Microsoft® Silverlight™

Hands-On Lab

Building Applications in Silverlight 4

Module 2: Event Manager Application with WCF RIA Services, Creating and Editing Data, and Data Binding

Contents

Introduction 3
Exercise 1: Creating the Application 4
    Create the Solution 4
    Create the Entity Data Model 5
    Create the Domain Services 7
    Add Data Bindings and Domain Context 8
    Format the DataGrid and its Columns 10
Exercise 2: Editing Events 12
    Create the Events Details Page 12
    Setting up the Navigation Parameter 13
    Filtering the Domain Data Source 16
    Allow Saving of Events 17
    Conclusion 18
Exercise 3: Creating New Events 19
    Creating a New Event 19
Exercise 4: Track and Session Hierarchy 21
    Creating a Custom Domain Service Query 21
    Allow Related Objects to be Serialized 22
    Implement Adding Tracks and Talks 24

Introduction

In this lab, you will create the application that will be the basis for the other labs in this course. (Don't worry if you don't manage to complete a particular lab. These lab manual instructions are accompanied by completed solutions, so you can either build your own solution from start to finish, or dive straight in at any point using the solutions provided as a starting point.)

The example application is a web site for managing conferences and similar events. In this part of the lab, you will build a Silverlight user interface that will allow administrators to set up new events, configuring the talk tracks for those events, and the tracks that constitute the talks.

Estimated completion time for this lab is 45 minutes.

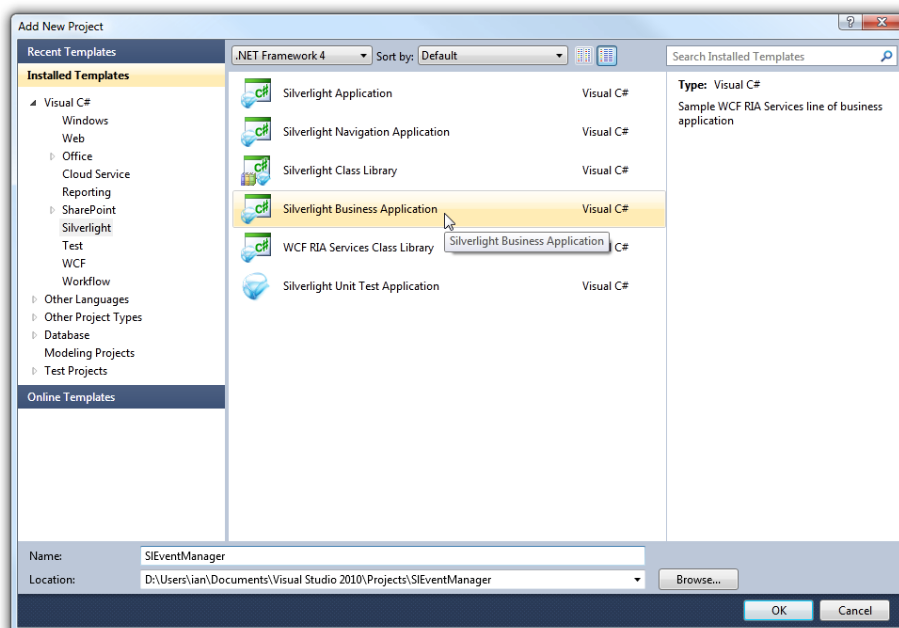
Exercise 1: Creating the Application

In this first step, you will create the Visual Studio solution to build the web site and Silverlight application. You will also add a prebuilt database, which you'll use via the Entity Framework, as the basis for a WCF RIA Services "Domain Service". This service will be used for the remaining parts of the lab, but you'll verify that the service is operating correctly by adding a simple DataGrid to display the events as the final step of this part.

Create the Solution

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#(C#) or Visual Basic(VB)→Silverlight** in the treeview on the left
4. In the center, select the Silverlight Business Application template.
5. Name the project **SlEventManager**.

C#:



Visual Basic:

