple text box to hold such information, but that’s a little limited. It might be better to use the RichTextBox added in Silverlight 4, which provides a way to display and edit formatted text—unlike the plain TextBox, the edited text can contain a mixture of plain, bold, or italic text, with different font sizes, colors, and typefaces if you choose. And the RichTextBox offers a read-only mode enabling such formatted text to be displayed easily. (It was possible to render formatted text in earlier versions of Silverlight, but the RichTextBox makes it much easier.)
  2. In fact we could use this feature in other places. For example, the event description itself is currently just plain text, but could be enhanced with the use of more flexibly styled text. Since the RichTextBox is very broadly applicable, this part of the lab will look at it in isolation, rather than wiring it into a specific part of the example application, so that we can focus on the details of using the RichTextBox itself.
  3. Open Visual Studio 2010.
  4. In Visual Studio 2010, press Ctrl+Shift+N, or use the File→New→Project... menu item.
  5. Select Visual C#→Silverlight in the treeview on the left.
  6. In the center, select the Silverlight Application template.
  7. Name the project RichTextApp.
     1. 
     2. Figure 2
     3. Add New Silverlight Application Project
  8. This project does not need a web site, so uncheck the “Create directory for solution” checkbox.
  9. Click OK.
  10. In the New Silverlight Application dialog that appears, uncheck the checkbox labeled Host the Silverlight application in a new Web site. Click OK.
  11. In the MainPage.xaml file that appears in Visual Studio, add the following inside the Grid:
      1. XAML
      2. <RichTextBox x:Name="rtb">
      3. </RichTextBox>
  12. Run the application. The empty rich text box will be filling the whole browser frame. Click in it and try typing something—at this stage it will behave in the same way as an ordinary TextBox. (It does not automatically handle common shortcuts such as Ctrl+B, so at this stage, you can’t exploit the ‘rich’ in rich text box.)
  13. A RichTextBox can be pre-loaded with formatted content. Inside the <RichTextBox> element, add the following XAML:
      1. XAML
      2. <Paragraph>
      3. This is some text.
      4. <Bold>This is some bold text.</Bold>
      5. </Paragraph>
  14. Run the application. You will see a mixture of normal and bold text. You can edit the text, and whatever text you type will be in the same style as the surrounding text.
      1. The design surface in the Release Candidate of Visual Studio 2010 is not completely consistent with what you’ll see at runtime. It handles whitespace differently. If you look closely, you’ll see that in Visual Studio, there is no space between the period at the end of the first sentence, and the start of the next sentence. But when you run the application, there is a space.
      2. The rules for whitespace handling in XAML are somewhat complex, but they are designed to provide the results you are most likely to want most of the time while letting you indent your XAML to reflect its structure. (They are similar in spirit to HTML’s whitespace handling.) For example, although both lines containing text in the XAML are indented, those indentation spaces do not appear at runtime. A run of whitespace in the middle of mixed content typically collapses down to a single space—so although the XAML has a new line, and space for indentation before the <Bold> tag, at runtime this is reduced to one space, which is what we want here. Whitespace at the beginning and end is handled differently—the whitespace between the opening <Paragraph> tag and the first bit of text is collapsed down to nothing, as is the whitespace between the closing </Bold> and the closing </Paragraph> tag.
      3. Although the whitespace handling usually does what you what, it’s occasionally necessary to take more control, so the next few steps will show the two ways to do this in XAML.

