.NET RIA Services Demo – Library App

Note – the final project is in Library.App folder. All the intermediate steps are in the zip.

Step 1: New Business App (pre-done)

Run the app as-is, and **describe the template**, the navigation model, the individual pages. Point is to start with something, and not a blank screen, as that helps in overall productivity

Step 2: Add data model

We’re doing data-first.

Create a **DataModel folder** in Web app

Add **LibraryDB.dbml with Books and Categories** – describe simple schema we have

Step 3: Build and make sure we’re good

Step 4: Add a domain service

This is the first of .NET RIA Services… this class is central to both the framework, and to your app – it encapsulates your application logic that is specific to your domain

In this case, I am building a Library management app, so I’ll call it **LibraryManagementService**. Add it in the Services folder. Choose empty in the wizard, because I want to explain as I go along. Describe the class that comes out – and point the base class.

If I was not using LINQ to SQL I could have a different base class, or just derive from DomainService directly.

Point out [EnableClientAccess]

Step 5: Add first query method

Query is the most basic thing – you want to provide access to your data, but in a controlled way. Add rules for example, filter out books that have been marked as missing. Add **GetBooks**

public IQueryable<Book> GetBooks() {

return from b in Context.Books

where b.Missing == false

orderby b.PublishDate descending

select b;

}

Step 6: Go to the client project and build

We want to start using this query right away. Lets build to make sure its fine.

Step 7: Add a **DataGrid** to the HomePage.xaml

This lets us quickly see what data is coming down to the client, by using its auto-generate columns feature.

<data:DataGrid x:Name="booksDataGrid" Grid.Row="4" />

Step 8: Go to code-behind in HomePage.xaml.cs and start using our **LibraryManagementContext**.

This became available to the client because of enable client access. Show what is on the Context object.

private LibraryManagementContext \_context;

public HomePage() {

InitializeComponent();

\_context = new LibraryManagementContext();

booksDataGrid.ItemsSource = \_context.Books;

}

// Executes when the user navigates to this page.

protected override void OnNavigatedTo(NavigationEventArgs e) {

\_context.Load(\_context.GetBooksQuery());

}

Step 9: Back to the server – we can add more than one way to retrieve entities, by adding additional query methods

Add a **SearchBooks**

public IQueryable<Book> SearchBooks(string searchText) {

if (String.IsNullOrEmpty(searchText)) {

return GetBooks();

}

string searchPattern = "%" + searchText + "%";

return from b in Context.Books

where b.Missing == false &&

(SqlMethods.Like(b.Title, searchPattern) ||

SqlMethods.Like(b.Author, searchPattern) ||

SqlMethods.Like(b.Description, searchPattern))

orderby b.PublishDate descending

select b;

}

Notice the DAL specifics – The use of % is a SQL thing, and SqlMethods is a Linq to SQL API. Basically when you are implementing a DomainService, you’re working directly with the DAL APIs you chose, and encapsulating them. The client doesn’t know about it (this is why I am doing the % thing on the server, and not on the client).

The client also doesn’t care how the search was done. I could change the implementation to use full text search and the client would be just fine.

Step 10: On the client, **add some search UI** to HomePage.xaml

Make sure everything is good to go – compile/build

<StackPanel Grid.Row="2" Orientation="Horizontal">

<TextBox x:Name="searchTextBox" Width="200" VerticalAlignment="Center" />

<Button x:Name="searchButton" Content="Search" Width="75" VerticalAlignment="Center"

Click="searchButton\_Click" Margin="4,0,0,0" />

</StackPanel>

Step 11: Add **search code**

private void searchButton\_Click(object sender, RoutedEventArgs e) {

\_context.Books.Clear();

EntityQuery<Book> bookQuery = \_context.SearchBooksQuery(searchTextBox.Text);

\_context.Load(bookQuery);

}

Step 12: Using LINQ on client

Going back to initial load, I don’t want to bring all books down to the client… this is a toy db, but a real db might have several 1000s of books.