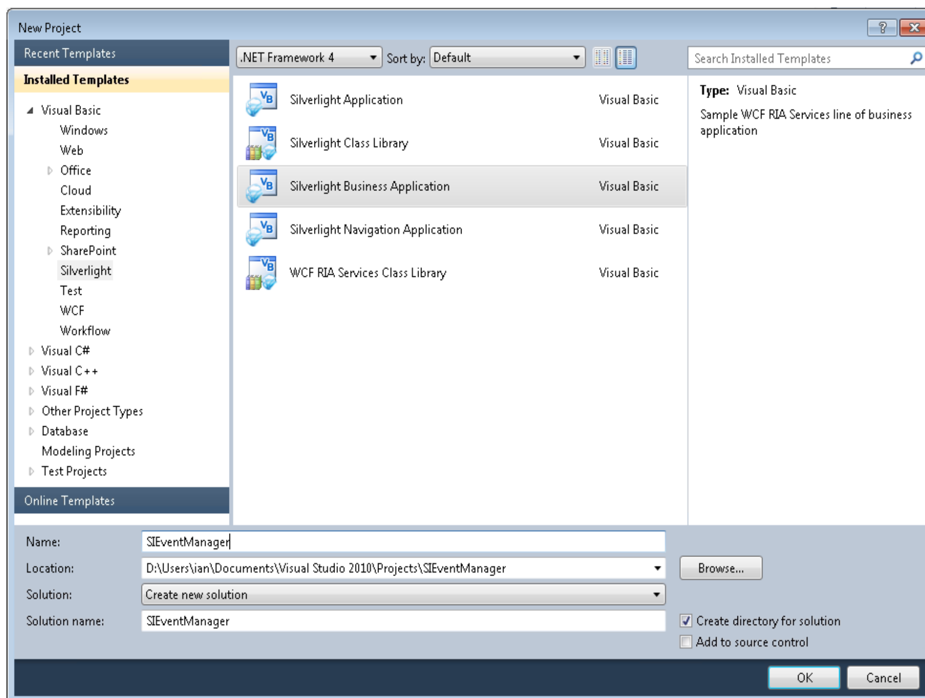


Figure 1

Add New Silverlight Business Application Project

6. Go to the **SlEventManager** project's Properties. (You can open this by double clicking on the Properties node under the project in the Solution Explorer.)
7. Click on the Silverlight tab. Notice that it shows that the project is linked to WCF RIA Services code in the Web project.

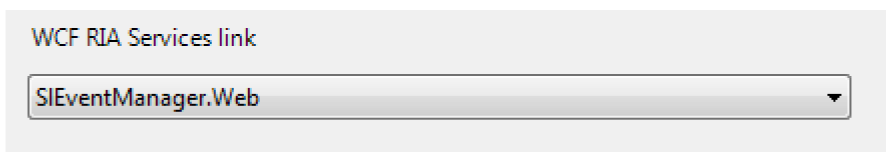


Figure 2

WCF RIA Services Link

This means that any domain services you define in the **SlEventManagement.Web** project will automatically be made available in the **SlEventManagerment** Silverlight project.

8. Close the properties page.

Create the Entity Data Model

1. This application will need a database. In the **SlEventManagement.Web** project in the **Solution Explorer**, right click on **App_Data** and choose **Add→Existing Item…**.

2. In the **Add Existing Item** dialog, find the folder for this lab and open the **DatabaseFiles** subdirectory.
3. Add the **SlEventManager.mdf** the project. Visual Studio will make a copy in the **App_Data** directory.
4. Double click on the newly-added file, **SlEventManager.mdf**, in the **Solution Explorer**.

> Visual Studio will add an entry for this database in the **Server Explorer** panel. You can look in its **Tables** node to see the structure of the database. It contains various tables whose names begin with **aspnet_**, which we'll be using in later labs when we add user management features to the application; these are standard tables used by ASP.NET. There are also various tables specific to this application, such as **Event**, which contains an entry for every event the application knows about. (The example database has been preloaded with some data so that there will be something to see as soon as you start wiring up the UI.)

5. In this application, we're going to use WCF RIA Services' ability to make server-side Entity Framework entities available to a Silverlight client. So we'll need to define an Entity Data Model.
6. Add a new **ADO.NET Entity Data Model** item to the **SlEventManager.Web** project. You can find this template by choosing **Visual C#(C#) or Visual Basic(VB)→Data** in the treeview on the left.
7. Set the name to **EventManagerDbModel** and click **Add**.
8. Select the **Generate from database** option in the **Entity Data Model Wizard** dialog and click **Next**.
9. On the next screen select the connection from the dropdown list for the **SlEventManager.mdf** database and then click **Next**.

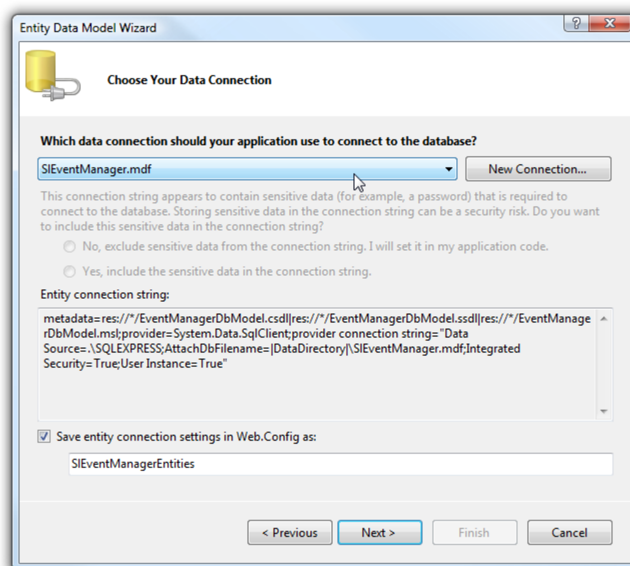


Figure 3

Choosing a Data Connection

10. The **Entity Data Model Wizard**'s next page lets you choose which database objects to represent in the Entity Data model. Expand the Tables section, and check the **Attendee**, **AttendeeEvent**, **AttendeeScheduleTalk**, **Event**, **EventTrack**, and **Talk** tables.
11. Click **Finish**. Visual Studio will show a diagram representing the Entity Data Model (EDM) you just created.
12. Rebuild the solution.

> Without this step, the current WCF RIA Services preview won't find the EDM you just added when you go onto the next steps. The web project will make this EDM available to Silverlight clients through WCF RIA Services.

Create the Domain Services

1. Expand the **Services** node in the web project.
2. Add a new service by right-clicking on the **Services** item and select **Add→New Item…**.
3. In the **Add New Item** dialog, choose the **Visual C#(C#) or Visual Basic(VB)→Web** item on the left.
4. Select the **Domain Service Class** template in the middle.

> There are a lot of web templates. Visual Studio 2010 provides a quick way to find the templates you require: there's a ***Search Installed Templates*** box at the top right of the form (and you can use the common ***Ctrl+E*** shortcut to get there). If you type ***Domain*** into that box, the dialog will just show those templates whose name contains the word Domain.

