Introduction

John Papa unveiled the "BookShelf" (aka "BookClub") application at PDC 2010 in a talk titled "Kung Fu Silverlight: Architectural Patterns and Practices with MVVM and RIA Services". The same application was featured prominently and repeatedly at the December 2010 Firestarter; every session was recorded and both John's and Dan Wahlin's sessions are worth watching.

John's seminal presentation is a thoughtful explanation and demonstration of the Model-View-ViewModel pattern in working code.He has a great teaching style: intelligent, approachable, and not the least dogmatic. I urge you to watch his one hour video if MVVM is new to you and if you want a guided tour of the original "BookShelf" application.

John designed the application to be indifferent to the modeling and persistence framework. Almost all interactions with the "backend" are abstracted behind a "BookDataService". The UI layers sit "above" the persistence framework, relying upon the BookDataService to retrieve and save BookShelf data.

He built the demonstration application using the WCF RIA Services framework. We, at IdeaBlade, thought you'd like to see this application with DevForce under the hood instead. You can download our version of bookshelf here; you might also enjoy watching a 30 minute video tour of the architecture.

This document describes the steps to migrate John's version to DevForce. The end result is "BookShelfDF", the DevForce application whose code accompanies this document.

In truth, this document goes a beyond the essential steps to get BookShelf running with DevForce. It describes "improvements" that show DevForce makes it easy to fill in some of the gaps in the original demo and tidy up some of the architectural infelicities in the original design. Most of this document – certainly most of the interesting material in this document – is concerned with a discussion of design issues and the changes we recommend. "BookShelfDF" incorporates these recommendations.

Prerequisites

Both versions assume Silverlight 4 and .NET 4. Visual Studio is essential and Blend is highly recommended.

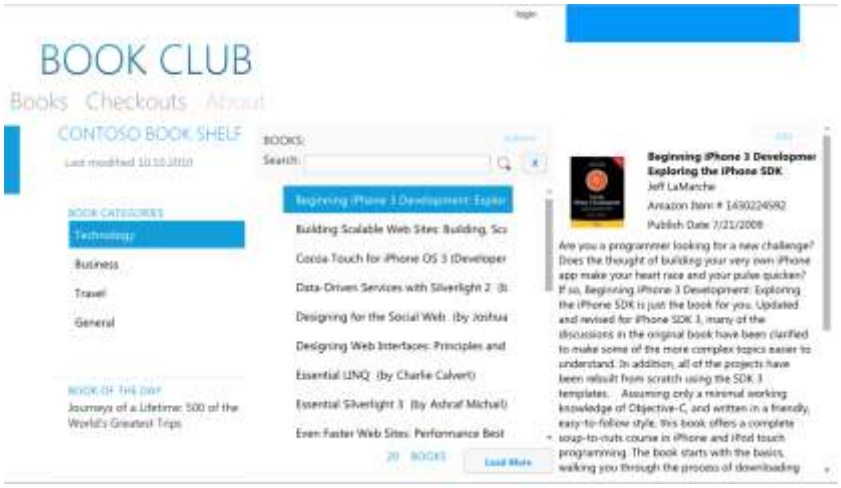
You must have installed **DevForce 2010** version 6.0.7 or later (the free version is fine) to build and run BookShelfDF.

A Quick Look At BookShelf

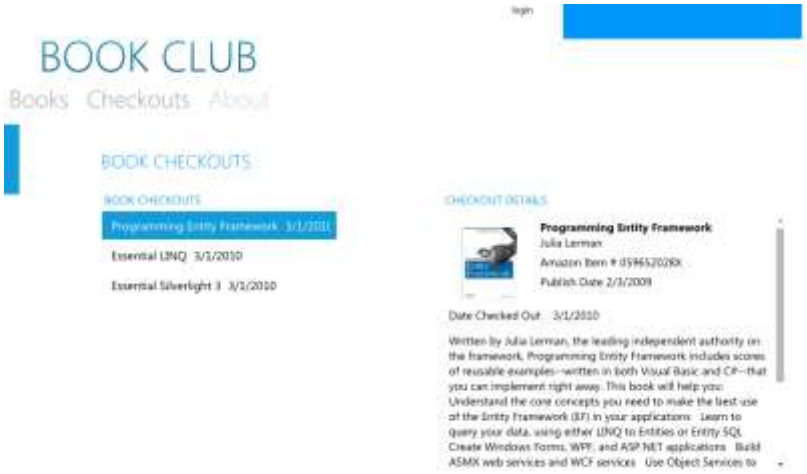
Please watch John's video and consider running either the original RIA-based version or (our preference) run the converted DevForce version, **BookShelfDF** (our preference). The video tour of the architecture may be a helpful prelude.

BookShelf purports to be an application for managing a library of books. The UI presents a "paged navigation" UI metaphor based on the Silverlight Navigation Project Template. You can navigate

between two pages: a searchable list of books “on the shelf” (BookView) and a page of books that have been “checked out” (CheckoutView).

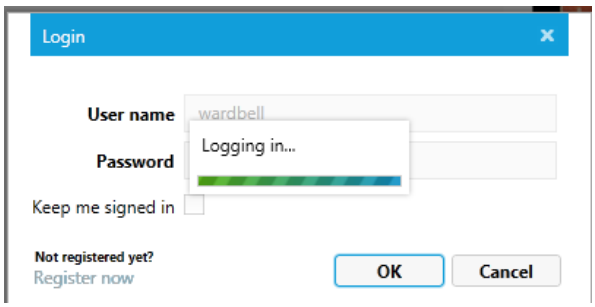


The BookView

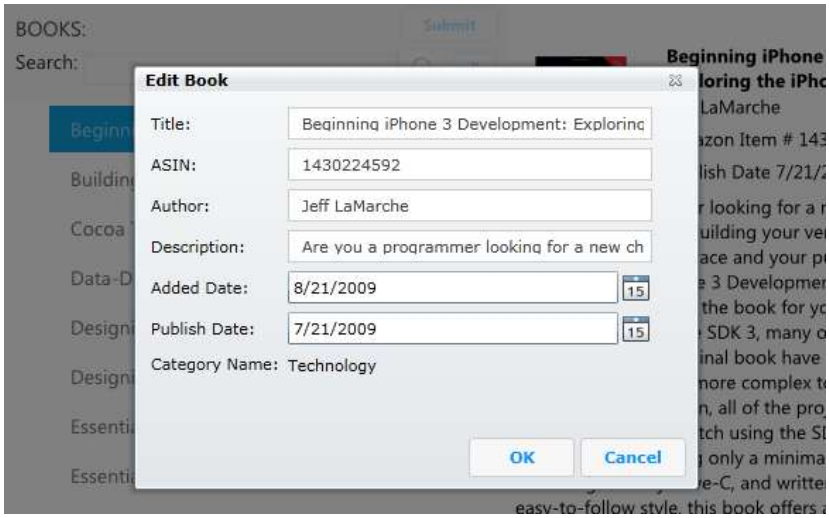


The CheckoutView

There’s a disabled Edit button on the BookView. It is only enabled after you login, which you do by clicking the tiny “login” link at the top of the screen. That link opens a login dialog such as you’d find in the Silverlight Business Application Template.



Now you can click the edit button which results in an overlay child window, the “Book Editor”, focused on the currently selected book.



Agenda

This document is superficially a migration walkthrough. The truly diligent among you could follow it, step-by-step, and arrive at the finished product … although you would still need the finished “BookShelfDF” at your elbow; many of the instructions refer to files in that solution rather than ask you to copy-and-paste from this document.

The document is organized in several major sections:

- Pre-migration preparation
- Remove RIA Services specifics
- Add DevForce specifics
- BookDataService Translation – *where the rubber meets the road*
- Improving the design and implementation
- Suggested refinements
- Silverlight Automated Testing
- Design-time entities from a resource file

The real point of the document is to explain what migration means.

- What parts of the application are RIA Services specific?
- What parts of RIA Services can we get rid of?
- What parts of RIA Services have no analog in DevForce … because they are unnecessary in DevForce?
- What DevForce parts do we add and what do they do?
- How does DevForce change the way you write an application?
- What are some of the new capabilities DevForce makes possible?

Migrating the PDC2010 MVVM BookShelf to DevForce

We recommend that you open the RIA BookShelf in one Visual Studio session, open the DevForce BookShelfDF in another Visual Studio session, and pretend to play along, comparing the two solutions side-by-side.

Pre-migration Preparation

Rename the Solution to "BookShelfDF"

This simple touch helps you quickly distinguish between the DevForce BookShelf Visual Studio session and the RIA BookShelf Visual Studio session.

Remove the TFS dependency.

Not sure of the best way; this way worked.

- Selected all files in Windows Explorer and unchecked "read only" for them and their descendents.
- Removed the Papa.Common.csproj.vspscc
- Removed every <Scc…> tag in all of the projects.
- Removed the "GlobalSection(TeamFoundationVersionControl)" section from BookShelf.sln

SQL Server Database [Optional]

Some of you may prefer to use SQL Server instead of SQL Server Express.

You can copy the database from the AppData directory under the BookShelf.Web project and put it wherever you like; then attach it to your SQL Server.

The shipped SQL Server Express connection string was:

```
  <connectionStrings>

    <add name="BookClubEntities"
connectionString="metadata=res://*/Models.BookClubModel.csdl|res://*/Models.BookClubModel.ssdl
|res://*/Models.BookClubModel.msl;provider=System.Data.SqlClient;provider connection
string=&quot;Data Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\BookClub.mdf;Integrated
Security=True;User Instance=True;MultipleActiveResultSets=True&quot;"
providerName="System.Data.EntityClient" />

  </connectionStrings>
```

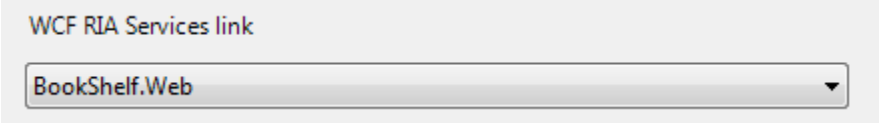
When the database is attached to a local SQL Server, the string is:

```
  <connectionStrings>
    <add name="BookClubEntities" connectionString="metadata=res://*/Models.BookClubMo
del.csdl|res://*/Models.BookClubModel.ssdl|res://*/Models.BookClubModel.msl;provider=
System.Data.SqlClient;provider connection string=&quot;Data Source=.;Initial Catalog=
BookClub;Integrated Security=True;MultipleActiveResultSets=True&quot;" providerName="
System.Data.EntityClient" />
  </connectionStrings>
```

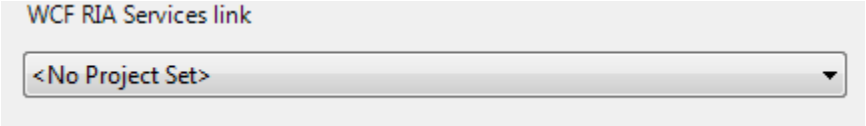
Remove RIA Services Specifics

BookShelf Project

Remove the WCF RIA Services link from the BookShelf project properties



Becomes



Delete hidden “Generated_Code” folder.

Remove references to System.ServiceMode.xxx

Delete Security related services and entities (to be replaced by DevForce versions)

- LoginInfo
- RegistrationDataExtensions
- UserExtensions
- Views/Login folder

Find and remove all “using”s in this project that include the word “DomainServices”

BookShelf.Web Project

Remove references to System.ServiceMode.xxx

Delete the web.config (to be replaced by DevForce version). Everything in BookShelf is vanilla RIA Services BAT so we’re not losing anything that’s not also in DevForce BAT

Delete Security related services and entities (to be replaced by DevForce versions)

- AuthenticationService
- UserRegistrationService
- RegistrationData
- User
- Models/Shared/User.shared
- Resources directory

Migrating the PDC2010 MVVM BookShelf to DevForce

Delete Metadata class file. No modifications to the generated metadata classes; nothing here to transfer to DevForce so delete them.

The "**BookClubService**" has some modifications we'll want, all of them queries. There is no obvious reason to force these methods on the server so we'll move them to the client.

Until we do, we'll set the "Build Action" to "None" to prevent compilation … and we'll keep it around for reference.

- GetBooks() includes Categories and orders by Title
- GetBooksByCategory(int categoryID) fetches Books that match the categoryID. Includes the Category itself and orders by Title.
- GetBookOfDays() performs a "Take(1) after ordering BookOfDay entities by "Day descending". It also includes the related Book entity.
- GetCategories() orders by CategoryName
- GetCheckouts() always includes the Book and Member entities and orders by "CheckoutDate descending"

Add DevForce Specifics

You'll want to borrow files and code snippets from the finished BookShelfDF solution in this and all subsequent sections.

> You could get through this one section without BookShelfDF. In a separate Visual Studio session, create a new application using theDevForce Silverlight Business Application Template (BAT).

BookShelf.Web Project

Open the DevForce Web project in Windows Explorer. We'll drag files and folders into the Visual Studio web project; VS will copy the files from our DevForce source directory as needed.

Drag over:

- Global.asax
- Web.config
- Web.Resources directory
- Services/RegistrationServices to Services/
- Models/… to Models/
- log/ (which should be empty … delete contents if it is not)

Fix the Web.Resources

[This step should be unnecessary in a near-term future version of DevForce]

- Delete the 2 compiled files (xxx.cs)
- Set the custom tool in their properties to "PublicResXFileCodeGenerator"; the compiled files will generate immediately

Generate DevForce EF entities

- Open the BookClubModel.edmx
- Update from Database (to ensure we get the connection string into the Web.config; do not actually make changes).
- Save – will take a little time as DevForce Object Mapping extension adds references and then generates the code

Pull up "BookClubModel.Shared.cs out of Shared folder.

Add DevForce server references

- IdeaBlade.EntityModel.Edm
- IdeaBlade.EntityModel.Server
- IdeaBlade.EntityModel.Web

Select all DevForce references and, in the property sheet, set

- **Copy Local = true ← CRITICAL**
- Specific Version = false

Make sure Silverlight debugging is enabled: look at Web project properties / Web / Debuggers and be sure **both ASP.NET and Silverlight are checked**. Telltale sign of trouble is the appearance of the "Script Documents" pseudo folder when you run.

BookShelf.Web should now BUILD SUCCESSFULLY !

BookShelf Project

Open the DevForce Web project in Windows Explorer. We'll drag files and folders into the Visual Studio web project; VS will copy the files from our DevForce source directory as needed.

Model fix-up

DevForce model generation resulted in a linked file, BookClubModel.IB.Designer.cs.

BookShelf project keeps all Model stuff in the model folder so drag it there.

Add links to the server-side "User.cs" and "BookClubModel.shared.cs" files

Web/Resources fix-up

The resx file links are broken because the server-side "Resources" directory is now called "Web.Resources".