The standard query metaphor we have is LINQ, and I can use exactly that on the client, and RIA Services will execute the query on the server for me.

protected override void OnNavigatedTo(NavigationEventArgs e) {

EntityQuery<Book> bookQuery =

from b in \_context.GetBooksQuery()

where b.Author.StartsWith("A")

select b;

\_context.Load(bookQuery);

}

The framework saves the query, sends it to the server, executes your query method, adds the client query on top of your queryable, and then executes it. So in the case of LINQ to SQL, the query for authors starting with A goes all the way to the database.

Step 13: How about having the framework produce queries for me, for standard things like sorting, filtering, paging?

We have some convenient components – DataSource, DataPager etc. If your client-side code doesn’t really have much logic other than taking some inputs, creating queries, and loading data, because all the interesting logic is on the server, you might as well use these declarative components.

<ria:DomainDataSource x:Name="booksDataSource"

QueryName="SearchBooksQuery" AutoLoad="False">

<ria:DomainDataSource.DomainContext>

<app:LibraryManagementContext />

</ria:DomainDataSource.DomainContext>

<ria:DomainDataSource.QueryParameters>

<riadata:ControlParameter ControlName="searchTextBox" ParameterName="searchText" />

</ria:DomainDataSource.QueryParameters>

</ria:DomainDataSource>

<data:DataGrid x:Name="booksDataGrid" Grid.Row="4"

ItemsSource="{Binding ElementName=booksDataSource, Path=Data}"/>

<data:DataPager x:Name="booksDataPager" Grid.Row="5"

PageSize="10"

Source="{Binding ElementName=booksDataSource, Path=Data}"/>

Step 14: Remove the code from code-behind and instead call Load when the search button is clicked.

public partial class HomePage : Page {

public HomePage() {

InitializeComponent();

}

// Executes when the user navigates to this page.

protected override void OnNavigatedTo(NavigationEventArgs e) {

}

private void searchButton\_Click(object sender, RoutedEventArgs e) {

booksDataSource.Load();

}

}

Run, show paging, sorting, filtering

Step 15: Customize the DataGrid – its good for initial display of data, but I don’t want that many columns.

<data:DataGrid x:Name="booksDataGrid" Grid.Row="4"

ItemsSource="{Binding ElementName=booksDataSource, Path=Data}"

AutoGenerateColumns="False">

<data:DataGrid.Columns>

<data:DataGridTextColumn Header="Title" Binding="{Binding Title}" Width="200" />

<data:DataGridTextColumn Header="Author" Binding="{Binding Author}" Width="150" />

</data:DataGrid.Columns>

</data:DataGrid>

Step 16: Add a DataForm to show details

First make some room for DataForm by adding a column to layout

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*" />

<ColumnDefinition Width="10" />

<ColumnDefinition Width="400" />

</Grid.ColumnDefinitions>

Next add the DataForm

<dataControls:DataForm x:Name="booksForm" Grid.Row="4" Grid.Column="2"

VerticalAlignment="Top" AutoEdit="False" AutoCommit="False"

CommandButtonsVisibility="Edit,Cancel,Commit,Navigation"

ItemsSource="{Binding ElementName=booksDataSource, Path=Data}">

</dataControls:DataForm>

Notice they bind to the same thing, and the DataSource manages currency so both selection in DataGrid and DataForm are automatically synchronized.