5. Name the new item **EventManagerDomainService**, and click **Add**. This will show a dialog that will let you pick the Entity Framework EDM you just created, and the entities from that EDM you'd like to use in your service.
6. Check all the checkboxes since we want all the entities to be available and editable to the Silverlight client. (We'll deal with access control in a later lab.)
7. Check the **Generate associated classes for metadata** checkbox:

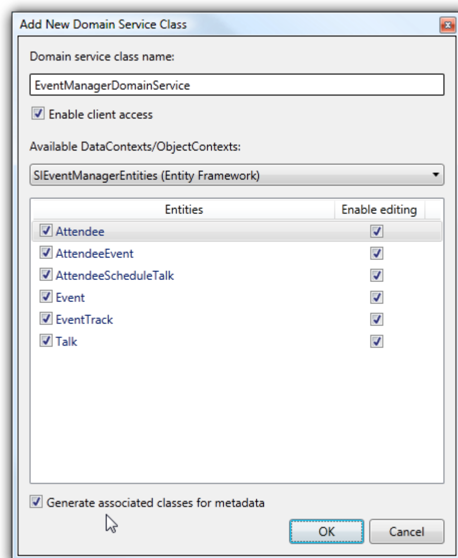


Figure 4
Add a New Domain Service Class

8. Click OK.

This will add a new class to your project that derives from **LinqToEntitiesDomainService<SlEventManagerEntities>**. This class is annotated with the **[EnableClientAccess()]** attribute, which tells WCF RIA Services to make this class and its operations available to clients. This also lets the **WCR RIA Services link** in Visual Studio know that it should provide code generation in the linked Silverlight project to make the domain services available. We'll verify that this is working by consuming some data from the service in the Silverlight UI.

Add Data Bindings and Domain Context

1. In the **SlEventManager** project, expand the **Views** folder and double click **Home.xaml.**
2. Delete the contents of the Grid. You should just have the root navigation: Page element containing a Grid, which should be empty.

3. Open Visual Studio's **Data Sources** panel, which can be opened with the **Data→Show Data Sources** menu item.

> Unless you've rebuilt the project since you added the service, you may see just one entry in here, **UserRegistrationContext**, which corresponds to the domain service that was put in the project by the **Silverlight Business Application** template.

4. Build the project again. You should see **EventManagerDomainContext**, which will contain child items for each of the entities in your EDM:

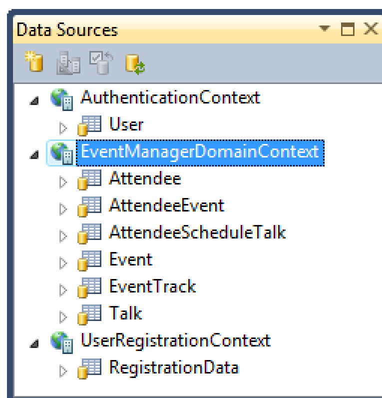


Figure 5
Locating the Domain Context in the Data Sources Window

5. Drag the **Event** item from the Data Sources window to the XAML design surface to add a DataGrid to the page.
6. Arrange the DataGrid so that it fills most of the space, with a bit of space around the edges.
7. Anchor the DataGrid's bottom and right hand edges to the containing space.

> By default, you'll see little circles against the right and bottom edges. Click these, and they will be replaced with arrows and lines, indicating that the edges are now anchored. (This works by removing the **VerticalAlignment** and **HorizontalAlignment** in the XAML. By default, the Visual Studio designer sets them to **Top** and **Left** respectively, but by removing them the DataGrid will default to resizing with its container.)

8. Run the application.

> You should see a web browser showing a grid that is briefly empty. WCF RIA Services fetches data asynchronously, so the UI appears before the data is ready. After a few seconds the data will become visible and you should see a list of events in the grid:

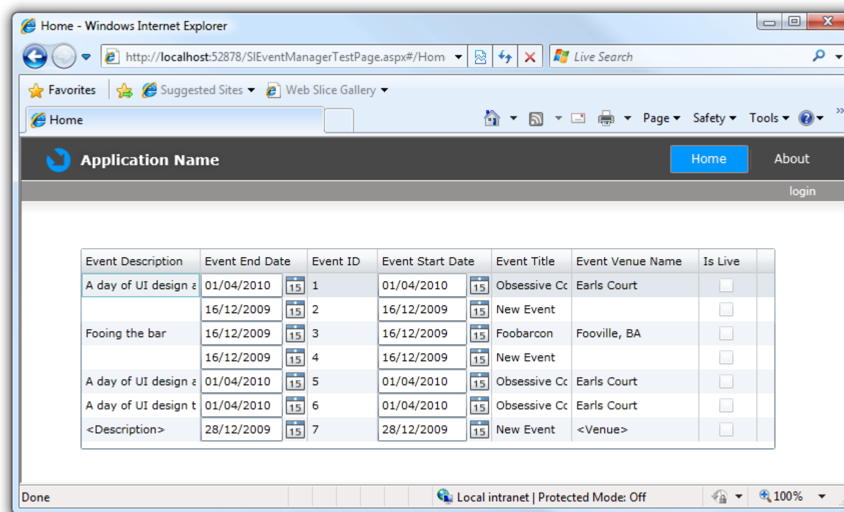


Figure 6

Running the Application for the First Time

9. Close the web browser.

Inspect the XAML that Visual Studio generated when you dragged the **Event** entity collection onto the UI. Notice that it added a **DataGrid**, which was visible at runtime, and a **<riaControls:DomainDataSource>** element. This is a non-visual control that can automatically fetch data (notice that its **AutoLoad** property is set to **True**) from a WCF RIA Services domain service. Its **DomainContext** property determines which service it will use. Visual Studio has put an **EventManagerDomainContext** in there. That's a type generated by the RIA services link that acts as the client-side representation of the **EventManagerDomainService** you added to the web project.