[This step should be unnecessary in a near-term future version of DevForce]

- Delete the 2 broken resx file links
- Add the 2 corrected resx links.
- Set the custom tool in their properties to "PublicResXFileCodeGenerator"; the compiled files will generate immediately

Login

- Drag **AuthenticationManager** into Services folder
- Search-and-replace all "WebContext" with "AuthenticationManager" (3 files)
- Drag Views/Login into Views
- Drag Models/Login folder into Models
- Add links to Server-side Models/Login files
    - CreateUserResult
    - RegistrationData
    - VerifiableObject

- MainPage.cs: LoginStatus is now under BookShelf.Views.Login rather than BookShelf.LoginUI so correct “using” to read “using BookShelf.Views.Login”.

- Open `Styles.xaml`
  and replace `xmlns:loginWindow="clr-namespace:BookShelf.LoginUI"`
  with `xmlns:loginWindow="clr-namespace:BookShelf.Views.Login"`

- Make the Bookshelf assembly’s internals visible to .NET serialization libraries so that Silverlight can call the internal setters in *CreateUserResult*. Do this by adding the following three lines to the bottom of the BookShelf project’s *AssemblyInfo*.

```
// These allow the expression tree built by DevForce for certain complex queries to be pr
ocessed correctly in Silverlight.  Without it, some complex queries may fail with a Metho
dAccessException.
[assembly: InternalsVisibleTo("System.Core, PublicKey=0024000004800000940000000602000000240
00005253413100040000010001008d56c76f9e8649383049f383c44be0ec204181822a6c31cf5eb7ef486944
d032188ea1d3920763712ccb12d75fb77e9811149e6148e5d32fbaab37611c1878ddc19e20ef135d0cb2cff2b
fec3d115810c3d9069638fe4be215dbf795861920e5ab6f7db2e2ceef136ac23d5dd2bf031700aec232f6c6b1
c785b4305c123b37ab")]
[assembly: InternalsVisibleTo("System.Runtime.Serialization, PublicKey=0024000004800000940
000000602000000240000525341310004000001000100 8d56c76f9e8649383049f383c44be0ec204181822a6
c31cf5eb7ef486944d032188ea1d3920763712ccb12d75fb77e9811149e6148e5d32fbaab37611c1878ddc19e
20ef135d0cb2cff2bfec3d115810c3d9069638fe4be215dbf795861920e5ab6f7db2e2ceef136ac23d5dd2bf0
31700aec232f6c6b1c785b4305c123b37ab")]
```

App.Xaml.cs

DevForce doesn’t use the same start-up for authorization.

Add “using IdeaBlade.EntityModel;”

Replace the following as shown

- App()
- Application_Startup(…)
- Application_UserLoaded(…)

```
/// <summary>
/// Creates a new <see cref="App"/> instance.
/// </summary>
public App()
{
  InitializeComponent();
}

private void Application_Startup(object sender, StartupEventArgs e)
{
  // This will enable you to bind controls in XAML files
  // to AuthenticationManager.Current properties.
  this.Resources.Add("AuthenticationManager", AuthenticationManager.Current);

  // This will automatically authenticate a user when using Windows authentication
  // or when the user chose "Keep me signed in" on a previous login attempt.
  AuthenticationManager.Current.LoadUser(Application_UserLoaded);

  // Show some UI to the user while LoadUser is in progress
  this.InitializeRootVisual();
}
```

```
/// <summary>
/// Invoked when the <see cref="BaseOperation"/> completes. Use this
/// event handler to switch from the "loading UI" you created in
/// <see cref="InitializeRootVisual"/> to the "application UI"
/// </summary>
private void Application_UserLoaded(BaseOperation operation) { }
```

Miscellaneous Steps Before BookDataService Translation

Global Search and Replace of the DevForce template application name (e.g., “DevForceBat”) to “BookShelf” . If we compile the application now, all remaining errors (12 by my count) concern Services - the more substantive difference between RIA and DevForce

BookDataService Translation

Now it gets interesting.

Save method of the IBookDataService

BookViewModel and CheckoutViewModel classes both depend upon "Operation" coordination objects for Save method. That's the way the IBookDataService defines the Save method.

It would be better to abstract away that dependence and also give caller a happy path and (optional) sad path (OnSuccess, OnFail)

We'll stay with the current interface for now and just replace the operation object type to the DevForce flavor. We'll revise it after we get it all working.

- Add "using IdeaBlade.EntityModel;"
- Change Save method in interface to

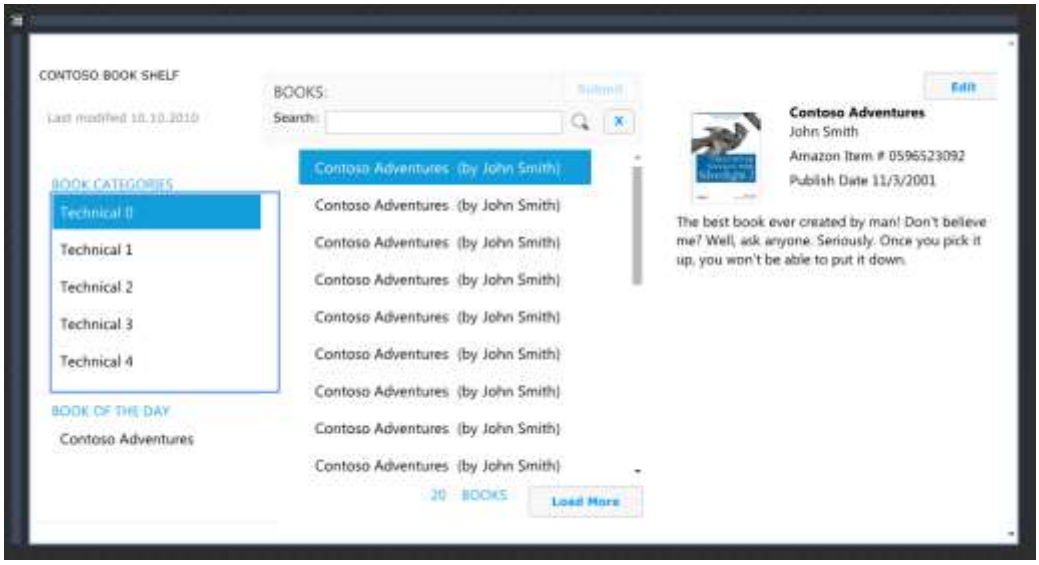
```
void Save(Action<EntitySaveOperation> saveCallback, object state);
```

- Change BookViewModel and CheckoutViewModel to use this type which involves adding that IdeaBlade "using" as well
- Change the DesignBookDataService in the same manner

The two IBookDataService implementations will remain.

After Build, all remaining errors are in the "production" BookDataService … right where we want them!

DesignDataService now works and we can see that by looking at BookView.xaml in Cider or Blend



BookDataService

The author's goal was to minimize the dependence on any particular persistence framework – RIA Services in particular.

The RIA BookDataService depends upon query specifications defined in the DomainService. We frown on that in DevForce, preferring to move those specs to the client where they are motivated. The actual query is still performed on the server; we're just moving the definition.

Make a disabled copy for reference

- Copy and paste BookDataService
- Rename it "BookDataService_DELETE_SOON.cs
- Set its Build Action to None
- Open it in Visual Studio for reference
- Open the Web Project's BookClubService_DELETE_SOON for reference as well

Replace the "Context" definition

- Delete unused "usings", especially "using BookShelf.Web.Services"
- Add DevForce usings
    - IdeaBlade.EntityModel;
    - IdeaBlade.Core;
- Replace "BookClubContext" definition with "BookClubEntities"
- Replace Context.PropertyChanged with Context.EntityChanged
- Update ContextPropertyChanged handler signature to

```
private void ContextPropertyChanged(object sender, EventArgs e)
```

- DevForce has a method, not a property: "Context.HasChanges()"
- Move the ContextPropertyChanged method up and under the ctor for easier understanding and to keep the query method section clean.

Replace the Save Method

- Replace "SubmitOperation" with "EntitySaveOperation"
- DevForce has a method, not a property: "Context.HasChanges()"
- DevForce verb is "SaveChangesAsync"
- Move "Save" above the queries so it doesn't get lost.

The result looks like this:

```
public void Save(Action<EntitySaveOperation> saveCallback, object state)
{
    if (Context.HasChanges())
    {
        Context.SaveChangesAsync(saveCallback, state);
    }
}
```

Query Methods

The RIA approach to querying is superficially the same as the DevForce approach.

The most important difference is that RIA wants the developer to work directly with the cache of entities of each type (e.g., Context.Books) as if they were the query results.

That creates confusion. The cache and query results are not the same thing.

It also means that the developer is encouraged to clear entity caches to avoid that confusion. Exhibit number one is the ClearBooks method … which also resets the Page index every time even though that is meaningful only for GetMoreBooksByCategory

We note also that every BookShelf query involves a trip to the server … even though it is unlikely that much would have changed since the last query.

The caching decision

We agree that the full application should manage its local cache. We don't think it should grow endlessly and there should be some provision to refresh with the latest information entered by other users. Such refresh can be automated or on demand.

This is a demo … we'll assume that refreshing is a topic for another time. We'll let the application remember prior queries and accumulate books throughout the session. This is by no means necessary and can be revisited easily at a later date.

Replace ClearBooks() with ResetPaging()

We don't need to clear the cached books for reasons just given. We'll preserve the application's rule about when to reset the PageIndex … and rename the method accordingly.

Query Error Handling

There isn't any in this demo. If a query fails, the exception will bubble up to the application's "Unhandled Exception" handler in the App.xaml.cs.

Not our favorite approach but we get it … and we'll stay with it.

Delete the private Operation variables

They serve no useful purpose

Delete the private Callback variables

We will use lambdas and won't be needing these. Actually, we would have a subtle bug if we did use them … as discussed in "GetCheckOuts".

GetCheckOuts – our first query conversion

We'll use this one to establish the pattern for future queries.

The query specifics were defined in the DomainService. Let's bring those over, redefining "query" variable:

```
var query = from checkout in Context.Checkouts.Include("Book").Include("Member")
            orderby checkout.CheckoutDate descending
            select checkout;
```

The only change to the copied DomainService method was replacing "ObjectContext" by "Context".

Now we'll invoke it async with a lambda for the callback

Delete unused OnLoadCheckoutsCompleted.

Entire method looks like this:

```
public void GetCheckouts(Action<IEnumerable<Checkout>> getCheckoutsCallback)
{
    ResetPaging();

    var query = from checkout in Context.Checkouts.Include("Book").Include("Member")
                orderby checkout.CheckoutDate descending
                select checkout;

    query.ExecuteAsync(op => {
          if (!op.CompletedSuccessfully) return; // let app handle exception
          getCheckoutsCallback(op.Results);
          });

}
```

Don't like lambdas? Why? They're so easy. And they alleviate the risk posed by holding on to the caller's callback method in a member variable (e.g., _getCheckoutsCallback) that accidentally could be replaced by a double call! Preventing the double call … or its consequences … is a lot of work.

I could show you how to do it the hard way if you insist … but that will be another lesson for another time.

Refactor to RunQuery<T>

Looking over the query methods it looks like all of them are going to end up with code that looks like

```
    query.ExecuteAsync(op => {
          if (!op.CompletedSuccessfully) return; // let app handle exception
          getCheckoutsCallback(op.Results);
          });
```

In fact, author was part way there when he wrote **RunBooksQuery** and reused it in three of the queries. He didn't refactor far enough, most likely because the many moving parts - adding and removing completed handlers and holding on to entity-specific callback methods in member variables – these mechanics clouded his vision of the opportunity.

Because our approach is simpler, we can share a common “RunQuery” method

```
private static void RunQuery<T>(
IEntityQuery<T> query, Action<IEnumerable<T>> queryCallback)
  {
    query.ExecuteAsync(op => {
      if (!op.CompletedSuccessfully) return; // let app handle exception
      queryCallback(op.Results);
    });

  }
```

GetBooksByCategory and GetMoreBooksByCategory

The central logic of these two methods can be combined into a single method which can absorb RunBooksQuery as well.

Known Issues

Cancel in Editor really doesn't Cancel

It doesn't in the original either. There is no ViewModel behind the editor; everything is in code-behind. And nothing is doing a thing with the result of either button click. We'll add cancel/rollback logic later.

Logout doesn't disable Submit button

Logout disables the Edit button but not the Submit (which is connected to the "Has Changes").

We will address this in an optional section of this document by changing the CanSave logic in BookViewModel to pay attention both to HasChanges and to IsAuthenticated.

Application freezes in the BookEditor

It doesn't happen every time. But the experience is easy to reproduce as follows:

1. Login
2. Go to Checkout view
3. Return to Books view
4. Press Edit button … editor opens …

The application is frozen solid. You can't even close the browser. You'll have to terminate the debugging session in Visual Studio.

Debugger reveals that a new edit window is created ***for each instance of the BookView***.

> *Put a breakpoint on OnLaunchEditBook() to confirm. Pausing for the breakpoint gives you a chance to see the two Editor windows and you can work through it by canceling both; the UI freeze up occurs when there is no debugger window to stir up the physical display. You're stuck with no choice but to terminate debugging. Grrrr.*

The navigation framework that came with the application template creates a new view every time you arrive on a "page". So navigating from Book to Checkout and back to Book again creates a second BookView instance.

Sadly those BookViews are kept alive by that same navigation framework.

We don't know why. The Messenger shouldn't be the cause; it's Registration method maintains weak references to the registering object (the BookView in this case). Something, somewhere is keeping the first BookView alive after it has disappeared from the screen. The weak reference never becomes a dead reference. Therefore, the Messenger retains knowledge of both first and second BookView instances, both view instances hear the LaunchEditBookMessage, and both try to open a BookEdit window. Do it a few more times and the problem multiplies

A similar danger lurks for the SavedBookDialogMessage message as well.