Step 17: Customize the DataForm

<dataControls:DataForm x:Name="booksForm" Grid.Row="4" Grid.Column="2"

VerticalAlignment="Top" AutoEdit="False" AutoCommit="False"

CommandButtonsVisibility="Edit,Cancel,Commit,Navigation"

ItemsSource="{Binding ElementName=booksDataSource, Path=Data}">

<dataControls:DataForm.ReadOnlyTemplate>

<DataTemplate>

<Grid>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="120" />

<ColumnDefinition Width="10" />

<ColumnDefinition Width="\*" />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="4" />

<RowDefinition Height="\*" />

</Grid.RowDefinitions>

<TextBlock Text="{Binding Title}" Grid.ColumnSpan="3" FontSize="16" FontWeight="Bold" />

<Image Source="{Binding ImageUri}" Grid.Row="2" VerticalAlignment="Top" />

<StackPanel Grid.Column="2" Grid.Row="2" VerticalAlignment="Top">

<TextBlock Text="{Binding Author}" />

<TextBlock Text="{Binding PublishDate}" Margin="0,0,0,10" />

<TextBlock Text="{Binding Description}" TextWrapping="Wrap" />

</StackPanel>

</Grid>

</DataTemplate>

</dataControls:DataForm.ReadOnlyTemplate>

</dataControls:DataForm>

Step 18: We need the ImageUri on Book, which is not part of the data model in the database, but we can add it on the server, since the server knows the topology of the server.

public partial class Book {

public Uri ImageUri {

get {

return new Uri("/BookCovers/" + ASIN + ".jpg", UriKind.Relative);

}

}

}

Because we want to share this computed property with the client, we make the file shared.

Step 18: Lets add editing. So far we’ve only been doing query.

In LibraryManagementService, add an UpdateBook method.

public void UpdateBook(Book book) {

Context.Books.Attach(book, ChangeSet.GetOriginal(book));

// TODO: Email notifications... and other business logic

}

Similarly I can add AddBook, DeleteBook for full CRUD.

Step 19: Lets add some editing UI for the DataForm

<dataControls:DataForm.EditTemplate>

<DataTemplate>

<StackPanel>

<dataControls:DataField>

<TextBox Text="{Binding Title, Mode=TwoWay}" />

</dataControls:DataField>

<dataControls:DataField>

<TextBox Text="{Binding Author, Mode=TwoWay}" />

</dataControls:DataField>

<dataControls:DataField>

<TextBox Text="{Binding Description, Mode=TwoWay}" Height="100" TextWrapping="Wrap"

ScrollViewer.VerticalScrollBarVisibility="Visible" />

</dataControls:DataField>

<dataControls:DataField>

<controls:DatePicker SelectedDate="{Binding PublishDate, Mode=TwoWay}" />

</dataControls:DataField>

<dataControls:DataField Label=" ">

<CheckBox IsChecked="{Binding Missing, Mode=TwoWay}" Content="Missing" />

</dataControls:DataField>

<dataControls:DataField Label=" ">

<CheckBox IsChecked="{Binding OnShelf, Mode=TwoWay}" Content="On Shelf" />

</dataControls:DataField>

<dataControls:DataField>

<TextBox Text="{Binding ASIN, Mode=TwoWay}" />

</dataControls:DataField>

</StackPanel>

</DataTemplate>

</dataControls:DataForm.EditTemplate>

Step 20: Run – some validation – eg. ASIN is not more than 10 characters

Add some validation. On the server, add Book.metadata.cs

This provides a way to add metadata to existing members of another class.

[MetadataType(typeof(BookMetadata))]

[CustomValidation(typeof(BookRules), "EnsureOnShelfIsNotMarkedMissing")]

public partial class Book {

private static class BookMetadata {

[Required]

public static readonly object Title = null;

[Required]

public static readonly object Author = null;

[Required]

[CustomValidation(typeof(BookRules), "EnsureMeaningfulDescription")]

public static readonly object Description = null;

[Required]

public static readonly object ASIN = null;

}

}

Step 21: We can add field level validation, as well as entity level. We can add declarative validation like Required, or write custom validation functions.