Format the DataGrid and its Columns

1. Select the DataGrid.
2. Go to the **Properties** panel to see the properties for the **DataGrid**.
3. Select the **Columns** property and click the "**…**" button to edit it.
4. Remove the **Event ID** and **Is Live** columns.
5. Reorder the columns so that the title appears first, followed by the start then end dates, then the venue, and finally the description.

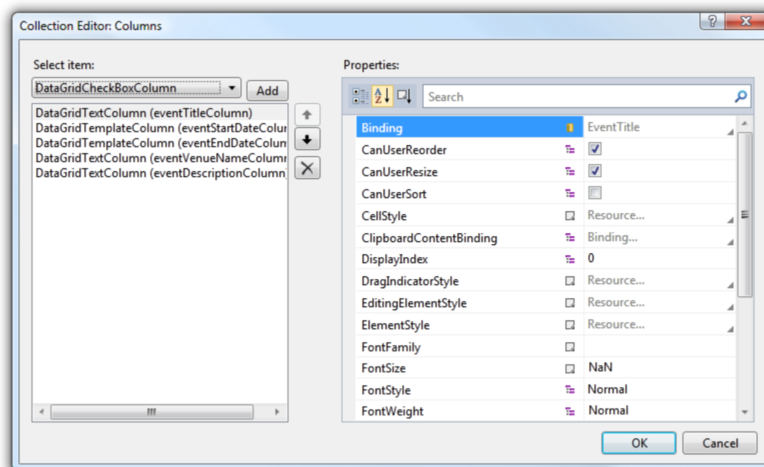


Figure 7

Editing the Columns

6. Set the **Description** column's **Width** to **Star**, so that it fills all the space not used by the other columns.
7. Click **OK**.
8. Run the application again. The grid should now be in a slightly more sensible order.

Exercise 2: Editing Events

In this part, you will add a new page to the Silverlight application for editing the details of an event. This will illustrate how the domain service we created in the previous part provides the ability to fetch data selectively and to push back changes to the server after the user has edited the data.

Create the Events Details Page

1. In the **SlEventManager** project, right-click on the **Views** folder and choose to **Add a New Item**.
2. Select the **Silverlight Page** template.
3. Name the new page **EditEvent.**
4. Click Add**.**
5. Open the **Data Sources** window.
6. Find the same **Event** entity item you added to the **Home** page in the previous part of this lab and select it.
7. Click on the dropdown button for the **Event** entity.

   > This lets you choose what sort of UI element or elements will appear when you drag the item onto the design surface. By default, a collection of entities such as this will produce a **DataGrid**, but the menu offers an alternative, **Details**.

8. Select Details.

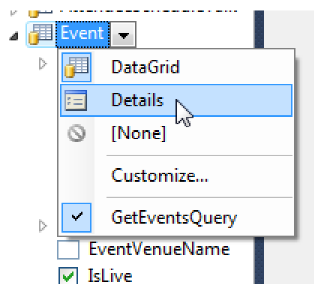


Figure 8
Changing the Layout to Details

9. Drag the **Event** entity onto the new XAML page.

   > This time, instead of a **DataGrid**, you should see a set of editable text fields, date pickers, and a checkbox to represent textual, date, and boolean values respectively:

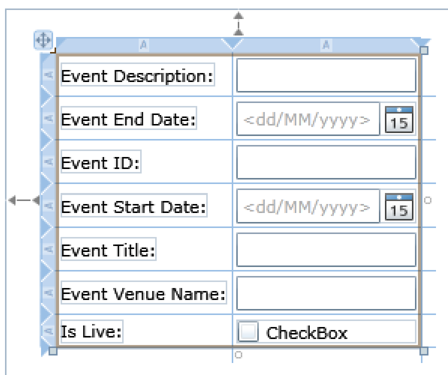


Figure 9

10. Laying out the Details

If you inspect the XAML, you'll see that once again, Visual Studio has added a **DomainDataSource** non-visual control to the page, which it has configured in exactly the same way as it did in the previous part. This would make sense if we were writing a master/details view, where the fields edited the currently-selected item in some list. But that's not how this UI will work: this page will show a single event. (It will eventually also use a master/details style, but that will be for managing the tracks and sessions. To do that for all the events as well would be overkill.) So we need to do some work to get the domain data source to pick just a single value. But how will it know which event it's supposed to edit? We'll put this into the URL so an event administrator can bookmark the page for editing a particular event. We'll have a URL of the form: **http://sitename/SlEventManagerTestPage.aspx#/EditEvent?EventID=3**

Right now, there's no way to get to this new page, so we'll start by adding a button to the **Home.xaml** page to edit the currently selected event. We'll make that navigate to this new page, putting the event ID into the URL query string.