Solution #1: Unregister when the user navigates away. To be crystal clear about what is happening when, we'll move the registration and add deregistration to BookView.xaml.cs as follows

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
  RegisterMessages();
}

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
  UnregisterMessages();
}

private void RegisterMessages()\
{
  Messenger.Default.Register<LaunchEditBookMessage>(this, OnLaunchEditBook);
  Messenger.Default.Register<SavedBookDialogMessage>(this, OnSaveBookDialogMessageReceived);
}

private void UnregisterMessages()
{
  Messenger.Default.Unregister<LaunchEditBookMessage>(this, OnLaunchEditBook);
  Messenger.Default.Unregister<SavedBookDialogMessage>(this, OnSaveBookDialogMessageReceived);
}
```

This approach gives us insight into probable memory leaks with the Navigation framework … or at least with this application's use of that framework.

Later we'll look at a better ~~workaround~~ solution for the message registrations and handlers. We won't have time to figure out why old views are neither re-cycled nor disposed of properly.

This author recommends against using the Navigation framework in a production application and would consider Prism v.4 as an alternative.

Improving the Design and Implementation

In this section we move beyond what is necessary to make the application work with DevForce and consider how we could simplify matters and add a few features to make it more robust.

Simplification is the first priority. “Simplification” is an elusive term; everyone claims it for themselves but it is rarely measurable. We will try. Our tangible measures of simplification:

- Fewer classes
- Fewer class members
- Eliminate unused parameters
- Fewer lines of code

The intangible measure are just as important. Sometimes we think we can find names that better express intent.

These measures of simplicity are not always accurate. The “Extract Method” refactoring, in which you pull one or more lines of code out and make a separate method of them, certainly adds code even as it improves readability (an important simplicity intangible). We’ll have to use good judgment.

Simplicity isn’t the only thing either. We want to make the application a little more robust. We’d like to better account for some of the error conditions that are sure to arise. Error handling clouds the design. But we can’t pretend forever that it is unimportant. We should design for it and do our best to make that as simple as we can.

Without further ado …

Simplify ServiceProviders using EntityManagerProvider and one DataService

Delete DesignBookDataService; we’ll only use the BookDataService now.

BookDataService changes behavior based on its EntityManager. DevForce entity caching and offline moe makes this work.

BookDataService takes an EntityManager parameter (more precisely a BookClubEntities parameter) that facilitates testing and design modes.

The ServiceProvider (acting like an IoC) creates the BookDataService with an EntityManager it acquires from an EntityManagerProvider.

Add two EntityManagerProviders that implement the single-method IEntityManagerProvider interface. The production one is the trivial EntityManagerProvider. The design version, DesignEntityManagerProvider, creates an offline EntityManager which it populates with design-time entities. These entities come from the original Design classes (more about them shortly).

Simplify ServiceProviderBase by chucking out the dead wood.

- No more virtual PageConductor and BookDataService properties and no virtual methods in constructors.
- All public instance properties have protected setters, to be set by their derived classes
- Delete the commented out code.
- Add the new EntityManagerProvider property

DesignServiceProvider and ServiceProvider simplified to default ctors with different PageConductor and EntityManagerProvider initializations.

Design Data Classes

Tweaked some of the Design classes (e.g., DesignBooks) to ensure (a) that they all have unique keys and (b) that related entities are established by means of FKs rather than entity assignment. These entities are not part of a manager while being constructed; can neither set their navigation properties nor use them until they are attached to a manager.

The Design classes return ObservableCollection<T> because they used to participate in binding. For our purposes they only have to be IEnumerable<T> but we'll leave them as OC<T>.

Revising the BookDataService

The API of the BookDataService is always a work in progress. It stands at the boundary between the needs of the UI and the capacities of the backend data service. UI requirements pull in one direction; performance and model business logic pull in another direction.

I think of a data service as belonging to the UI. The UI is the "customer" . The customer is always right … is always king. But if you've ever run a business, you know there are limits. The customer does not always know what's best for him. You can't perform magic. There is always room for negotiation.

Our eye turns to IBookDataService with all of this in mind. What do we see?

- GetBooksOfTheDay can only return one book but it's written to return a list
- No provision for failures in the async signatures
- Query parameters precede callbacks which means we can't add a failure callback in the right place and we can't provide default values for callback parameters
- The Save method signature:
    - takes a "State" parameter that no one uses
    - depends unnecessarily on a persistence class, EntitySaveOperation

GetBookOfTheDay replaces GetBooksOfTheDay

Seems silly to get a one element list. Why do that? It really should be BookOfTheDay.

Why is the RIA example plural? Perhaps because it's hard to return a single book asynchronously? Easy in DevForce.

We deprecate the GetBooks… (plural) in favor of GetBook… (singular)

- In **IBookDataService** and **BookDataService**
  - Add GetBookOfTheDay
  - Comment out GetBooksOfTheDay

- In the **BookView**
  - Comment out "BookOfDayItemTemplate" DataTemplate
  - Comment out <ListBox > bound to BooksOfTheDay
  - Replace with:
    <TextBlock Text="{Binding BookOfTheDay.Book.Title}"
    Style="{StaticResource ContentTextStyle}"/>

- In **BookViewModel**
  - Comment out
    - BooksOfTheDay in the BookViewModel
    - BooksOfTheDay the ObservableCollection stuff
    - LoadBookOfDays
    - "LoadBookOfDays" call in LoadData()
  - Add BookOfTheDay property
  - Add LoadBookOfTheDay method and its callback
  - Add call to LoadBookOfTheDay to LoadData()

Rearrange Signatures

I want to move the callbacks to the end.

This could be tedious without refactoring tools. It's quite easy with them. Use your tool of choice for each method in IBookDataService: GetBooksByCategory, GetMoreBooksByCategory, GetBooksByTitle

Add optional FailCallback to interface and implementation

The ViewModel should determine what should happen if the service fails.

The service succeeds most of the time and the application may not know what to do if it crashes. But the ViewModel should be able to address that possibility and we should structure the signatures to make the less-frequent failure path as unobtrusive as we can.

To each method wel add a fail callback parameter that takes an Action<Exception>. We set the parameter to equal null so that ViewModels don't HAVE to provide one.

This is an opportunity to introduce one of my naming preferences. In the current API, the callback parameter name explains what the callback should do. In my opinion, this doesn't add much information and makes the code a little harder to maintain. I also don't want to work that hard to follow that pattern in my "fail" callback naming.

My convention is to name the parameter "onSuccess" and "onFail". I want to stress that this is my personal preference, not a criticism of the original names.

Adjust **IBookDataService** accordingly.

Go to BookDataService and adjust those signatures (but omit the "=null" defaulting for onFail callbacks).

> The application should still build and run.

Change RunQuery to take an onFail callback … and implement it

Account for the fact that the onFail method might be null.

Important: if the asynchronous operation failed, DevForce (like RIA Services) will re-throw the exception – and mostly likely crash the application – unless we handle it … and tell DevForce we handled it. If the caller supplies an onFail callback, we'll call MarkAsHandled() to save the caller the trouble.

While we're here, we'll also guard against a possible null onSuccess callback.

```
private static void RunQuery<T>(
  IEntityQuery<T> query,
  Action<IEnumerable<T>> onSuccess,
  Action<Exception> onFail) {
  query.ExecuteAsync(op =>
  {
    if (op.CompletedSuccessfully) {
      if (null != onSuccess) onSuccess(op.Results);
    } else {
      if (null != onFail) {
        op.MarkErrorAsHandled();
        onFail(op.Error);
      }
    }
  });
}
```

Now let the compiler guide you back through the class to add ", onFail" to all "RunQuery" calls.

Add "onFail" to GetBookOfTheDay

GetBookOfTheDay returns a single value instead of a list so it doesn't use "RunQuery"; it calls on of the scalar asynchronous execution methods.

We only have one of these scalar queries so we don't have to generalize how we handle the result.

> We could generalize for scalar queries. We'd have to switch from the callback to event-based style of handling asynchronous calls. It's a lesson for another time.

We'll follow the same "if … else …" pattern, adapted for the scalar query.

```
  .FirstOrNullEntity(op =>
    {
       if (op.CompletedSuccessfully) {
          if (null != onSuccess) onSuccess(op.Result);
       } else {
          if (null != onFail) {
```

```
                op.MarkErrorAsHandled();
                onFail(op.Error);
            }
        }
    });
```

The application should still build and run.

Revise Save to conform to "onSuccess, onFail" signature

We observed earlier that the BookDataService's Save method has a state parameter that no one is using. We'll remove that from the interface in a moment.

The Save method signature also specifies a delegate. Unlike the query methods, this callback has a parameter with a "funny" type. It's the persistence layer's "operation coordination object", *EntitySaveOperation* in DevForce and *SubmitOperation* in RIA Services.

The Save callers in this application are only using that operation object to determine if the process succeeded or failed. We have that distinction covered with our "onSuccess, onFail" approach.

We'll update the BookDataService to play along and adjust the ViewModels as well. They will no longer know about operation coordination objects.

The revised signature: `void Save(Action onSuccess, Action<Exception> onFail);`

The implementation:

```
public void Save(Action onSuccess, Action<Exception> onFail)
{
    Context.SaveChangesAsync(op => SaveCallback(op, onSuccess, onFail));

}

private static void SaveCallback(
    EntitySaveOperation op, Action onSuccess, Action<Exception> onFail)
{
    if (op.CompletedSuccessfully) {
        if (null != onSuccess) onSuccess();
    } else if (null == onFail) {
        return;
    } else if (op.HasError) {
        op.MarkErrorAsHandled();
        onFail(op.Error);
    } else {
        onFail(new InvalidOperationException("Save cancelled"));
    }
}
```

The SaveCallback reshapes the result of the DevForce asynchronous save (which has three possible outcomes) into the onSuccess and onFail invocations that the BookDataService caller expects.

> The Save method in the final BookShelfDF is a bit different, reflecting a subsequent refactoring to support testing that is described in the Silverlight Automated Testing section below.

The updated BookViewModel:

```
private void OnSaveBooks()
{
    BookDataService.Save(SaveSuccess, SaveFailed);
}
private void SaveSuccess()
{
    // reconsider whether confirmation dialog is a good experience
    ShowMessage("Save was successful", "Save");
}

private void SaveFailed(Exception ex)
{
    ShowMessage("Save was unsuccessful" + Environment.NewLine + ex.Message,
                "Save Failed");
}

private void ShowMessage(string message, string title="Information")
{
    var dialogMessage = new SavedBookDialogMessage(message, title);
    Messenger.Default.Send(dialogMessage);
}
```

Is that simpler? We've grown from 11 to 21 lines.

Simpler isn't always shorter and it's not always better. We discovered a few things in our investigation of the save process:

- Maybe the success path should behave differently than the fail path. Currently we present a confirmation dialog box that forces the user to click OK after a save. Many people feel that's a bad experience; the application should trigger a more subtle confirmation clue.
- We can provide the user with more useful information when the save fails.
- We can reuse the last six lines of the extracted "ShowMessage" method when we implement failure logic for our query methods as we will do below.

Generalize Save Handling with new BookSaver Class

CheckoutViewModel can save too (or so it seems). It's save logic is identical.

You could copy and paste the code; maybe that's ok because you've convinced yourself that you'll never save a third time.

Or you could pull the logic into the BookDataService itself. I'm reluctant to do that because it troubles the data service with user experience responsibilities.

I decided to pull all of the logic into a helper class, **BookSaver**, that I've added to the Services folder. The ViewModels instantiate it by passing in their BookDataService, then call its Save method at the appropriate time.

Now both BookViewModel and CheckoutViewModel have the same *delegated* implementation.

Generalize SavedBookDialogMessage Class and rename it "ConfirmationDialogMessage"

The BookViewModel and CheckoutViewModel soon will tell the user about query failures too. How might they do that?

This application currently presents save success and error notifications in a MessageBox that the user dismisses by clicking "OK". If the MessageBox is good enough for save notification it should be good enough for all kinds of notifications.

Let's see how it works … which seems odd at first:

```
var dialogMessage = new SavedBookDialogMessage(message, title);
Messenger.Default.Send(dialogMessage);
```

Messenger is a MVVM Light component for sending intra-application messages in a loosely-coupled fashion that you may recognize as an example of the "Event Aggregator" pattern.

The code sends a message of type "SavedBookDialogMessage". Something must be listening for a message of that type. The *something* is the **BookView**;. The code-behind in BookView.cs registers for this message type and does the job of displaying a dialog box from the instructions in that message instance.

It's a pretty round-about way to the user's attention. We won't quarrel with it. In fact, we'll re-use it … and we'll simplify the sending protocol so we can compose and send the message in a single step.

- Refactor the name from **SavedBookDialogMessage** class to **ConfirmationDialogMessage** (make sure all the affected classes catch up; the application should still build).
- Make it a public class
- Give it a static "Send" method so it can send instances of itself via "Messenger.Default"
- Make the "title" parameter optional with a default value of "Information"

The new syntax looks like this:

```
ConfirmationDialogMessage.Send("Save was successful", "Save");
```

Update the **BookSaver**'s *SaveSuccess* and *SaveFailed* methods to use it.

Add the ErrorReporter class

We are reporting a Save error in the BookSaver. We expect to report query errors in the BookViewModel and in the CheckoutViewModel. And we expect to use the same user notification technique in all three places.

It's time for a helper class: the **ErrorReporter** with two ***ShowError*** methods:

```
public void ShowError(
   Exception ex, string operationDescription = null, string title = null)
{
  operationDescription = operationDescription ?? "A service operation";
  var exMessage = (null == ex)
```

```
        ? " failed or was cancelled."
        : " failed with exception: " + Environment.NewLine + ex.Message;
  var msg = operationDescription + exMessage;
  ShowError(msg, title);
}

public void ShowError(string msg, string title = null)
{
  ConfirmationDialogMessage.Send(msg, title ?? "Error");
}
```

Add query failure notification to the BookViewModel

It's time for the BookViewModel and CheckoutViewModel to do something when the query fails. In this example, we'll just notify the user with our new **ErrorReporter**.

Add an ErrorReport property that is initialized in the constructor; we'll call it within a new ReportError method described below.

Wire up all of the query ("Load") methods appropriately as in this example:

```
public void LoadBookOfTheDay()
{
  BookDataService.GetBookOfTheDay(
    book => BookOfTheDay = book,
    ReportError("Book-of-the-Day loading"));
}
```

Add a **ReportError** method:

```
private Action<Exception> ReportError(string message)
{
    return ex => ErrorReporter.ShowError(ex, message);
}
```

Functional Programming Sideshow

ReportError is an example of functional programming for the purpose of sweetening the callback syntax.

It "curries" an action of two parameters (the exception and the message) into an action of one parameter, Action<Exception>. Because GetBookOfTheDay expects an Action<Exception>as its second parameter, the compiler lets us use ReportError without the ugly lambda arrow syntax.

If you feel this is too tricky, you go "old school" and write ReportError like this:

```
private void ReportError(Exception ex, string message)
{
    ErrorReporter.ShowError(ex, message);
}
```

Then use it as shown here:

```
public void LoadBookOfTheDay()
{
  BookDataService.GetBookOfTheDay(
```

```
        book => BookOfTheDay = book,
        ex => ReportError(ex, "Book-of-the-Day loading"));
    }
```

This author prefers the tasty “curried lambda”.

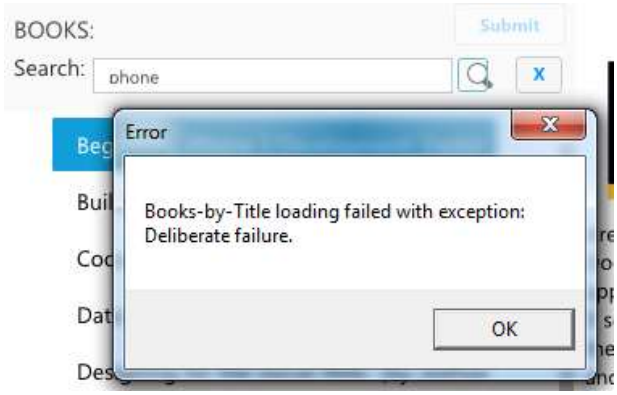
onFail in Action

We’d like to see if our efforts are worthwhile when failure comes … as it must. BookShelf lacks automated testing at the moment. We can modify the **BookDataService** so that one of its queries appears to fail.

Add the following lines to **GetBooksByTitle**, right after “`ClearPaging();`”

```
// Try onFail
onFail(new InvalidOperationException("Deliberate failure."));
return;
```

Run the application. The other queries should be fine. Enter something in the Search TextBox and click the magnifying glass; you should see the expected pop-up message:



Don’t forget to remove the test code or comment it out.

Update the CheckoutViewModel

We make the same changes to the CheckoutViewModel. It’s only query is in LoadData().

The **CheckoutView** isn’t yet wired to show a **ConfirmationDialogMessage**.

We could follow the pattern for **BookView** (see the repair we made to the BookView code-behind in an earlier section). But that would only render more odious what is already a serious code smell.

Move Confirmation Message Display to MainPage

We settled upon the Confirmation Dialog as our common user notification experience. It should be available everywhere. Individual views need not be concerned with it.

As it stands, the BookView is listening for the ConfirmationDialogMessage. It shouldn’t have to. Worse, it is listening in its code-behind (yuck) and we had to add message registration and unregistration goop to cope with the way the Navigation framework keeps the page alive long after it has disappeared.

We're considering spreading this disease to other Views.

Fortunately, we can move the feature to the MainPage, the root visual which is always present, and remove the smell from this view.

> We're relocating the smell to MainPage but at least it's only in one view; we won't propagate it across all of our views.

- Add the ConfirmationDialogMessage registration to the constructor of MainPage.xaml.cs
- Move the DialogReceived method to MainPage.xaml.cs
- Delete everything having to do with ConfirmationDialogMessage from BookView.xaml.cs

Suggested Refinements

Everything in this section is optional.

Some recommendations are purely a matter of taste and coding style.

Some recommendations address weaknesses in the current design that you may not care about. Suggestions of this type almost always involve more code and another new class (see Authentication Revisions).

All are reflected in the revised code base.

Clean the ViewModel Query Callbacks

The callbacks in the BookViewModel and CheckoutViewModel are ungainly to my eye. A touch of LINQ crisps them up.

Replace the "`foreach`" statements with the DevForce `IdeaBlade.Core.Linq.ForEach` extension method. It eliminates lines of noise, bringing the essential logic more clearly into view.

Use `FirstOrDefault` to replace `someCollection[0]`.

Remove or reverse the null value tests and eliminate more noise.

Clean the ViewModel Base Class

Remove empty virtual methods

ViewModel is the abstract base class for all ViewModels.

It defines two protected virtual "Register" methods – RegisterCommands and RegisterMessages – that it calls in the constructor. It also defines an empty public LoadData method.

The design intention is to remind the developer that ViewModels tend to have commands and messages. The constructor calls these Register methods.

Several problems with these methods:

1. They have no implementations and therefore do nothing.
2. One should avoid calling virtual methods in constructors because there are known problems with getting the sequence right for all contributors to the virtual method across the class hierarchy. Resharper warns you about this. While there is no actual problem here (the base methods are empty), it's a design that begs for trouble.
3. Virtual methods always add complications; the reader must look across the call hierarchy to be sure of the class behavior. We should use them when we need them, not if we don't.

4. The LoadData method is potentially callable by some external force (a test method?) because it is part of the base class which could be seen as a general ViewModel API. However, nothing actually calls it and the derived classes are calling it in their constructors (see #2).
5. **The base methods are never actually called by any ViewModel constructor!**
6. **None of these methods is ever referred to except by a derived class!**

We get rid of them

- Remove the Register and LoadData methods from **ViewModel**
- Remove the "overload" keyword from the corresponding methods in **BookViewModel** and **CheckoutViewModel**.
- In CheckoutViewModel, LoadData only called LoadCheckouts; fold LoadCheckouts into LoadData.

Remove ServiceCall tracking

Not sure what this is all for. It appears to be the remains of a mechanism to support a sequence of async calls that all must complete before the UI can proceed.

It is not used in this example. We don't need that feature ever. We have DevForce Coroutines. So we remove these parts.

Our ViewModel class has been reduced to

- A provider of "IsBusy" services
- An implementor of INotifyPropertyChanged (by way of its base class)

It probably isn't worth having such a small class complicate the inheritance hierarchy but we'll leave it for now.

Security Checks in the ViewModel, not the View

The BookView.xaml controls whether the Edit button is enabled by checking the singleton instance of the AuthenticationManager that was added to the application resources in App.xaml.cs.

> The DevForce AuthenticationManager replaces the RIA Services WebContext

Here is the pertinent line in the view, setting the IsEnabled attribute of the Edit button.

```
IsEnabled="{Binding Path=User.IsAuthenticated, Source={StaticResource AuthenticationManager}}"
```

I feel rather strongly that checking for an authenticated user is not a View concern. That kind of logic belongs in the ViewModel.

I could hold my nose and let it pass. But I noticed that the Submit button would save changes from a logged out user. One could:

- Login (which enables editing)

- Prepare changes
- Logout
- Submit [!?!]

Of course the server should guard against unauthorized saves no matter what the client does. But the client should be well behaved and block this kind of thing.

Moreover, we're sure to add more features that are enabled only for a user who is authenticated and authorized. The checking logic shouldn't be in the view.

We won't want that `IsEnabled` attribute when were done. Be bold! We'll rely on command binding. Delete the line with that attribute now.

Be bolder still and delete the line in App.xaml.cs which adds the AuthenticationManager as a resource.

Both the Edit and Submit buttons are bound to **RelayCommands** exposed from the BookViewModel. The optional second parameter of the RelayCommand constructor is a "CanDo" method that returns a Boolean. A button bound to such a command is enabled or disabled depending upon that value.

Change the initialization of the **SaveBooksCommand** and the **EditBookCommand** in **RegisterCommands**.

```
EditBookCommand = new RelayCommand(OnEditBook, CanEditBook);
SaveBooksCommand = new RelayCommand(OnSave, CanSave);
```

Now add these two "CanDo" methods

```
private bool CanEditBook()
{
   return null != SelectedBook &&
          SecurityService.IsAuthenticated();
}

private bool CanSave()
{
  return HasChanges &&

         SecurityService.IsAuthenticated();
}
```

The **SecurityService** doesn't exist yet; we'll create it in the next section. Fake it for now with this:

```
// Will be replaced shortly
private static class SecurityService
{
    public static bool IsAuthenticated() { return true; }
    public static event EventHandler UserChanged;
}
```

CanEditBook depends upon the presence of a SelectedBook. When the SelectedBook changes, the button binding should revalidate enablement. The **SelectedBook** setter must tell the binding to revalidate as shown in the highlighted line:

```
public Book SelectedBook
{
    get { return _selectedBook; }
    set
    {
        _selectedBook = value;
        RaisePropertyChanged("SelectedBook");
        EditBookCommand.RaiseCanExecuteChanged();
    }
}
```

CanSave depends upon whether or not there are pending changes. The HasChanges property needs a similar line.

```
set
{
    _hasChanges = value;
    RaisePropertyChanged("HasChanges");
    SaveBooksCommand.RaiseCanExecuteChanged();
}
```

The original source had just this line in another method, **BookDataServiceNotifyHasChanges. That's the handler that listens for pending changes and happens to be the only place that sets the HasChanges property. We like that handler; we don't think it know about commands.**

Delete **the *RaiseCanExecute* call from *BookDataServiceNotifyHasChanges*.**

Both "CanDo" methods depend upon whether the current user is authenticated or not. Authentication status doesn't change for a given user but the current user changes with each login and logout. The AuthenticationService raises an event when the user changes.

The BookViewModel registers to listen for that change in its constructor; the handler triggers the command bindings:

```
  SecurityService.UserChanged += UserChanged;

  /* ... more code ...*/

private void UserChanged(object sender, EventArgs e)
{
  EditBookCommand.RaiseCanExecuteChanged();
  SaveBooksCommand.RaiseCanExecuteChanged();
}
```

Finally, fix the *BookViewModel* constructor so that ***RegisterCommands*** is called before ***InitializeModels***.

If we ran the application now, the ViewModel would think that the user is always authenticated; the Edit and Submit buttons would always be lit. Time to add a real SecurityService.

Add *SecurityService* Class

ViewModels could learn if the user is authenticated by checking with the singleton AuthenticationManager in the same way that BookView.xaml used to do.

The DevForce AuthenticationManager replaces the RIA Services WebContext

This application is overly liberal with singletons. Singletons create problems for automated tests because they are static and they are (usually) impossible to change. We should use a few of them as possible.

We certainly shouldn't use a singleton for access to the current user and his authorizations. When we test our ViewModels, we will certainly test for behavior that varies with the user's authentication status and roles. If we can't easily change the user, we can't easily test behavior that depends upon the user's security constraints. We need another service instead, a "SecurityService".

The interface is simple:

```
public interface ISecurityService
{
   UserBase User { get; }
   event EventHandler UserChanged;
}
```

We'll want some convenience methods, starting with "***IsAuthenticated***" which reports if the user is authenticated. I think it's a little easier to write and read "`service.IsAuthenticated()`" than "`service.User.IsAuthenticated`".

If you agreed, you might be tempted to extend the interface with **IsAuthenticated**. But that would mean every derived class would have to re-implement that method. We loathe duplication. Use extension methods instead:

```
public static class SecurityServiceExtensions
{
  public static bool IsAuthenticated(this ISecurityService auth) {
     return auth.User.IsAuthenticated;
  }
}
```

Now every ***ISecurityService*** implementation gets ***IsAuthenticated*** for free.

Next we need an implementation. Actually, we need two implementations: SecurityService and DesignSecurityService. Drag them in from the finished solution; both are easy to understand without further comment.

Services are delivered through the ServiceProviders; extend ServiceProviderBase, ServiceProvider and DesignServiceProvider to accommodate the new SecurityServices; see the finished solution.

The SecurityService is "injected" into a BookViewModel through its constructor; add a new ISecurityService parameter and protected property. Delete the inner static SecurityService class we added in the previous section; that was a placeholder until we implemented the real service property.

Finally, update the BookViewModel initialization in the **ViewModelLocator** to pass the SecurityService into the BookViewModel constructor.

Add *BookEditor* Service Class and Clear the BookView Code-behind

The BookView.xaml.cs code-behind has material in it that continues to disturb me.

I am not rigidly opposed to code in the code-behind file. But I'm always both suspicious and curious.

A couple of things bother me about this code-behind.

It creates another view, the EditBookWindow, when the user wants to edit a book. I don't like one view to depend upon another. Changing one view risks having to change the other. I could stomach it in this demo, if it weren't for the second issue.

The ViewModel isn't allowed to launch the EditBookWindow directly. Instead it has to send a message into space (via the Messenger) and trust that the view will hear the message and show the book editor. That's like two people in the same room holding the conversation by phone. It's silly and could be difficult to debug.

It's done this way because the author (correctly) does not want to the BookViewModel to be dependent upon a particular book editor implementation and certainly doesn't want a dependency on the EditBookWindow itself. ViewModels are never allowed to refer to UI elements; that's one of the few ViewModel laws.

The way out of this conundrum should seem familiar by now: add a new service – the BookEditor service – and inject it into the ViewModel. Here's the BookEditor.

```
public interface IBookEditor
{
  void Show();
}

public class BookEditor : IBookEditor
{
  public void Show()
  {
    new EditBookWindow().Show();
  }
}
```

We simply relocated the "OnLaunchEditBook" method of the BookView.xaml.cs to this class.

Delete this method and the SavedBookDialogMessage registration from that code-behind.

Delete the SavedBookDialogMessage class too.

Add the BookEditor service to the three service provider classes as we've done before. We don't need a DesignBookEditor; use null for the DesignServiceProvider.

The BookEditor is "injected" into a BookViewModel through its constructor; add a new IBookEditor parameter and protected property.

Revise the "**OnEditBook**" method

```
private void OnEditBook()
{
  if (null != BookEditor) BookEditor.Show();
}
```

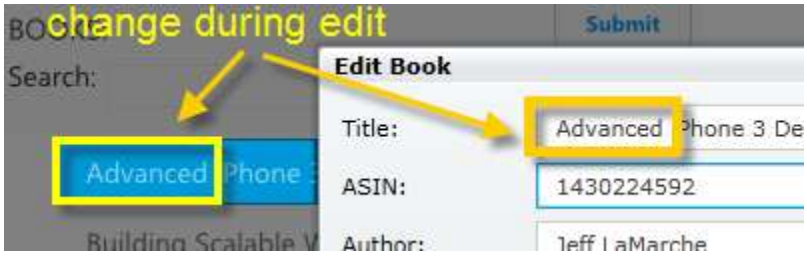
Finally, update the BookViewModel initialization in the **ViewModelLocator** to pass the BookEditor service into the BookViewModel constructor.

Make Cancel work

Try this:

- Login,
- open the book editor
- make a change to the book title.

Notice that the book title in the main BookView changes to match.



The EditBookWindow (the book editor view) and the BookView views share the same BookViewModel instance, the same BookDataService, the same supporting EntityManager, and the same physical Book entity. There is no isolation between the book editor and the list-of-books views.

Many users like seeing the update propagated across the application as they happen.

Some do not. Someday we may be asked to isolate the book edit session from the main view so that changes are reflected in the main view only after the user clicks "OK". We'll have to "sandbox" the editor. But, today, there is no such requirement and propagating changes across views is fine.

Now click the "Cancel" button or the close box ("x"). Notice that the book title retains its changed value. That does ***not*** accord with anyone's notion of cancel. Everyone expects that changes made within the editor will be "undone".

"Undo" could mean many things. In this example, closing the dialog without clicking "OK" means revert the selected book to its original state – the condition it was in when last retrieved or saved.

> Other interpretations include reverting the book to the state just prior to opening the editor or rolling back all unsaved changes to this book and all other books. Multilevel "undo" might be required. We can support all of these interpretations with DevForce at the cost of writing more sophisticated code.

Begin by adding a callback parameter to the "Show" method in the **IBookEditor** interface. When the editor closes, the editor should invoke the callback with `true` if the user approved the changes.

Now implement it.