Whitespace Handling

* 1. In the XAML, add a couple of extra spaces between the words “is” and “some” on the first line of text. Notice that this has no effect, either in the designer or when you run the application.
  2. Add an xml:space="preserve" attribute to the opening <Paragraph> tag. The extra spaces you added will now be faithfully reproduced. So will any new lines—the two pieces of text new appear on two separate lines. More subtly, there’s a blank line at the start, because the opening <Paragraph> tag is followed immediately by a new line, and that has also been preserved. Moreover, the indentation has carefully been maintained too, which is not especially useful here.
     1. The xml:space attribute is a standard feature of XML, and you can add it to any XML document. The XML specification deliberately leaves some latitude for how applications use this attribute, and XAML’s whitespace rules explicitly take this attribute into account. So while this is a standard attribute, there is some XAML-specific behavior here.
  3. Remove the xml:space attribute you just added.
  4. Inside the <Paragraph> element, replace the first line of text with this:
     1. XAML
     2. <Run Text="This is some text." />
  5. Note that these extra spaces are honored for this particular run of text, but the other whitespace is collapsed as before. This is the usual technique for taking fine-grained control of whitespace when you only need precise control in a specific area.
     1. This works because the whitespace processing rules are applied to plain text when it appears as element content, but not to text inside attribute values.
     2. Be aware that if you don’t specify a Run explicitly as we just have, the XAML compiler converts plain textual content into Run elements for you—Run is the only type in the RichTextBox text object model that is capable of containing text. If you look at the properties offered by types such as Paragraph or Bold, you’ll see that that the only things these elements are capable of containing are other text elements. Only Run has a Text property. So you will end up with Run elements whether or not you put them explicitly in the XAML.

Nested Formatting

* 1. Formatting tags can be nested within one another, and their settings combine. And you can also apply text formatting attributes to the Paragraph container. Copy this XAML inside your RichTextBox to see this in action:
     1. XAML
     2. <Paragraph FontStyle="Italic">
     3. This is italic.
     4. <Bold>This is bold italic.</Bold>
     5. </Paragraph>
     6. <Paragraph>
     7. <Bold FontSize="20">
     8. Bold
     9. <Italic>Italic</Italic>.
     10. Bold,
     11. <Span FontFamily="Georgia">and in a different font.</Span>
     12. </Bold>
     13. </Paragraph>
     14. The <Span> element here is similar to <Bold> and <Italic> in that it can be used to apply formatting to a range of text. In fact the Bold and Italic classes both derive from Span. The difference is that Span has no intrinsic formatting of its own—it has no visible effect unless you set one or more font properties. The Bold element is essentially just a Span that sets the FontWeight to Bold for you.
     15. Because Span supports the FontWeight and FontStyle properties, you can use a Span to do anything you could do with Bold or Italic. Those elements are only provided for convenience.

Read-only RichTextBox

* 1. Add an IsReadOnly="True" attribute to the RichTextBox element. Run the application again. Observe that the text is displayed but can no longer be edited. Instead, it can be selected and copied.
     1. This mode is useful for displaying formatted text. If you are familiar with WPF, you would normally use one of the flow document reader controls for this purpose, but Silverlight combines editing and viewing functionality into the one control, RichTextBox.
  2. When using a RichTextBox in this way, it is common to want to support scrolling to show more text than will fit. This is built in. Add a VerticalScrollBarVisibility="Visible" attribute to the RichTextBox, and add some more paragraphs to the content so that you have enough text to make a scrollbar necessary. Run the application to verify that the scrollbar works as you’d expect.
  3. When displaying large volumes of text, you will typically want text wrapping enabled. This is disabled by default—add a long paragraph and you’ll see that it gets cropped rather than flowing across multiple lines. Add a TextWrapping="Wrap" property to the RichTextBox to fix this.