Setting up the Navigation Parameter

1. Open the **Home.xaml.cs(C#) or Home.xaml.vb(VB)** codebehind file.
2. Add the following helper function, which will help us navigate to EditEvent:

C#

```
private void NavigateToEditEvent(int eventId)
{
    NavigationService.Navigate(new Uri("/EditEvent?EventID=" +
        eventId, UriKind.Relative));
}
```

Visual Basic

```
Private Sub NavigateToEditEvent(ByVal eventId As Integer)
    NavigationService.Navigate(New Uri("/EditEvent?EventID=" & eventId,
UriKind.Relative))
End Sub
```

3. Add the following using declaration for this to compile:

C#

```
using System;
```

Visual Basic

```
Imports System
```

This method builds a URL with the event ID in the query string, and navigates to the event editing page. (We will eventually need to perform this step in a couple of different places, which is why we're putting it in a helper function.)

4. Go back to the **Home.xaml** page in Visual Studio.

5. Make some extra space at the top of the page by dragging the top of the data grid down a bit.

6. Add a grid row by clicking in the blue bar to the left of the design surface:

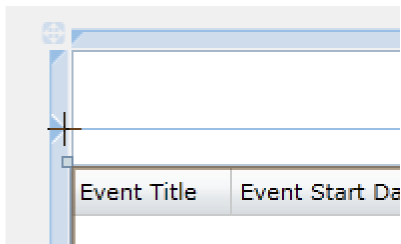


Figure 10

Adding a Grid Row

7. After the closing tag for the DataGrid, but before the closing </Grid> tag, add the following XAML:

XAML

```
<StackPanel Orientation="Horizontal" HorizontalAlignment="Right">
    <Button x:Name="editCurrentButton" Content="Edit current event" />
</StackPanel>
```

8. In the Grid.RowDefinitions section near the top of the file (which was added when you created the extra row), modify the Height of the first row to Auto.

9. Double click on the **Edit current event** button in the design view. This generates and displays the button's **Click** event handler.

10. Add the code shown below to the **Click** hander:

C#

```
private void editCurrentButton_Click(object sender,
     System.Windows.RoutedEventArgs e)
{
    Event currentEvent = eventDataGrid.SelectedItem as Event;
    if (currentEvent != null)
    {
        NavigateToEditEvent(currentEvent.EventID);
    }
```

```
}
```

Visual Basic

```
Private Sub editCurrentButton_Click(ByVal sender As System.Object, ByVal e
As System.Windows.RoutedEventArgs) Handles editCurrentButton.Click
	Dim currentEvent As [Event] = TryCast(eventDataGrid.SelectedItem,
[Event])
	If currentEvent IsNot Nothing Then
		NavigateToEditEvent(currentEvent.EventID)
	End If
End Sub
```

> The **Event** type that this refers to comes from the domain service you added earlier. The **WCF RIA Services link** feature of Visual Studio generates client-side classes that correspond to the entities exposed by the service. However, it generates these in a separate namespace, so the code above won't compile.

11. Add the following using declaration to the top of the codebehind file:

C#

```
using SlEventManager.Web;
```

Visual Basic

```
Imports SlEventManager.Web
```

12. We should only enable the button when the user has selected an item (an Event) in the DataGrid. Go to the XAML, find the button and set its IsEnabled property to false.
13. Double click on the **DataGrid**. This will add a **SelectionChanged** event handler.
14. Add the code shown in this **SelectionChanged** handler:

C#

```
private void eventDataGrid_SelectionChanged(object sender,
    SelectionChangedEventArgs e)
{
    editCurrentButton.IsEnabled =
        eventDataGrid.SelectedItem != null;
}
```

Visual Basic

```
Private Sub EventDataGrid_SelectionChanged(ByVal sender As System.Object,
ByVal e As System.Windows.Controls.SelectionChangedEventArgs) Handles
EventDataGrid.SelectionChanged
    editCurrentButton.IsEnabled = EventDataGrid.SelectedItem IsNot Nothing
End Sub
```

15. Run the application.
16. Select an event in the grid
17. Click the Edit current event button.

The application should navigate to the event editing page, and you'll see the ID of the item you selected in the URL. However, the editing page is currently ignoring this ID completely. It will just edit the first event that comes back from the service every time.

18. Close the web browser.

Filtering the Domain Data Source

1. Go to the **EditEvent.xaml.cs(C#) or EditEvent.xaml.vb(VB)** page. Visual Studio will have added an **OnNavigatedTo** method that gets called when the user navigates to this page.

2. Implement the **OnNavigatedTo** method as shown:

C#

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    string eventId = NavigationContext.QueryString["EventID"];
    eventDomainDataSource.FilterDescriptors.Add(
        new FilterDescriptor
        {
            PropertyPath = "EventID",
            Operator = FilterOperator.IsEqualTo,
            Value = eventId
          });
}
```

Visual Basic