Add BookRules.cs

public static class BookRules {

public static ValidationResult EnsureMeaningfulDescription(string description) {

if (String.IsNullOrEmpty(description)) {

return new ValidationResult("Every book needs a description.");

}

string[] words = description.Split(new char[] { ' ', '.' },

StringSplitOptions.RemoveEmptyEntries);

if (words.Length < 10) {

return new ValidationResult("Every book needs a good description. Please supply a paragraph of text.");

}

return ValidationResult.Success;

}

public static ValidationResult EnsureOnShelfIsNotMarkedMissing(Book book) {

if (book.OnShelf && book.Missing) {

return new ValidationResult("A book on shelf cannot be marked as missing as well.");

}

return ValidationResult.Success;

}

}

Step 22: We want them to be run on client as well, as they aren’t doing anything server specific. So make it shared as well.

Step 23: Compile and run

Step 24: Authorization

Add a [RequiresAuthentication] to LibraryManagementService

Run – no data

Change config to have Windows auth

Run – now we have data … also the UI shows me authenticated.

Point out the stuff in the business app template for auth built in that we won’t cover today

Step 25: POCO types

So far we’ve been seeing using a LINQ to SQL entity. However the framework isn’t hard-bound to specific entity types, you can write your own.

Add the following class in LibraryManagementService.cs

namespace Library.DataModel {

public class BookYearStat {

[Key]

public int Year {

get;

set;

}

public int BookCount {

get;

set;

}

}

}

All you need is the key identified at minimum

Step 26: Now I can write a query method that returns these types just like any other.

public IEnumerable<BookYearStat> GetStats() {

var bookGroups =

Context.Books.Where(b => b.Missing == false).GroupBy(b => b.PublishDate.Year);

List<BookYearStat> stats = new List<BookYearStat>();

foreach (var group in bookGroups) {

BookYearStat stat = new BookYearStat() {

Year = group.Key,

BookCount = group.Count()

};

stats.Add(stat);

}

return stats;

}

Step 27: Add the code-behind to load stats into the chart series

public partial class StatsPage : Page {

private LibraryManagementContext \_library;

public StatsPage() {

InitializeComponent();

\_library = new LibraryManagementContext();

}

// Executes when the user navigates to this page.

protected override void OnNavigatedTo(NavigationEventArgs e) {

\_library.Load(\_library.GetStatsQuery(), delegate(LoadOperation<BookYearStat> operation) {

((ColumnSeries)booksChart.Series[0]).ItemsSource = \_library.BookYearStats;

}, null);

}

}

Step 28: Custom method beyond CRUD

Sometimes CRUD is too open-ended. You want more specialized domain operations. Lets add a Checkout function

public void Checkout(Book book) {

Book existingBook = Context.Books.Where(b => b.BookID == book.BookID).FirstOrDefault();

if ((existingBook == null) || existingBook.Missing) {

throw new ValidationException("The specified book does not exist.");

}

if (existingBook.OnShelf == false) {

throw new ValidationException("The specified book is already checked out.");

}

existingBook.OnShelf = false;

book.OnShelf = false;

}

Step 29: Show how it appears on the client

Concepts to talk about/Additional Demos

One unified app across server and client

* No service references required
* No URIs required – instead just instantiated the object
* Directly callable
* Flowing metadata and behavior through shared code – eg. custom validation rule

DomainService to DomainContext pattern

* Show generated code

Metadata pipeline

* Show generated code
* DAL metadata is propagated to server and then to client – for example StringLength constraints were picked up from the DAL and propagated all the way through the entire app

Reusable app logic and app logic pattern

* Ajax … only difference is script vs. managed code… while that is a big difference, usually your core business logic remains the same (Library.Ajax project)
* Server-side rendered HTML, eg. MVC (Library.MVC project)
* Different data technology – might be interested in moving to the cloud – Azure (Library.Azure project)  
  You’ll need an Azure access token to run this … otherwise just show the code.
* What do we get with this DomainContext on the client… why isn’t the model just request/response, but instead a stateful context object … offline (if you want to see the isolated storage file, it will be within C:\Users\<local user name>\AppData\LocalLow\Microsoft\Silverlight\is) (Library.Offline project)
* There is still a services layer… one is created implicitly for you. But you can create one yourself … all layered on top of ADO.NET Data Services and WCF – Astoria service (Library.Service project)