Programming the RichTextBox

* 1. To enable the end user to apply formatting, we need to provide some additional UI and code. Add the following XAML at the **top** of the Grid (before the RichTextBox).
     1. XAML
     2. <Grid.RowDefinitions>
     3. <RowDefinition Height="\*" />
     4. <RowDefinition Height="Auto" />
     5. </Grid.RowDefinitions>
     6. <StackPanel Orientation="Horizontal" Grid.Row="1">
     7. <Button x:Name="boldButton" Content="Bold" />
     8. <Button x:Name="italicButton" Content="Italic" />
     9. </StackPanel>
  2. Add Click event handlers to both buttons.
  3. In the code behind, add this helper:
     1. C#
     2. private void ToggleSetting(DependencyProperty property,
     3. object onValue, object offValue)
     4. {
     5. object currentValue = rtb.Selection.GetPropertyValue(property);
     6. bool alreadyApplied = currentValue.ToString() == onValue.ToString();
     7. rtb.Selection.ApplyPropertyValue(property,
     8. alreadyApplied ? offValue : onValue);
     9. }
     10. This detects whether a particular property is already applied across the current selection in the RichTextBox. If it is not already present, then that property is applied. If it is already present, then the ‘off’ value is applied instead. (We cannot just remove the property. For one thing, the selection object does not offer a method for doing that, but in any case, the property may be inherited from a containing element. For example, the entire Paragraph might be bold, but we might want to unbold some selected text. So the absence of a property on the selection is not enough—we need to explicitly apply a property forcing the property back to normal to be able to override formatting applied at a containing scope.
  4. Implement the bold button’s click handler like this:
     1. C#
     2. private void boldButton\_Click(object sender, RoutedEventArgs e)
     3. {
     4. ToggleSetting(TextElement.FontWeightProperty,
     5. FontWeights.Bold, FontWeights.Normal);
     6. }
  5. Similarly, implement the italic button like this:
     1. C#
     2. private void italicButton\_Click(object sender, RoutedEventArgs e)
     3. {
     4. ToggleSetting(TextElement.FontStyleProperty,
     5. FontStyles.Italic, FontStyles.Normal);
     6. }
  6. In the XAML, remove the IsReadOnly attribute from the RichTextBox.
  7. Run the application, select some text in the RichTextBox, and verify that the Bold and Italic buttons toggle the bold and italic appearance of the text.
  8. Finally, the ability to edit formatted text wouldn’t be much use if we couldn’t get hold of the current contents, so we’ll see how to retrieve the formatted text. The RichTextBox makes this available as XAML. Add the following to the XAML, after the Grid.RowDefinitions element, but before the StackPanel:
     1. XAML
     2. <Grid.ColumnDefinitions>
     3. <ColumnDefinition Width="\*" />
     4. <ColumnDefinition Width="\*" />
     5. </Grid.ColumnDefinitions>
     6. <ScrollViewer Grid.Column="1" Grid.RowSpan="2">
     7. <TextBlock x:Name="textAsXaml" TextWrapping="Wrap" />
     8. </ScrollViewer>
  9. Add a handler for the RichTextBox element’s ContentChanged event. Implement it as follows:
     1. C#
     2. private void rtb\_ContentChanged(object sender, ContentChangedEventArgs e)
     3. {
     4. textAsXaml.Text = rtb.Xaml;
     5. }
  10. Run the application, and make a modification to the contents of the RichTextBox. The right-hand side of the UI should show the Xaml for the RichTextBox’s current contents.