```
Protected Overrides Sub OnNavigatedTo(ByVal e As NavigationEventArgs)
      Dim eventId As String = NavigationContext.QueryString("EventID")
      eventDomainDataSource.FilterDescriptors.Add(New FilterDescriptor
With {.PropertyPath = "EventID", .Operator = FilterOperator.IsEqualTo,
.Value = eventId})
End Sub
```

This retrieves the **EventID** from the URL. It then adds a filter descriptor collection to the **DomainDataSource** non-visual control telling it to filter events by an exact match on the **EventID** property.

3. Run the application.

4. Select an event in the **DataGrid**.

5. Click the **Edit current event** button. This time, the event editing page will show the details for the item you have selected, rather than the first item.

Although we specified the filter criteria on the client, the filtering is in fact taking place on the server, thanks to the "composability" feature of WCF RIA Services. When a domain service operation returns an IQueryable<T>, WCF RIA Services allows clients to pass in optional filter, sorting, and grouping criteria as part of the request. These criteria are then

applied to the IQueryable<T> that the domain service operation returns, using the standard LINQ query operators for filtering, sorting, and grouping.

Since our domain services returns IQueryable<T> directly from the Entity Framework, this means that filtering, sorting, and grouping are done by the Entity Framework. And the Entity Framework implements these standard LINQ operators in the database. So in this example, the client-side filter specification is ultimately implemented as a WHERE clause in the SQL query executed against the database.

6. Stop the application.

Allow Saving of Events

1. In **EditEvent.xaml**, drag a button on from the **Toolbox**. (You can open the **Toolbox** from the **View** menu.)

While the user can edit the event details, these edits remain on the client side. RIA Services will not push changes back to the database unless asked, because it has no way of knowing when it should do that. We need to add some code to make that happen.

2. Set the button's **Name** to **saveChangesButton**
3. Set the button's **Content** to **Save Changes**.
4. Double click the button to generate the **Click** handler.
5. Add just one line of code to the **Click** event handler:

C#

```
private void saveChangesButton_Click(object sender,
    RoutedEventArgs e)
{
    eventDomainDataSource.SubmitChanges();
}
```

Visual Basic

```
Private Sub saveChangesButton_Click(ByVal sender As System.Object, ByVal e
As System.Windows.RoutedEventArgs) Handles saveChangesButton.Click
    EventDomainDataSource.SubmitChanges()
End Sub
```

This **SubmitChanges** method builds a message describing all of the properties that have changed since the entities were fetched from the services. (The client-side entity objects that WCF RIA Services generates all perform change tracking to enable this.) It sends this message to the server where the server-side parts of RIA Services will start up a transaction, and then call all the methods on your domain service needed to perform all of the updates.

So as far as network communications are concerned, updates happen as a single operation, but this batching is done for you. Your domain service just provides the individual insert, update, and delete operations and RIA Services will call what it needs to call to process the

> batched update. (The relevant insert, update, and delete operations were generated when you added the domain service, because you checked the **Enable editing** checkboxes.)

6. Run the application again.
7. Pick an event to edit and click the **Edit current event** button.
8. Change a couple of the properties.
9. Click **Save Changes**.
10. Click the **Home** link at the top left to go back to the home page with the event list. This will reload the data, so you should now be able to see the changes you made in the list.

Conclusion

The functionality you just added wasn't technically necessary to add this new page. The DataGrid is editable by default. It does two-way binding to the source objects, and you can double click on grid cells to edit them. So we could have added a button to save changes on the original form, using the same SubmitChanges method. However, in general it's common to have more selective pages, such as the single event editing page added in this part. When we add features later in this lab to edit track and session information, there will be more data to show per event, and having all of the events be editable from a single page would involve fetching a rather large quantity of data up front. (We'd be not far away from having the client download a complete copy of the database on startup. That might work for the small volumes of data we're working with in this example, but it's clearly not a sustainable approach for a real application.)

Exercise 3: Creating New Events

Administrators will need the ability to create new events, rather than just editing existing ones, so we need to support this in the user interface. Just as we were able to modify existing entities on the client and submit changes to the server, we can also create new entities. There's just one extra step to add newly-created entities to the corresponding entity collection. All we need to do is provide an extra button to do this on the main page, and we can then reuse the existing event editor to edit the newly created event.

Creating a New Event

1. Open **Home.xaml**, and find the **StackPanel** containing the **Edit current event** button.
2. Add another button to the panel named **createNewEventButton**.
3. Set its **Content** to **Create New Event**.
4. Add a Click handler to the new button.
5. Add the following using declaration to your code behind. This is necessary to use the domain context class for your domain service.

C#

```
using SlEventManager.Web.Services;
```

Visual Basic

```
Imports SlEventManager.Web.Services
```

6. Implement the Click handler as follows:

C#

```
private void createNewEventButton_Click(object sender,
    System.Windows.RoutedEventArgs e)
{
    EventManagerDomainContext ctx = new EventManagerDomainContext();
    DateTime today = DateTime.Now.Date;
    Event newEvent = new Event
    {
        EventTitle = "<Title>",
        EventVenueName = "<Venue>",
        EventDescription = "<Description>",
        EventStartDate = today,
        EventEndDate = today
    };
    ctx.Events.Add(newEvent);
    ctx.SubmitChanges((op) =>
    {
        if (!op.HasError)
        {
```

```
            NavigateToEditEvent(newEvent.EventID);
        }
    }, null);
}
```

Visual Basic