```
public interface IBookEditor
{
  void Show(Action<bool> onClose);
}

public class BookEditor : IBookEditor
{
  public void Show(Action<bool> onClose)
  {
    var editor = new EditBookWindow();
    editor.Closed += delegate { onClose(editor.DialogResult ?? false); };
    editor.Show();
  }
}
```

The Show method attaches a handler to the EditBookWindow's "*Closed*" event. If the DialogResult is `true`, the user clicked "OK"; else the user clicked "Cancel" (`false`) or the close box (`null`).

The **BookViewModel**'s *OnEditBook* method is revised to this:

```
private void OnEditBook()
{
  if (null != BookEditor)
    BookEditor.Show(ok =>
       {
           if (ok) return;
           SelectedBook.EntityAspect.RejectChanges();
           SelectedBook.EntityAspect.ValidationErrors.Clear();
       });
}
```

If the user clicked "OK", we preserve the users changes by doing nothing.

Otherwise, we use the DevForce technique for reverting the pending changes of a single entity. We ask the SelectedBook for its "`EntityAspect`" – an object that reveals its "entity characteristics" such as the `RejectChanges` method.

> We are fortunate in our example. The book editor can only modify a single entity. If the conceptual book were represented by a complex entity graph we'd have to revert the multiple entities of its

graph. Such complex rollback logic would belong in a method dedicated to the purpose, perhaps in the Book class, perhaps in the BookDataService, definitely not in the ViewModel.

We also clear any ValidationErrors that the user might have introduced while editing.

Delete the PageConductor

The PageConductor is an important design pattern and its implementation and use in the BookShelf would be a notable improvement over the Navigation Framework inherited from the RIA Services Business Application Template.

It is not used in the BookShelf and shows signs of having been borrowed from a different application and then quickly abandoned (note the commented-out "Tweet" references").

We should delete it from this application. It's dead wood. We can always find it again when and if we're prepared to switch to PageConductor navigation.

Deleting the PageConductor affects:

- IPageConductor, PageConductor, DesignPageConductor
- ServiceProviderBase, ServiceProvider, and DesignServiceProvider
- BookViewModel and CheckoutViewModel (which did nothing with it)
- ViewModelLocator (which need not pass it to the ViewModels).

Start ViewModels Manually

Both the BookViewModel and the CheckoutViewModel load data in their constructors.

As a general rule, constructors shouldn't do work; they should do the minimal amount of wiring and throw exceptions only if invoked incorrectly (e.g., with bad parameters). A simple network access problem could cause BookShelf to fail in the constructor.

As a practical matter, constructors shouldn't do work because failures are difficult to debug and give design tools fits. Constructors that do work also interfere with automated testing. When we get to testing the ViewModel, we'll want to tweak the tested instance before it loads data.

Loading data in ViewModel constructors is a legacy from a time when they were created directly in the View xaml. Xaml is a restricted programming environment; you can't create something and call a method on it in xaml – not without ugly trickery. Because designing views in Blend with real ViewModels is a top priority and that meant creating ViewModels in xaml, we sacrificed the "no work in constructors" principle. That sacrifice is worth it if it's necessary. Fortunately, it's not necessary.

Views no longer instantiate their ViewModels directly; they acquire ViewModels from the ViewModelLocator. A ViewModelLocator can call almost anything including constructors with parameters and methods.

The original code has a "Book" property which returns the BookViewModel, not a Book entity. Rename the property to "BookViewModel" to properly reflect its true type.

```
public BookViewModel BookViewModel { get; set; }
```

Correct the DataContext setting in the BookView. xaml to match.

```
DataContext="{Binding BookViewModel, Source={StaticResource Locator}}"
```

John made these same changes in a later version of his code.

Update the ViewModelLocator to call Start() for both ViewModels; here is the BookViewModel case

```
BookViewModel ~~Book~~ = new BookViewModel(
   _sp.BookDataService,
   _sp.SecurityService,
   _sp.BookEditor
   ).Start();
```

Open each ViewModel, rename the LoadData method as **Start**, make it public, and have it return the ViewModel. Here is the BookViewModel Start method:

```
public BookViewModel Start()
{
    LoadBookOfTheDay();
    LoadCategories();
    return this;
}
```

While we're poking around in the ViewModel constructors, we reincorporate the one or two lines of *InitializeModels*, (which initialized properties, not models) and delete the null entity initialization of the selected item (SelectedBook, SelectedCheckout) properties which served no useful purpose.

Confirm that the application continues to work and that the views still Blend.

Protect the ViewModel Setters

The API of ViewModels – like any class –should be buttoned down and as skinny as possible. Too many properties are public for my tastes.

No caller should be able to set the collection properties: **Books**, **Categories**, **Checkouts**. Their contents change but they are never replaced. They should be private. In fact, they don't need to raise property changed either because they are always instantiated in the constructor. We can make them automatic properties with private setters.

Other methods and properties should be private:

- setters of the RelayCommands
- BookOfTheDay (only set by the ViewModel)
- all Load… methods

We might be interesting to poking a few properties during testing. A common test approach is to create a subclass of the ViewModel, instrument with probes and test the behavior of this derived class.

When this is our purpose, we can make some properties protected (or at least make their setters protected). The service properties such as BookDataService and SecurityService were protected for this reason. Even this step should be taken cautiously and only where actually needed.

Protected members were changed to private in BookShelfDF without loss of functionality or testability.

> Some test gurus look askance at this approach, in part because it implies that the class could be sub-classed in production when, in fact, we only subclass it for testing. The convenience of this technique trumps the objection in our opinion.

If there were an IBookViewModel (and there won't be) it would look like this:

```
public interface IBookViewModel
{
    RelayCommand ClearFilterCommand { get; }
    RelayCommand SearchBooksCommand { get; }
    RelayCommand LoadMoreBooksCommand { get; }
    RelayCommand SelectCategoryCommand { get; }
    RelayCommand EditBookCommand { get; }
    RelayCommand SaveBooksCommand { get; }
    ObservableCollection<Book> Books { get; }
    ObservableCollection<Category> Categories { get; }
    bool HasChanges { get; }
    BookOfDay BookOfTheDay { get; }
    Category SelectedCategory { get; set; }
    Book SelectedBook { get; set; }
    string TitleFilter { get; set; }
    bool IsBusy { get; }
    BookViewModel Start();
    event PropertyChangedEventHandler PropertyChanged;
}
```

Cache the Navigation Pages

You navigate among the pages of this application by clicking a page link in the band across the main page view (the "shell"). "Books" and "Checkouts" are the two links in the Bookshelf application.

As shipped, each click results in a new page instance. The click sequence "Books-Checkouts-Books-Checkouts" results in the creation of two BookViews and two CheckoutViews.

You won't notice with the BookView because each new BookView instance binds to the same BookViewModel (see the ViewModelLocator). The BookViewModel transfers the display state of the first BookView instance to the new, second BookView instance.

The CheckoutView reveals more clearly what is going on. Try this:

- Click “Checkouts”.

  After a pause to retrieve Checkouts, the view displays the first book, “Programming Entity Framework”.

- Select the third book, “Essential Silverlight 3”
- Click “Books” to switch to Books view
- Click “Checkouts” again.

The Checkout page state is not as you left it. Instead of displaying “Essential Silverlight 3” it shows the first book, “Programming Entity Framework”. The Checkouts page was built entirely from scratch. You see the ***initial*** page state each time you navigate to it.

That’s because each new CheckoutView binds to a new CheckoutViewModel every time (see the ViewModelLocator). Were it not for DevForce caching, you would have seen the pause as the application re-retrieved the Checkout data.

When users return to a page, they expect it to be in the same state as they left it and they don’t expect to have to wait for data. If they want a refresh, they’ll ask for it.

You might revise the ViewModelLocator to conform to preserve the CheckoutViewModel across navigations, just as it does the BookViewModel.

That might not be the best solution.

The decision to create a single global ViewModel or a new ViewModel each time has no obvious relationship to user navigation. It’s a hack to fool the navigation system.

Some view state might be held in the view instead of the ViewModel (e.g., whether a portion of the screen is expanded or collapsed). That state would be lost when the view is recreated.

Views are notorious for memory leaks. Every time you create a new view, you risk creating an inadvertent memory leak. For example, views typically attach event handlers to ViewModel lists and to individual entities. Although the first instance of CheckoutView is no longer visible it could (and probably does) have lingering event handlers attached to Checkout entities that are still alive. Those event handlers will prevent garbage collection of the CheckoutView. The BooksView is in the same boat.

In my experience, it is safer to re-use views – especially views that are conceptually permanent as these two are - rather than recreate them.

The navigation framework has a little view caching feature. We can ensure that a previously displayed view is restored from the navigation cache by adding a single line to the constructors in the view’s code-behind.

```
public CheckOutView()
{
```

```
    InitializeComponent();
    Title = ApplicationStrings.CheckoutPageTitle;
    NavigationCacheMode = NavigationCacheMode.Required; // always cache this view
}
```

The same line was added to the BookView constructor.

Add NoVM – the "No ViewModel" Page

The Bookshelf application is a terrific code base for presentations on the MVVM pattern. Such presentations often benefit from an example of a view that doesn't use MVVM. The shipped BookshelfDF sample sports such a page called "NoVM "which consists of a DataGrid displaying books.

This walkthrough assumes that you don't want this page and, therefore, does not describe how to add it. If you too are pontificating on MVVM and find the page useful, please look directly at the code, these files in particular:

BookShelf/MainPage.xaml.cs	Add the page to the routes
BookShelf/Views/NoVM.xaml	The xaml for the "No ViewModel" view
BookShelf/Views/NoVM.xaml.cs	The view code-behind that exercises DevForce to query Books

Silverlight Automated Testing

Easier automated testing is one of the benefits of the Model-View-ViewModel (MVVM) pattern. This section is for you if you care about automated testing. Otherwise, drop the BookShelf.Test project from the finished DesktopDF solution and carry on.

This section is not a tutorial on automated testing. Brian Noyes wrote a superb post on automated testing of RIA Services applications that refers to useful resources.

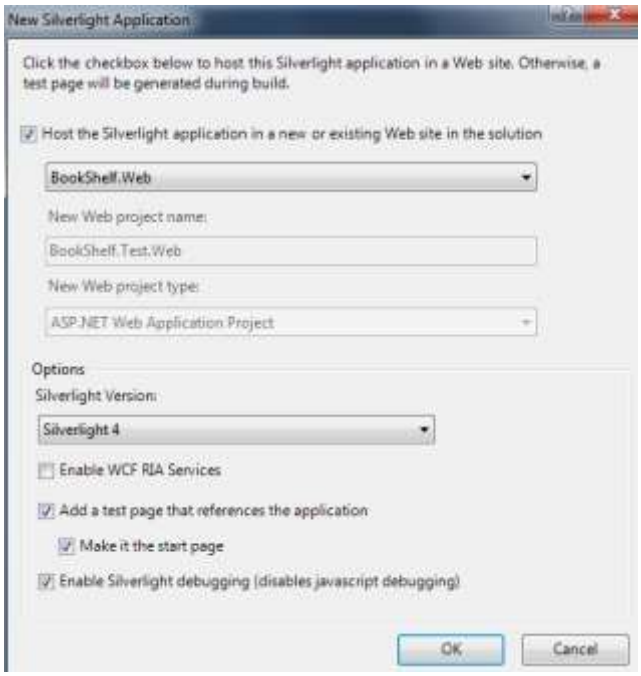
Brian describes RIA Services as "***test-resistant***" code, an understatement in light of the gymnastics required to test even simple things. DevForce is much more test-friendly as we'll see in this section.

Setup the Test Project

Add a new "Silverlight Unit Test Application" project to the solution; call it "BookShelf.Test".

> BookShelfDF relies on Microsoft's Silverlight Unit Test Framework which is distributed separately as part of the "Silverlight Toolkit". The "Silverlight Unit Test Application" template is installed with the Silverlight Toolkit.
>
> BookShelfDF includes the test framework libraries in the test project's "Libs" folder; we strongly recommend that you install the full Silverlight Toolkit anyway.

Choose the following settings in the template wizard:



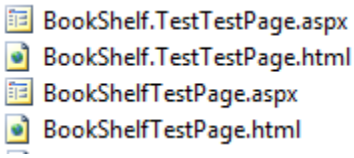
The template wizard generates a new "BookShelf.Test" project.

Rename the Web Host Pages

The template wizard also added web pages to host the tests ("BookShelf.TestTestPage.*").

The Silverlight automated test is actually a Silverlight application that runs in the browser; it needs its own web host page.

The web project now has four Silverlight host pages with similar, confusing names.



We don't need both ".html" and ".aspx" pages; either one will do. Arbitrarily delete the aspx versions.

Rename the BookShelf ***application*** web page, "BookShelfTestPage.html", to "default.html", a well-known name that the web project will start by default in the absence of a designated Start Page.

Rename the BookShelf test web page, "BookShelfTestPage.html", to "BookShelfTest.html"

Select that test page, right-mouse-click, and from the context menu pick "Set As Start Page". Press [F5] and Visual Studio should run the one "do-nothing" test.

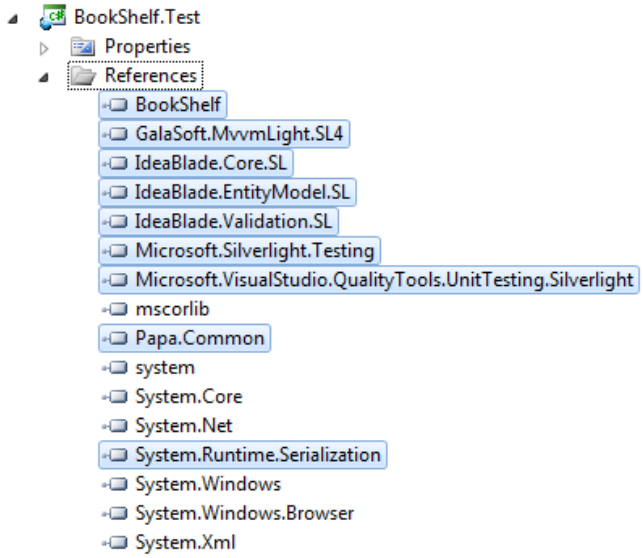
To re-establish the application page, select "default.html" and pick "Set As Start Page".

> You can launch either page directly with right-mouse-click | "View in Browser" … if that option is available in the context menu, which it is unless the Visual Studio session is debugging at that moment.

Add References

We'll test the Silverlight application project, "BookShelf"; we'll need access to the auxiliary project, "Papa.Common". We'll need references to some DevForce libraries, an MVVMLight library, and a couple of Silverlight assemblies too.

The added references are selected in the following screen shot:



Test the Model

The first tests are simple tests of the model entities.

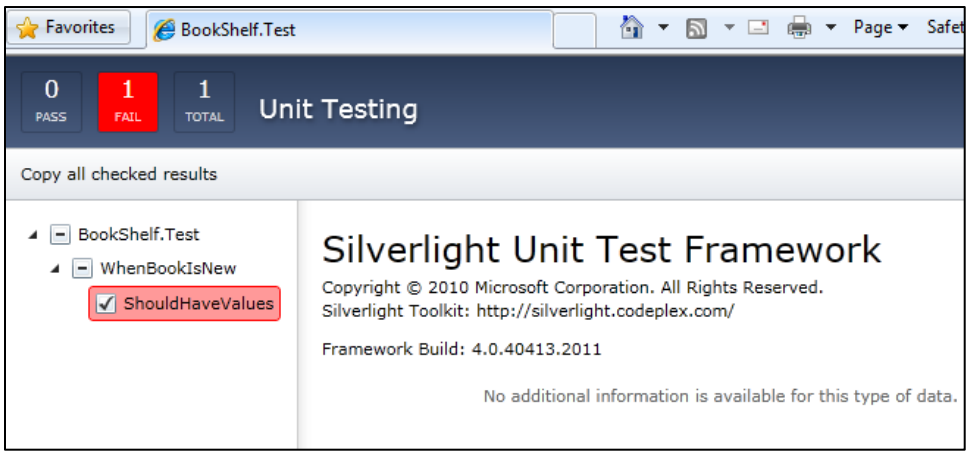
We'll step tediously through the first example and pick up the pace thereafter.

- Rename the template-generated "Tests.cs" file to "ModelTests.cs"
- Rename the test class to "WhenBookIsNew"
- Rename TestMethod1" to "ShouldHaveValues"
- Fill in the method as follows:

```
[TestMethod]
public void ShouldHaveValues()
{
    var newBook = new Book
                      {
                          BookID = 42,
                          Author = "John Papa",
                          Title = "Data-Driven Services",
                      };
    Assert.AreNotEqual(42, newBook.BookID);
}
```

The assertion will fail because the BookID should be "42". [F5] to Run it … and see it fail.

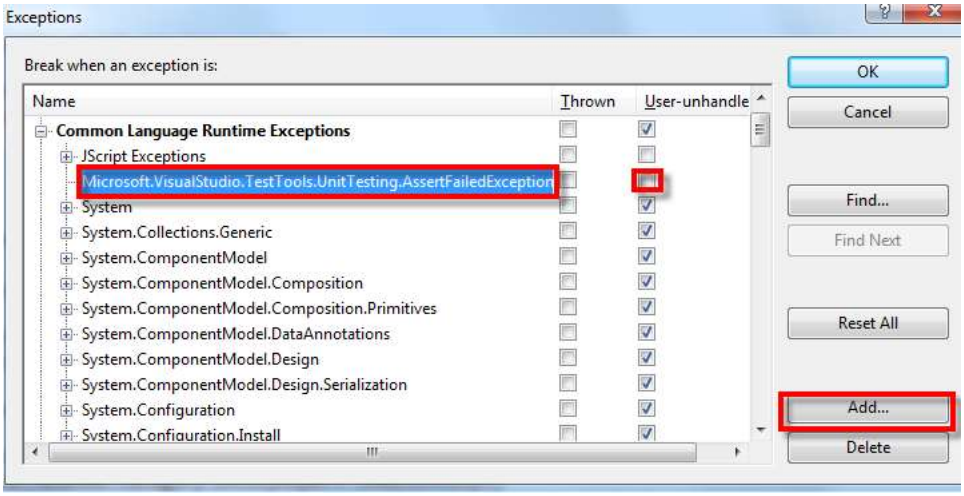
The debugger breaks at the assertion and highlights the broken line in green. The debugger caught the exception. You can press “Continue”. The test harness finishes and reports the failure.



Tell Visuals Studio Debugger to skip SilverlightTest Exceptions:

Halting for assertion exceptions is annoying. You can configure Visual Studio debugger to keep going.

- Open the menu “Debug | Exceptions … “ (Ctrl+Alt+E)
- Open the “Common Language Runtime Exceptions” node
- Click the “Add” button
- Select “Common Language Runtime Exceptions” as the type
- Enter the name of the testing exception
  “Microsoft.VisualStudio.TestTools.UnitTesting.AssertFailedException”
- Uncheck the “User unhandled” checkbox next to the newly added exception name



Press “OK” to close the dialog. Run the test harness again. It displays the test failure without interruption.

Fix the test – change the Assert to “AreEqual” – and run it again to see it pass.

Verify that new entities are detached

DevForce entities can't do much until they are attached to an EntityManager (**`BookShelfEntities`** in our application).

Try these tests of detached entities:

```
[TestMethod]
public void ShouldBeDetached()
{
    var newBook = new Book();
    Assert.AreEqual(EntityState.Detached, newBook.EntityAspect.EntityState);
}

[TestMethod]
public void ShouldReturnNullCategoryWhileDetached()
{
    var newBook = new Book();
    Assert.IsNull(newBook.Category);
}

[TestMethod]
public void ShouldNotHaveErrorsWhenSetNullTitle()
{
    var newBook = new Book();
    newBook.Title = null; // Title is required
    Assert.IsFalse(((INotifyDataErrorInfo) newBook).HasErrors);
}
```

Note that navigation properties such as *Category* return null when an entity is detached. Navigation properties of an attached entity always return an entity object of some kind as we'll see in the next series of tests.

New entities like these are "detached" until they are added to the manager explicitly. The entities returned by a DevForce query are attached to the EntityManager automatically.

Add entities to a Disconnected EntityManager

Examine the **`WhenBookIsAddedToManager`** test class in "ModelTests.cs"

The **`TestInitialize`** method, which runs before every test method, is nearly the same as the one in the first test class:

```
[TestInitialize]
public void TestInitialize()
{
    Context = new BookClubEntities(false /* disconnected */);
    NewBook = new Book
    {
        BookID = 42,
        Author = "John Papa",
        Title = "Data-Driven Services",
```

```
    };

    Context.AddEntity(NewBook);
}
```

It differs in two critical respects:

- A "Context" is defined as an instance of the BookShelf's own EntityManager type.
- The new book is added to that "context"

> Most DevForce developers use the word, "Manager", to denote the EntityManager instance that they're using. Because you may be coming from a RIA Services world, we use the word, "Context", to make you feel more at home.

Very important: the **`BookClubEntities`** context is created with the constructor overload that takes a Boolean parameter that tells EntityManager whether to connect to the server automatically. The value is false in this example meaning that the EntityManager should stay offline.

We rarely use a connected EntityManager in tests. We prefer tests that do not depend upon access to a server or to the database; such tests are faster and invulnerable to network or server outages.

> We absolutely never use a connected EntityManager when laying out views with visual design tools such as Blend. See the DesignEntityManagerProvider; it creates a disconnected EntityManager too.

When disconnected, the EntityManager confines query requests to the contents of its entity cache. Queries that would otherwise travel over the network and select from the database will select from the cache instead. That cache acts, in effect, as an in-memory representation of the database. It's a great place for test entities that we quickly create and destroy with impunity.

This disconnected EntityManager and its queryable cache are major contributors to easy application testing.

In the next test series we see how adding an entity to the context changes its behavior.

First, the state of the entity changes to "Added".

```
[TestMethod]
public void ShouldBeInAddedState()
{
    Assert.AreEqual(EntityState.Added, NewBook.EntityAspect.EntityState);
}
```

We haven't assigned the book to a category yet. The Category navigation property still returns "nothing" but it's a special kind of "nothing", a "NullEntity".

```
[TestMethod]
public void ShouldHaveNullEntityCategory()
{
    Assert.IsNotNull(NewBook.Category);
```

```
    Assert.IsTrue(NewBook.Category.EntityAspect.IsNullEntity);
}
```

DevForce navigation properties are examples of the "Null Object" pattern (aka, the "Nullo Pattern") . Rather than return null and risk a NullReferenceException, an attached DevForce entity returns a "null entity", a stand-in that behaves like a real entity. You can always ask an entity if it is real or the "NullEntity" as we saw in that test method.

The next test demonstrates that we can navigate to a real entity by adding one to the cache and associating it with the new book.

```
[TestMethod]
public void ShouldHaveAddedCategory()
{
    var category = new Category
    {
        CategoryID = 11,
        CategoryName = "Volume",
    };
    // Assigning the category pulls it into NewBook's EntityManager
    NewBook.Category = category;

    Assert.IsNotNull(NewBook.Category); // has a Category object
    Assert.IsFalse(NewBook.Category.EntityAspect.IsNullEntity); // not the nullo
    Assert.AreEqual(EntityState.Added,
                    NewBook.Category.EntityAspect.EntityState); // it too is added
}
```

Test the BookViewModel

View logic testing is supposed to be "easy" with MVVM because most of that logic is in the ViewModels which are unencumbered by visual components.

On the other hand, a ViewModel typically has a large API and exposes complex objects (such as entity graphs) that are difficult to mock or fake.

Worse, a Silverlight ViewModel retrieves and saves model elements asynchronously whether directly (bad idea) or with the help of a repository/"data service". Asynchronous method testing in most test frameworks requires specialized techniques and ceremony that confound many developers.

You might think that the IBookDataService, which "wraps" the persistence framework, liberates ViewModel testing from these challenges. You just mock the interface, right? Here's Brian Noyes again:

> *While that is actually fairly easy to do, it doesn't actually make the problem much easier. The real challenge about unit testing code … has to do with both the async API exposed, and the state management … behind the scenes maintaining the collections of entities and the change tracking on those entities. Also things like associating errors in the service calls with the entities for validation purpose, etc. Bottom line, … if you try to mock out its public API, your mocking code will have to do a ton of work to simulate the same behavior. … you won't really be testing*

> *the functionality of your view model anymore, you will just end up testing that your mock is doing what you told it to, which is pointless.*

In other words, pure unit tests won't work. Most of your meaningful tests will have to integrate with the model framework. The challenge is to minimize the noise, complexity, and brittleness of that integration. Above all, you should avoid connecting to the server or the database.

> Unless you want to connect. Automated testing includes deep integration tests that reach to the server and, beyond the server, to the database. Such tests can require elaborate setup, they run slowly, and they are prone to network, server and data failures far removed from the focus of testing. We have to write and run such tests but they shouldn't be the staple diet of our testing regime.
>
> We don't show deep integration testing in this demonstration. Even here, DevForce can make the job easier with its custom CompositionContext and n-tier FakeBackingStore.

Fortunately, the DevForce offline EntityManager with queryable cache makes a huge difference. As we'll see, you can test the entire BookViewModel with standard synchronous techniques familiar to test authors everywhere.

The BookViewModel Test Itinerary

"**BookViewModelSyncTests**", located in the BookShelf.Test project, demonstrates a completely synchronous approach to testing the BookViewModel.

It is supported by three helper classes

TestBookEditor	A substitute for the BookEditor that would present a ChildWindow
TestDataMother	Fills a test EntityManager with generated test entities
TestSecurityServices	BookViewModel behavior is sensitive to the security characteristics of the current user. Tests use this class to change that user.

We'll visit all four files, pointing out a few of the landmarks. We won't cover every detail; you are perfectly capable of doing that yourself.

BookViewModelSyncTests

`BookViewModelSyncTests` is an ordinary test class. It doesn't derive from anything … which makes it extraordinary for a Silverlight test class with calls to asynchronous methods.

A "normal" Silverlight Unit Test Framework test class would inherit from the framework's **`SilverlightTest`** class. It's test methods should carry the **`[Asynchronous]`** attribute and be littered with calls to **`EnqueueCallback`**.

The `BookViewModelSyncTests` class has none of these trappings. Yet it invokes the asynchronous calls of its BookDataService. That is only possible because – with a throw of a switch – the DevForce EntityManager stops trying to connect with the server, relies exclusively on entities in cache, and behaves synchronously. It executes entirely on the test thread and invokes the callback delegates when

done. The entire business is single-threaded. There is no need – and no reason – to make the test harness wait for an asynchronous process to complete.

> Save is the exception. At present there is no easy way to fully test the SaveChangesAsync() in synchronous fashion. We address this shortcoming below.

The ViewModel code hasn't changed to accommodate the DevForce framework. Neither has the BookDataService API. They are asynchronous performers in production. They simply behave synchronously under test.

You don't create test versions of DevForce framework classes either. There's no fake DomainClient as described by Brian Noyes and Nikhil Kothari

The "Test Initialization" region shows what you do to achieve this effect.

- create the test EntityManager – an instance of the BookClubEntities – in the offline mode.
- intercept attempts to contact the server by handling certain events (such as the Saving event)
- populate the entity cache with test data that appear to have been retrieved from the database.
- supply test versions of BookViewModel dependencies to its constructor

Setup should be minimal. The "Test Initialization" region is about 30 lines (not counting the TestDataMother) which hardly seems minimal. But it's amortized over roughly 30 tests and would handily support more if we wrote all of the tests we ought to have.

Let's jump ahead to see what a few test methods look like, then come back the setup.

A Few Simple Tests

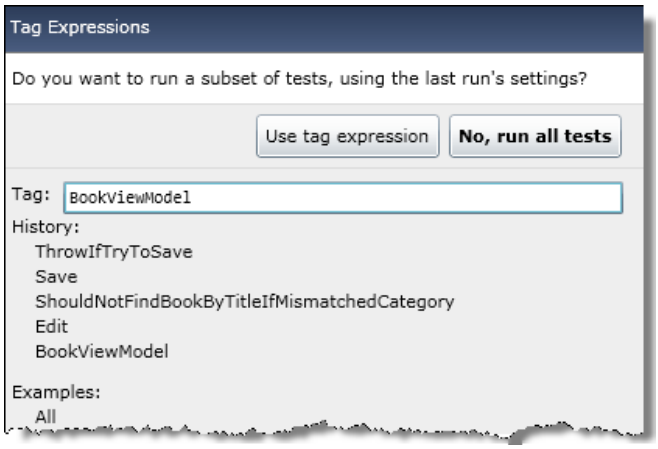
First, a negative test. The BookViewModel doesn't load automatically anymore. We have to call Start first. The ViewModel 's "Books" collection should be empty before Start.