Exercise 6: Implicit Styling

* 1. Since the user registration features are, by definition, visible to the end user, visual design starts to become more of a concern than it might be for a purely internal application. (Not that there’s anything wrong with wanting internal LOB applications to look good. But appearance tends to become a higher priority with UI that customers can see.) The buttons we added earlier in this lab aren’t very consistent with the look of the rest of the application. The main navigation chrome’s link buttons look like neither normal hyperlinks nor normal buttons. And even where moderately normal-looking buttons appear (e.g. in the login UI) they don’t use the standard Silverlight appearance for a button.
  2. To make our buttons look more harmonious with the rest of the application, we’ll define a style for them. We’ll use the implicit styling feature added in Silverlight 4 to apply this style automatically to all buttons in the application that don’t specify their own custom style. That way, all the unstyled buttons we have added already will pick up the new look, as will any more we add in the future.
  3. Open the SlEventManager project in Visual Studio 2010.
  4. Open the App.xaml file.
  5. You’ll find the file already contains an <Application.Resources> element, with a <ResourceDictionary> inside it. The ResourceDictionary contains a couple of merged dictionaries. You’re going to add the style *after* the closing </ResourceDictionary.MergedResources> tag and *before* the closing </ResourceDictionary> tag. Paste in the following Xaml between those two closing tags:
     1. XAML
     2. <Style TargetType="Button">
     3. <Setter Property="Background" Value="#FF484848"/>
     4. <Setter Property="Foreground" Value="White"/>
     5. <Setter Property="Padding" Value="6"/>
     6. <Setter Property="Margin" Value="3"/>
     7. <Setter Property="BorderThickness" Value="1"/>
     8. <Setter Property="BorderBrush" Value="#FF1C1D33"/>
     9. <Setter Property="Template">
     10. <Setter.Value>
     11. <ControlTemplate TargetType="Button">
     12. <Grid>
     13. <VisualStateManager.VisualStateGroups>
     14. <VisualStateGroup x:Name="CommonStates">
     15. <VisualState x:Name="Normal"/>
     16. <VisualState x:Name="MouseOver">
     17. <Storyboard>
     18. <DoubleAnimation Duration="0" To="1"
     19. Storyboard.TargetProperty="Opacity"
     20. Storyboard.TargetName="BackgroundAnimation"/>
     21. </Storyboard>
     22. </VisualState>
     23. <VisualState x:Name="Pressed">
     24. <Storyboard>
     25. <DoubleAnimation Duration="0" To="1"
     26. Storyboard.TargetProperty="Opacity"
     27. Storyboard.TargetName="BackgroundAnimation"/>
     28. <ColorAnimation Duration="0" To="#FF563AFF"
     29. Storyboard.TargetProperty="(Border.Background).(SolidColorBrush.Color)"
     30. Storyboard.TargetName="BackgroundAnimation" />
     31. </Storyboard>
     32. </VisualState>
     33. <VisualState x:Name="Disabled">
     34. <Storyboard>
     35. <DoubleAnimation Duration="0" To=".55"
     36. Storyboard.TargetProperty="Opacity"
     37. Storyboard.TargetName="DisabledVisualElement"/>
     38. </Storyboard>
     39. </VisualState>
     40. </VisualStateGroup>
     41. <VisualStateGroup x:Name="FocusStates">
     42. <VisualState x:Name="Focused">
     43. <Storyboard>
     44. <DoubleAnimation Duration="0" To="1"
     45. Storyboard.TargetProperty="Opacity"
     46. Storyboard.TargetName="FocusVisualElement"/>
     47. </Storyboard>
     48. </VisualState>
     49. <VisualState x:Name="Unfocused"/>
     50. </VisualStateGroup>
     51. </VisualStateManager.VisualStateGroups>
     52. <Border x:Name="Background"
     53. BorderBrush="{TemplateBinding BorderBrush}"
     54. BorderThickness="{TemplateBinding BorderThickness}"
     55. Background="{TemplateBinding Background}"
     56. CornerRadius="1"
     57. Padding="1">
     58. <Border x:Name="BackgroundAnimation" Background="#FF448DCA"
     59. Opacity="0"/>
     60. </Border>
     61. <ContentPresenter
     62. x:Name="contentPresenter"
     63. ContentTemplate="{TemplateBinding ContentTemplate}"
     64. Content="{TemplateBinding Content}"
     65. HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
     66. Margin="{TemplateBinding Padding}"
     67. VerticalAlignment="{TemplateBinding VerticalContentAlignment}"/>
     68. <Rectangle x:Name="DisabledVisualElement" Fill="#FFFFFFFF"
     69. IsHitTestVisible="false" Opacity="0" RadiusY="1" RadiusX="1"/>
     70. <Rectangle x:Name="FocusVisualElement" IsHitTestVisible="false"
     71. Margin="1" Opacity="0" RadiusY="0.5" RadiusX="0.5"
     72. Stroke="#FF6DBDD1" StrokeThickness="1"/>
     73. </Grid>
     74. </ControlTemplate>
     75. </Setter.Value>
     76. </Setter>
     77. </Style>
  6. Run the application.
  7. Log in. (You can log in either as ian or administrator (P@ssw0rd, in either case), as either kind of user will cause some buttons to appear.)
  8. The buttons that appear when you log in should now look slightly more in keeping with the overall look provided by the Silverlight Business Application template. (Feel free to express your creativity by modifying the style to suit your own tastes.)