```
Private Sub createNewEventButton_Click(ByVal sender As Object, ByVal e As
System.Windows.RoutedEventArgs)
    Dim ctx As New EventManagerDomainContext()
    Dim today As Date = Date.Now.Date
    Dim newEvent As [Event] = New [Event] With {.EventTitle = "<Title>",
.EventVenueName = "<Venue>", .EventDescription = "<Description>",
.EventStartDate = today, .EventEndDate = today}
    ctx.Events.Add(newEvent)
    ctx.SubmitChanges(Sub(op)
                          If Not op.HasError Then
                              NavigateToEditEvent(newEvent.EventID)
                          End If
                      End Sub, Nothing)
End Sub
```

This creates a new domain context that will communicate with the domain service. We could use the domain context already in the **DomainDataSource**, but that wants to be in charge of its own change submission so that it can update controls once changes are complete. In this case, we want to navigate to a different page as soon as the operation completes, so we want to take care of our own completion handling, so it's simpler just to create a new context.

Most of the code here just initializes the new event, populating all the properties to avoid validation failures. The new entity would be rejected if any of these were missing. Since we're calling **SubmitChanges** directly on the context (rather than the **DomainDataSource** non-visual control) we get to provide a callback function that will be invoked when the work is complete. If the operation succeeds, our newly created **Event** entity's **EventID** will have been updated with the ID generated by the database, so we can use that to navigate to the event editing page, with the same helper function we used before to edit existing events.

7. Run the application.
8. Click on the button to create a new event. You will be taken to the event editing page, which will show the newly-created event. If you navigate back to the home page, you'll see this new event in the list.

Exercise 4: Track and Session Hierarchy

Events won't be very interesting if they have no sessions for attendees to listen to. So our event editing page also needs to provide a list of the tracks in the event, along with the sessions for each track. So we'll need to add two data grids to our event editor, but we also need to modify the service to ensure that all the necessary entities are available.

The domain service operation our Silverlight application currently uses to get event information only returns **Event** entities. The **GetEvents** method in the **EventManagerDomainService** just returns the **ObjectContext.Events** query, and by default, the Entity Framework does not automatically fetch related entities. (It can perform automatic deferred fetching, but by the time the entities have been returned back to the Silverlight client it's too late for this to happen because the Entity Framework is no longer in the picture. Its work is already done by then.)

For the **Home.xaml**, this is what we want. Our page shows a list of all the events, so we really don't want to fetch all related data for all events as it would make the page too slow to load. However, for the individual event editing page, we do need to fetch the related track and session items.

Creating a Custom Domain Service Query

1. Find the **GetEvents** method in **EventManagerDomainService** (in the **SlEventManager.Web** project's **Services** folder).
2. Make a copy called **GetEventsWithTracksAndTalks**.
3. Modify its implementation as shown below:

C#

```
public IQueryable<Event> GetEventsWithTracksAndTalks()
{
    return this.ObjectContext.Events.Include("EventTracks.Talks");
}
```

Visual Basic

```
Public Function GetEventsWithTracksAndTalks() As IQueryable(Of [Event])
    Return Me.ObjectContext.Events.Include("EventTracks.Talks")
End Function
```

> The **Include** method tells the Entity Framework which navigation properties we are planning to use on the entities it returns. This causes it to fetch all of the EventTracks and Talks up front for each Event entity returned.

4. Open **EditEvent.xaml** in the Silverlight project.
5. Add split the main into **Grid** into two columns**.**

> You can do this by **c**licking in the blue bar along the top of the design surface to split the UI into two roughly equal-sized columns.

6. Open the **Data Sources** panel if it's not already open.
7. Expand the **Event** entity, and notice that it has an **EventTracks** item inside it.
8. Expand the **EventTracks** child item, and notice that it in turn contains a **Talks** item.

> These items behave differently than the **EventTrack** and **Talk** entities that are directly underneath the **EventManagedDomainController** element. Entity collections shown as children of another entity collection in the Data Sources window represent specifically those entities that are related to the parent (as opposed to all the entities of that type). If you drag the **EventTracks** item from inside the **Event** entity onto the UI, it will produce a **DataGrid** that shows only the tracks related to the current **Event**, rather than showing all the **EventTrack** entities.

9. Drag the EventTracks item (the one under the Event item) onto the UI.
10. Arrange the DataGrid so it is at the bottom left hand side.
11. Set the layout to resize as the containing window resizes.

> Inspect the XAML, and notice that Visual Studio has done something slightly different with this entity collection. It has not added an extra **DomainDataSource** but instead it has added a **CollectionViewSource** which refers to the existing **DomainDataSource**, so that it can show items related to the current **Event**. We're not yet using the new query operation so there will be no data in the EventTracks **DataGrid** yet.

12. In the XAML for EditEvents.xaml, find the **DomainDataSource**.
13. Change the QueryName property to use the **GetEventsWithTracksAndTalksQuery** to use the new operation you just added.
14. Run the application.
15. Select the first event and edit it. Be sure to select the first event in the list, as this one has some tracks configured.)

> This will *fail*. The **Tracks DataGrid** you just added will be empty. You just asked to edit an event that has tracks in, you bound a data grid to the tracks for the selected event, and you used a service operation that asked the Entity Framework to fetch related tracks and talks. Despite all that, nothing appeared.
>
> The reason this just failed is that it's not enough to tell the Entity Framework that we want it to fetch certain related entities. We also need to tell WCF RIA Services that we need those entities to be brought across to the client. By default, it will only bring across those entities that it can see need to be returned based on the signature of the domain service operation. Since our **GetEventsWithTracksAndTalks** method's signature only mentions **Event** entities, that is all we get back.

Allow Related Objects to be Serialized

1. Open the **EventManagerDomainService.metadata.cs(C#) or EventManagerDomainService.metadata.vb(VB)** file in the **Services** folder of the **SlEventManager.Web** project.
2. Find the **EventMetadata** class.
3. Add an **[Include](C#)<Include()>(VB)** attribute to the **EventTracks** field.
4. Run the application again.
5. Select the first event again and edit it. This time you should see some tracks show up in the track list.

> You might be wondering why we need to tell both the Entity Framework and WCF RIA Services to include the related entities. There is a good reason for it. WCF RIA Services needs to be conservative, and only expose the information we tell it too, as otherwise it would be hard to control what it made available—it might reveal information we don't necessarily want revealed.
>
> The **[Include]** attributes need to be there. But we can't depend entirely on those attributes, because we might want to include different sets of entities in different circumstances. For example, our application now has two domain operations for returning **Event** entities, one of which returns related tracks and talks, while the other does not. If inclusion of related entities was entirely down to the **[Include]** attribute we would not have been able to do this.

6. Go to the **Data Sources** window.
7. Drag the **Talks** from inside the **EventTracks** inside the **Event** onto the bottom right of the UI of EditEvents.xaml.
8. Add the **[Include]** attribute to the **EventTrackMetadata** class's **Talks** field.

> When you drag the new **Grid** on you'll find an extra **DomainDataSource** has been added called **eventDomainDataSource1**. What's happened here is that Visual Studio has been confused by us changing the **QueryName.** It makes it think that our XAML no longer contains a suitable **DomainDataSource** for the **Event** entities, so it has added another. (And it has also added two new CollectionViewSource items, when we only wanted it to add one.) But this will stop everything working. So you'll need to delete the new DomainDataSource, and point things (the CollectionViewSource, for example) back at the original DomainDataSource. You will also need to delete the duplicate CollectionViewSource that was created.

9. Delete the new DomainDataSource1 (see note above).
10. Delete the duplicate CollectionViewSource, called eventEventTracksViewSource1. You should now have 2 CollectionViewSource entries.

11. Point the CollectionViewSource called eventEventTracksTalksViewSource back at the original CollectionViewSource by modifying the Binding expression's Source property—make it refer to eventEventTracksViewSource instead of eventEventTracksViewSource1.

XAML