```
[TestMethod]
[Tag("BookViewModel")]
public void ShouldNotHaveBooksBeforeStart()
{
    Assert.IsFalse(ViewModel.Books.Any(), "Has books");
}
```

[TestMethod] is the necessary and standard "MS Test" annotation for any test method. We'll get to the [Tag (…)] attribute shortly.

The test asserts that the ViewModel.Books doesn't have any books. We highly recommend including the "optional" failure message ("Has books"), especially with True/False asserts which are incomprehensible when they fail without a message.

The [Tag (…)] attribute is an instruction to the Silverlight Unit Test Framework. The test application web page invites you to enter and use a tag expression to limit the run to matching test methods. In this illustration, the harness will run tests with the tag, "BookViewModel"



Start() loads the ViewModel with entities so we should have some books after calling Start:

```
[TestMethod]
[Tag("BookViewModel")]
public void ShouldHaveBooksAfterStart()
{
    ViewModel.Start();
    Assert.IsTrue(ViewModel.Books.Any(), "No books");
}
```

This is state-based testing. We don't know for sure that the BookDataService was called via Start. We only know that there weren't any books before and now there are books. Almost all tests in BookShelfDF are state-based.

Most tests in this class are short. Not always as short as these two. But they are short by design. They convey one or a few simple facts about the ViewModel and its behavior. We'd rather write several short tests than one large one.

Our final introductory test demonstrates the ViewModel's response when the View triggers the "SearchBooks" command.

```
[TestMethod]
[Tag("BookViewModel"), Tag("Command")]
public void ShouldFindBookByTitle()
{
    ViewModel.SelectedBook = null;
    ViewModel.TitleFilter = TestEntities.TestBook.Title;
    ViewModel.SelectedCategory = TestEntities.TestBook.Category;

    ViewModel.SearchBooksCommand.Execute(null);

    Assert.AreSame(
        ViewModel.SelectedBook,
```

```
        TestEntities.TestBook,
        "SelectedBook is not the 'TestBook' after search");
}
```

The test clears the SelectedBook because it will later check that the SelectedBook is the one searched for.

We know by inspection that the BookViewModel needs both the search text (TitleFilter) and a Category to look for a book; those are parameters to the BookDataService's GetBooksByTitle method. There are companion tests to verify what happens when values are empty or "bad".

The line in this test that executes the SearchBooksCommand is most interesting:

```
ViewModel.SearchBooksCommand.Execute(null);
```

The BookView binds a button to the ViewModel's SearchBooksCommand. The ViewModel doesn't know or care about the button. The impetus could come from a hyperlink or any visual cue. What matters is that something in the UI calls "Execute" on the command object.

One beauty of the MVVM pattern is that we can test ViewModel inputs, outputs, and behaviors without engaging the UI apparatus. That apparatus is irrelevant.

Finally, the assert checks the state of the SelectedBook. We take on faith that the assert succeeds only because the BookDataService was called and it returned the expected book.

Interaction Testing

We keep inferring how things happen from observable changes in the ViewModel. Such inferences are sufficient for most of us most of the time.

We'd need an interaction test to verify that the ViewModel actually searched for a book by calling the BookDataService.GetBooksByTitle method which in turn called EntityManager.ExecuteAsync.

The BookDataService implements IBookDataService. A mocking tool such as MOQ is perfect for verifying that leg of the interaction. Do we really need to know about the second leg of the journey?

The DevForce EntityManager is not mockable. There is no IEntityManager and EntityManager itself has few virtual methods. But it does raise events for almost everything asked of it. Here's an interaction test using the Querying event to ascertain (a) that a query was issued and (b) for the expected entity type.

```
[TestMethod]
[Tag("BookViewModel"), Tag("Query")]
public void ShouldIssueBookQueryWhenFindBookByTitle()
{
    IEntityQuery searchBooksCommandQuery = null;

    // Listen for query request from ViewModel by way of BookDataService
    Context.Querying += (s, e) => searchBooksCommandQuery = e.Query;

    // VM queries only if both Category and Filter are specified.
    ViewModel.SelectedCategory = TestEntities.TestBook.Category;
```

```
    ViewModel.TitleFilter = TestEntities.TestBook.Title;

    ViewModel.SearchBooksCommand.Execute(null);

    Assert.IsNotNull(
        searchBooksCommandQuery, "Context was not queried.");
    Assert.AreSame(
        typeof(Book), searchBooksCommandQuery.ElementType,
        "Context was not queried for a Book");
}
```

Setup for BookViewModelSyncTest

Many ViewModels have large APIs and they tend to depend upon several other classes, chief among them a repository/"data service" that itself is complex and, in practice, un-mockable.

BookViewModel is such a class. It's constructor takes three dependent services, one of the BookDataService which depends upon a DevForce EntityManager.

Proper test coverage will exercise the full ViewModel API and engage all of its dependencies. It takes more than a few lines of code to arrange the context for all of the tests. The best we can do is:

- minimize the amount of setup
- make each line short and clear
- amortize the setup cost over many test methods

Here's what we've got:

```
[TestInitialize]
public void TestInitialize()
{
    CreateContext();
    AttachTestEntities();
    CreateViewModel();
}
```

The [TestInitialize] attribute tells the test harness to run this method before running each test method.

> **Warning**: the Silverlight Unit Test Framework reuses the same test class instance from test to test. Class fields and properties retain their values from test to test opening the way for cross-test contamination. Remember always to reset fields and properties in a TestInitializer such as this one.

Initialization has three phases:

1. Create an offline EntityManager (aka "Context")
2. Populate it data used by the tests
3. Create the ViewModel (aka "system-under-test" or "SUT")

First, we create an offline EntityManager in order to (a) eliminate calls to the server or database and (b) ensure queries are synchronous even though call structures are asynchronous. Notice that each test method starts with a fresh, empty EntityManager.

```
private void CreateContext()
{
    Context = new BookClubEntities(false /* offline mode */);
}
```

Second, we create test entities and attach them to the EntityManager.

```
private void AttachTestEntities()
{
    TestEntities = new TestDataMother(Context);
    TestEntities.AttachAllTestEntities();
}
```

We delegate entity creation to a helper class called a “data mother”. The data mother creates the entities on demand for the provided EntityManager. The “demand”, in this example, is to create and attach all of possible test entities. We’ll look at TestDataMother itself shortly.

The word “Attach”, in DevForce parlance, means to populate the manager’s cache with unmodified entities that appear to have been retrieved from the database. In contrast, to “Add” entities means to populate the manager’s cache with entities that appear to be new.

We recommend initializing the manager’s cache with attached entities so that ViewModel queries return the unmodified entities expected from an actual query that pulled from the database.

Individual test methods can modify these entities or add their own test-specific entities as required.

Third, we create the ViewModel for the test. We create a new ViewModel before each time; nothing we do to a ViewModel in one test can affect the state of a ViewModel in another test.

```
private void CreateViewModel()
{
    ViewModel = new BookViewModel(
        BookDataService = new BookDataService(Context),
        SecurityService = new TestSecurityService(),
        BookEditor = new TestBookEditor()
        );

    // Fail on unintended save attempt
    BookDataService.AllternateSaveFunction =
        delegate { Assert.Fail("Expected save attempted"); };
}
```

A BookViewModel takes three parameters. These are the dependencies “injected” at runtime with the application versions. In the test class we supply test versions of these classes.

The BookDataService is a real data service but its dependency is the test EntityManager we just created. In so doing, this instance of the BookDataService is transformed into a "fake" – a version that performs all of the functions of a real service in a manner suitable for testing.

> Martin Fowler, in his post "Mock Aren't Stubs", described **fakes** as objects that "*actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in memory database is a good example)*."

The other two dependencies are pure test fakes (TestSecuri tyService, TestBookEditor).

The last line ensures that an inadvertent save during testing results in a test failure.

```
    // Fail on unintended save attempt
    BookDataService.AlternateSaveFunction =
        delegate { Assert.Fail("Unexpected save attempted"); };
```

The "**AlternateSaveFunction**" is a twist to the BookDataService implementation. We'll review it more thoroughly when we get to the "Save" tests.

TestDataMother

Each test begins with a new BookDataService backed by an empty offline EntityManager. If the ViewModel asks the data service for books, it will receive an empty list (one of the tests verifies that fact). The data service can only return books that are in the EntityManager's cache.

Each test method could add or attach its own test entities. In practice, these test entities tend to be the same across many tests. The entities themselves aren't interesting and the code to construct them obscures the test's purpose. Accordingly, we delegate routine test entity creation to a helper class that is widely and lovingly referred to as a "data mother".

A large application might have several "data mothers" for different scenarios; the BookShelf needs only one. It consists of several "Attach" methods that make collections of the BookShelf Model types and attach them to the DataMother's EntityManager (presumably the EntityManager instance used by the BookDataService) . Some of the entities are singled out as "well known" instances: `TestCategory`, `TestCategory2` , `TestBook`, `AnotherTestBook`.

The TestDataMother is reminiscent of the DesignEntityManagerProvider class in the Silverlight Application project. The resemblance is no coincidence. We need a source of data to layout the Views with visual design tools and those data must be created locally; we can't ask the server for data while in the visual design tools. Same problem, same solution

Technically, we could have used the DesignEntityManagerProvider as the DataMother in our test project. We did not do so because we fear that changes made for design purposes might break our tests. Changes to test data could make a mess of our designs efforts too. Best to keep them separate.

Faking the Save

The ViewModel "Save" method resolves eventually to the DevForce EntityManager.SaveChangesAsync method. Unlike the query, there is no offline save in DevForce and there's no easy way at the moment to fake the save synchronously.

> The DevForce CompositionContext and FakeBackingStore facilitate distributed tests that don't touch the database. They are terrific for non-destructive end-to-end testing and for developing "model-first" in the absence of any database at all. But this route still involves asynchronous connection to a server.
>
> In BookShelfDF we are concentrating on writing tests that can be run synchronously and don't require a server connection; such tests cannot use the FakeBackingStore.

We could mock the IBookDataService "Save" method or create a fake implementation of that interface. Both approaches would entail considerable effort because the ViewModel interleaves queries, state management and saving.

Instead we made a small change to the Save implementation in the real BookDataService to facilitate testing while preserving its integrity as a production component.

Here's what it used to look like:

```
public void Save(Action onSuccess, Action<Exception> onFail)
{
    Context.SaveChangesAsync(op => SaveCallback(op, onSuccess, onFail));
}

private static void SaveCallback(
    EntitySaveOperation op, Action onSuccess, Action<Exception> onFail)
{
    // Same as shown earlier in this document

}
```

Here it is after revision:

```
public void Save(Action onSuccess, Action<Exception> onFail)
{
    (AlternateSaveFunction ?? DefaultSave)(Context, onSuccess, onFail);
}

public Action<EntityManager, Action, Action<Exception>> AlternateSaveFunction { get; set; }

private static void DefaultSave(
   EntityManager manager, Action onSuccess, Action<Exception> onFail)
{
    manager.SaveChangesAsync(op => SaveCallback(op, onSuccess, onFail));
}
```

"Save" no longer has a fixed implementation; it invokes a function instead, either the value returned from the **AlternateSaveFunction** property or, if that property is null as it is in production, the **DefaultSave** method that implements the original **EntityManager.SaveChangesAsync** logic.

Tests can set the AlternateSaveFunction as they please. This test class initializes it to a function that asserts a failure; test methods shouldn't save by accident.

The battery of save tests reset the function to suit their needs as in this example:

```
public void ShouldSave()
{
    TestEntities.TestBook.Title = "The Papa-san Way"; // make a change

    var wasSaveCalled = false;
    BookDataService.AlternateSaveFunction = (c, s, f) => wasSaveCalled = true;

    ViewModel.SaveBooksCommand.Execute(null); // as when user clicks "Submit"

    Assert.IsTrue(
        wasSaveCalled,
        "Didn't try to save after SaveCommand executed");
}
```

The injected function doesn't save a thing. It doesn't bother with the ViewModel's callbacks. It simply captures the fact that the ViewModel called the BookDataService's Save method when the user clicked the submit button.

Please see the code itself for more elaborate examples.

STILL TO COME

- TestSecurityService
- TestBookEdit

Design-time Entities from a Resource File

One way DevForce supports offline scenarios is by making it easy to extract, serialize, and store an EntityManager's entity cache in a local file. That's a great way to keep entities – even changed entities – available to an application that must operate offline.

Imagine walking into a customer's office, prepared to review contracts, previous orders, pending orders, and everything necessary to secure the next big deal. You won't be plugging into the customer's network. You may be unable to establish a WiFi or mobile internet connection. But your DevForce-based Silverlight application continues to run with all of the data it needs because you stored a snapshot in isolated storage. You can take the new order and save that to isolated storage as well. You later save the new order to the server when you've re-established your connection.

We can use a similar approach using this same technology to support View development with visual design tools such as Blend.

You'll produce better Views more quickly when you design them in Blend with representative data. Unfortunately, you cannot obtain representative data directly from the database because you cannot make a network connection (even within your development box) within the visual design tools. That's why the BookShelf uses "Design Models" that create entities in code, on the fly, to populate the repository … and ultimately the ViewModel … when you're working in Blend.

Writing a few design-entity creation classes as we do in the "DesignModels" folder is easy and it's a highly recommended approach, especially in the early going before you have a database.

But, as your model grows, you may decide that the number of design creation classes has gotten out of hand. They've become yet another burden of classes to maintain. Meanwhile, you have a development database stocked with records that you display and manipulate during dry runs and smoke tests. Can they double as design entities? Yes they can. Follow these two steps:

1. Create a "data loader" that reads database entities and emits a file of serialized entities
2. Run the loader to create the file
3. When working with views inside visual design tools, read and reconstitute this file as design time entities and show them on screen.

Create the Data Loader

The "Data Loader" is a **console application** that loads selected entities into an EntityManager from the database using a synchronous, 2-tier DevForce client and then saves the manager's cache (an **EntityCacheState** or "**ECS**") to a local file in the project.

The downloaded example has completed project called "BookShelf.DataLoader". The ECS file (about about 25KB) is named "DesignCacheState.dat". It belongs to the project but has a "Build Action" of "None" which means it is not included in the assembly.

The primary steps to building it were:

1. Add a full .NET 4 Console Application project called “BookShelf.DataLoader”.
2. Add to the project a new text file called “DesignCacheState.txt”
3. Rename its extension to “.dat”
4. Open its property window and set its “Build Action” to “None”
5. Add references
6. Create an “app.config” file to tell DevForce how to find the database
7. Add a DataLoader class to do the work
8. Call the DataLoader in the **Main** routine of the Program.cs

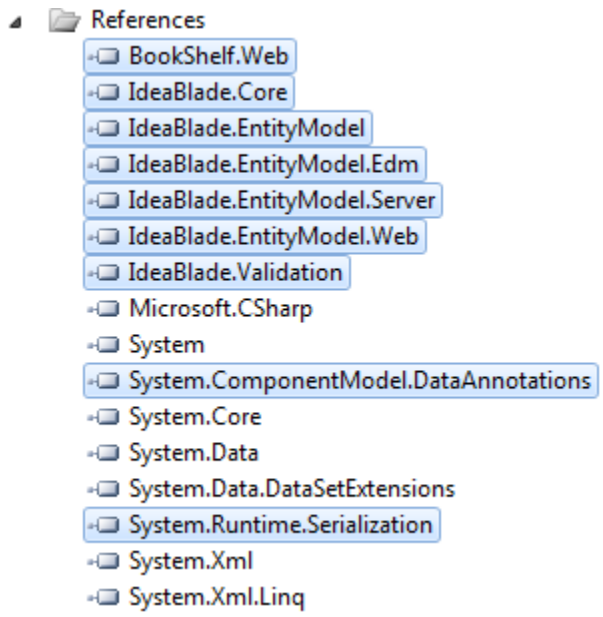
Steps #5, #6, and #7 invite commentary.

References

The “data loader” needs references to

- The domain model which is defined in the web project in this example
- IdeaBlade DevForce libraries
- Two supporting .NET libraries.

The added references are highlighted as shown:



Remember to set the IdeaBlade library property “Specific Version” to “False” as we have elsewhere.

App.config

The console application runs as a 2-tier, “client/server” application which means the EntityManager in the console application (the client) will make direct contact with the database (the server).

This DevForce client has to know the Entity Framework connection string so that Entity Framework can find its schema files and connect with the database. DevForce looks for this information in the console application's "app.config" file.

Only the <connectionStrings> element is needed and this we copy from the BookShelf.Web project's "Web.config" file.

The shipped BookShelf package is configured to access the BookShelf database, located in the BookShelf.Web project's "App_Data" directory, using SQL Server Express. Therefore, the app.config looks like this (after some reformatting and chopping):