```
<sdk:Page.Resources>
    <CollectionViewSource x:Key="eventEventTracksViewSource"
      Source="{Binding Path=Data.EventTracks,
      ElementName=eventDomainDataSource}" />
    <CollectionViewSource x:Key="eventEventTracksTalksViewSource"
      Source="{Binding Path=Talks, Source={StaticResource
      eventEventTracksViewSource}}" />
</sdk:Page.Resources>
```

12. Run the application again.
13. Select the first event. There's only one talk configured right now, so you'll only see one listed, and only one the first track is selected.

Implement Adding Tracks and Talks

1. Add two more buttons to the UI for EditEvents.xaml labeled **New Track** and **New Talk.**
2. Name the new buttons **newTrackButton** and **newTalkButton** respectively.
3. Add **Click** handlers for each and implement them as follows:

C#

```
private void newTrackButton_Click(object sender, RoutedEventArgs e)
{
    Event currentEvent =
        eventDomainDataSource.Data.Cast<Event>().Single();
    currentEvent.EventTracks.Add(new EventTrack
        { EventTrackTitle = "New Track" });
}

private void newTalkButton_Click(object sender, RoutedEventArgs e)
{
    EventTrack track =
        eventTracksDataGrid.SelectedItem as EventTrack;
    if (track != null)
    {
        track.Talks.Add(new Talk { TalkTitle = "New Talk" });
    }
}
```

Visual Basic

```
Private Sub newTrackButton_Click(ByVal sender As Object, ByVal e As
RoutedEventArgs)
    Dim currentEvent As [Event] = EventDomainDataSource.Data.Cast(Of
[Event])().Single()
```

```
    currentEvent.EventTracks.Add(New EventTrack With {.EventTrackTitle =
"New Track"})
End Sub

Private Sub newTalkButton_Click(ByVal sender As Object, ByVal e As
RoutedEventArgs)
    Dim track As EventTrack = TryCast(EventTracksDataGrid.SelectedItem,
EventTrack)
    If track IsNot Nothing Then
        track.Talks.Add(New Talk With {.TalkTitle = "New Talk"})
    End If
End Sub
```

4. You will need to add the following using declarations:

C#

```
using SlEventManager.Web;
using SlEventManager.Web.Services;
```

Visual Basic

```
Imports SlEventManager.Web
Imports SlEventManager.Web.Services
```

This code works a little differently than the code to add a new **Event**. That's because this time we do want the **DomainDataSource** to be in on the act: we want the data grids to show the newly added tracks. We are going to let the **DomainDataSource** remain in charge of the updates. Newly added tracks and talks will get added to the database at the same time as any other changes when the user clicks the **Save Changes** button added earlier. But there are a couple of other complications caused by the fact that we are not simply adding new items out of the blue. We are adding new items that are related to existing items.

For example, the new **EventTrack** belongs to an **Event**. So instead of adding it to the domain context's **EventTrack** collection, we add it to the current **Event** object's **EventTracks**. (The first line of code in the first handler discovers the current **Event**. It's a little eccentric because the **DomainDataSource** is designed to hold collections of objects, but this particular page has just a single **Event**. So we need to unwrap it from its collection.)

The second handler does something similar, but just grabs the currently selected **Track** from the track data grid.

5. Run the application.
6. Select an event you created and edit it.
7. Click the button to add a new **Track**. It should appear immediately in the **DataGrid**. You can use grid to edit the properties of the track.
8. Ensure the track is selected.
9. Add some talks to the track, and again edit them in the grid.

10. Click the **Save Changes** button.

    This will write all the new items to the database. You should see all the ID columns change at this point, because the database-generated IDs get passed back once the data has been inserted. You should find that if you go back to the home page and then go back into your event, the newly added tracks and talks are now visible.