```
<?xml version="1.0"?>
<configuration>
  <connectionStrings>
    <!-- SQL Server Connection String -->
    <!--<add name="BookClubEntities" connectionString="metadata=res://*/Models.BookClubMo

    <!-- SQL Server EXPRESS Connection String from BookShelf.Web/Web.config -->
    <add name="BookClubEntities"
         connectionString="metadata=res://*/Models.BookClubModel.csdl| ...
         provider connection string=&quot;Data Source=.\SQLEXPRESS;
         AttachDbFilename=|DataDirectory|\BookClub.mdf;
         Integrated Security=True;User Instance=True;MultipleActiveResultSets=True&quot;"
         providerName="System.Data.EntityClient" />
  </connectionStrings>
</configuration>
```

The highlighted "|DataDirectory|" token is a wrinkle dealt with in the DataLoader.

> The string refers to the application development database, the source for data while running the application as a developer and soon to be the source for design time data as well. Clearly the string for the production database would be something else in the production Web.config.

DataLoader Class

The "DataLoader" console application is a developer utility. It will never be deployed. Simplicity is the watchword; efficiency, robustness, and architecture are irrelevant.

The DataLoader has a single method, **Load**, which is called by the static **Main** in Program.cs:

```
public void Load()
{
    SetDataDirectory();
    var context = new BookClubEntities();

    // Get all categories, then hold onto the first one
    var cat1 = context.Categories.ToList().First();
    Console.WriteLine("Loaded categories including " + cat1.CategoryName);

    // Get a subset of books belonging to the first category
    var books = context.Books.Take(9)
        .Where(b => b.CategoryID == cat1.CategoryID)
        .ToList();
```

```
    // Customizing the design data
    var firstBook = books.First();
    firstBook.Title = "Developers in Wonderland";
    firstBook.EntityAspect.AcceptChanges();

    Console.WriteLine("Loaded books including " + firstBook.Title);

    var bods = context.BookOfDays.Include(b => b.Book).ToList();
    Console.WriteLine("Loaded Book-of-the-days including " +
                      bods.First().Book.Title);

    var cos = context.Checkouts.Include(c => c.Book).ToList();
    Console.WriteLine("Loaded checkouts including " + cos.First().Book.Title);

    context.CacheStateManager.SaveCacheState("../../"+DesignCacheStateFile);
    Console.WriteLine("Save design-time ECS as file, "+DesignCacheStateFile);
}
```

The "DataLoader" creates a BookShelf EntityManager (BookShelfEntities), queries selected entities from the development database, tweaks them, and writes the entity cache to file.

The console application executes four ***synchronous*** queries, something a Silverlight application can't do.

The query results are often typically discarded; we only want the side-effect of filling the entity cache. But the loader can do more than just retrieve database entities. It can add and modify entities programmatically (see the updated title of "firstBook") to suit specific design needs before writing the cache to file.

Saving the ECS to file

It takes just one line to both extract the EntityCacheState and write it to file:

```
    context.CacheStateManager.SaveCacheState("../../"+DesignCacheStateFile);
```

The "DesignCacheStateFile" is the name of the text file added to our project earlier. The "SaveCacheState" method would write to the executable directory which is two levels deeper (bin/Debug); the "../../" prefix backs up to the project directory level and overwrites the initial text file. Subsequent re-executions of the loader will overwrite previous versions of the file.

SetDataDirectory

The app.config holds the connection string to the database. This sample app assumes SQL Server Express and ships with a database in the web project; the given connection string instructs SQL Server Express to attach that database dynamically and locates the database symbolically:

```
  AttachDbFilename=|DataDirectory|\BookClub.mdf;
```

The "|DataDirectory|" is a token which is replaced at runtime with the full file location. If we were in the web project, that location would be something like

```
  C:\Users\Ward\Documents\Visual Studio 2010\Projects\
     Samples\BookShelf\BookShelfDF\BookShelf.Web\App_Data
```

That address changes as you move the solution around on your development machine. The "|DataDirectory|" token accounts for those shifts, leaving the relative part ("\BookClub.mdf") undisturbed.

Unfortunately, the "|DataDirectory|" token is null in this DataLoader project. If we do nothing, the console application won't be able to find the SQL Server Express database and will fail with a helpful error.

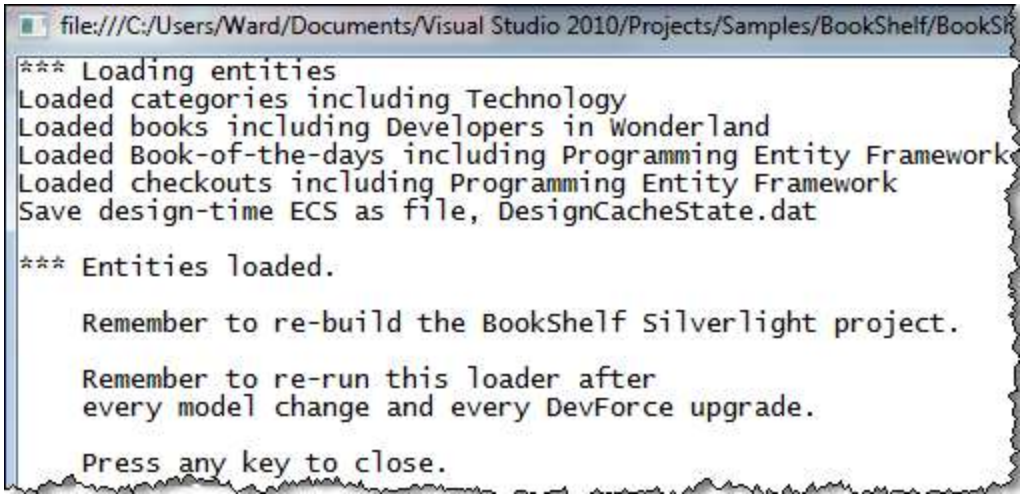
The SetDataDirectory method, called at the top of the Load method, determines the file location dynamically and sets the "|DataDirectory|" token accordingly. See the code for details.

Run The DataLoader

The DataLoader is a utility that you'll run infrequently compared to running the application or its test. A simple way to run it:

- Right-click the project file in the Visual Studio "Solution Explorer" window
- Select "Debug | Start new instance"

The console window shows the loader in action



Visual Studio doesn't detect the changes to the ECS resource file and won't know to rebuild the Silverlight application project on its own. You must manually rebuild the Silverlight BookShelf application whenever you run the Data Loader.

The serialized ECS is a product of a specific data model and DevForce version. Changing the model or upgrading DevForce could make the ECS file unreadable. Such vulnerability to change is a minor drawback of the ECS approach.

Use the ECS File for Design-time Entities

The third (and final) step is to retrieve design-time entities from the ECS file while in visual design tools. Please turn your attention to the BookShelf project.

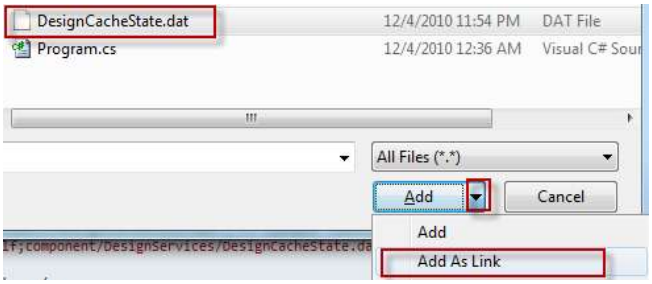
Review the DesignModels folder

The "DesignModels" folder holds the classes that generate collections of entities for View development in visual design tools such as Blend. We'll replace these design entity sources with the ECS file. We could delete this folder and its contents but we won't because it's useful, in a sample application, to show two reasonable ways to achieve this purpose.

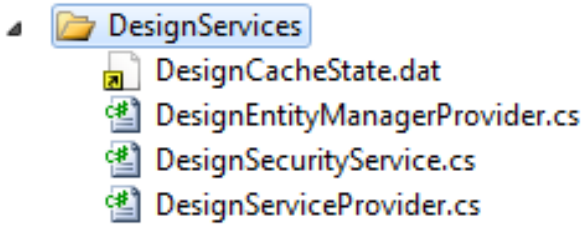
Add a linked ECS file as a Resource

The Silverlight application needs access to the ECS file when open in Blend. We link to it much as we linked earlier to the generated entity class file to get a Silverlight version of the domain model in the web project.

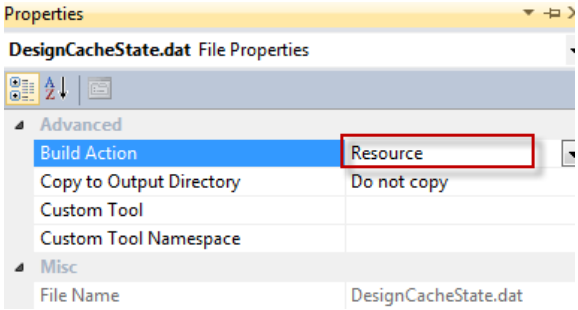
- Select the DesignServices folder in the "Solution Explorer" window
- Pick "Add | Existing item" (Shift – Alt – A)
- Navigate to the BookShelf.DataLoader project
- Select the ECS file, "DesignCacheState.dat"
- Click the drop-down arrow next to the "Add" button
- Click "Add As Link"



A shortcut to that file appears among the members of the DesingServices folder:



Set the linked file's "Build Action" to "Resource".



Revise the DesignEntityManagerProvider

The design entities we see in Blend are created within the **DesignEntityManagerProvider** located in the **DesignServices** folder.

This provider, when it created entities from Design Model classes, looked like this:

```
public class DesignEntityManagerProvider : IEntityManagerProvider
{
    public BookClubEntities CreateManager()
    {
        var context = new BookClubEntities(false /* offline mode */);

        // Populate cache with design data
        PopulateFromDesignModels(context);

        return context;
    }

    /// <summary>
    /// Populate the manager from design-time entities created by code.
    /// </summary>
    private static void PopulateFromDesignModels(EntityManager context)
    {
        context.AttachEntities(new DesignCategories());
        context.AttachEntities(new DesignBooks());
        context.AttachEntities(new DesignCheckouts());
        context.AttachEntity(new BookOfDay
                             {
                                 Day = DateTime.Today,
                                 Book = DesignBooks.ExampleBook,
                             });
    }
```

It created the BookShelf's EntityManager (BookShelfEntities) and attached design-time entities to that manager – entities created in code.

Comment out the call to that method and add a call to a new method,

```
    // Populate cache with design data
    //PopulateFromDesignModels(context);
    PopulateFromEntityCacheState(context);
```

The "populator" that draws upon the ECS file takes a tad more work to set up.

It might not seem worthwhile in an application this size although it's easy to forget the combined 100+ lines in the three Design Model classes. The ECS file approach really pays off when the application model has more than four entity types and the relationships among the entities are numerous and complex.

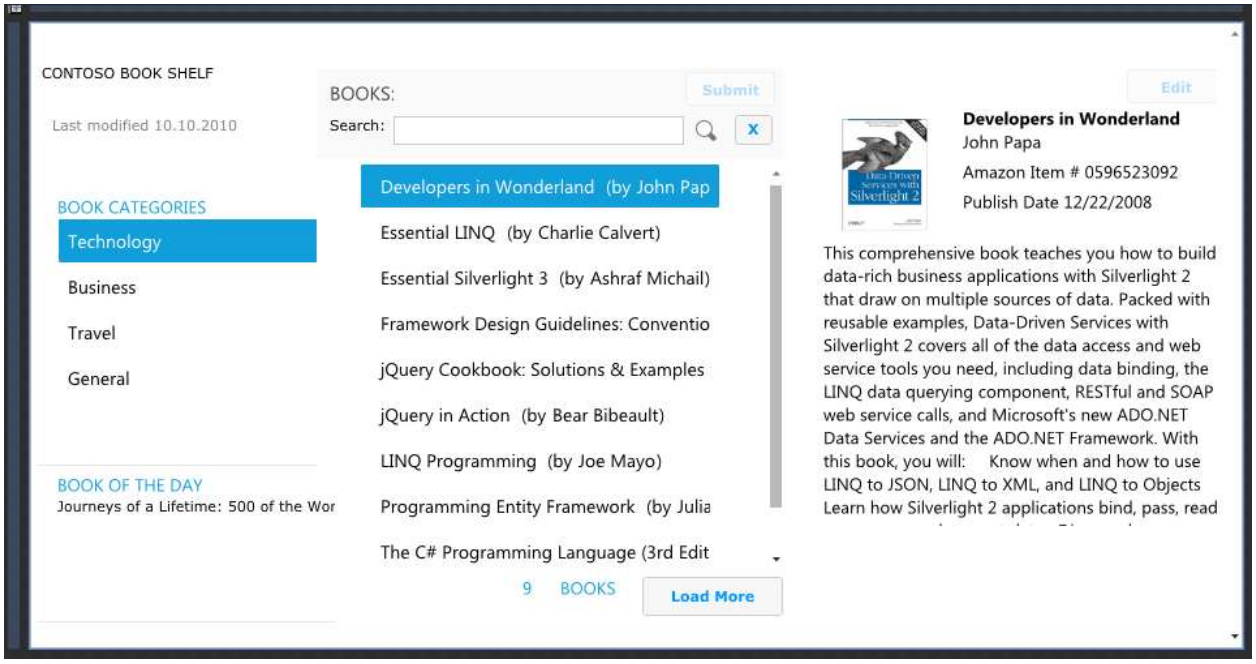
The ECS populator seen here remains the same whether the model is four entities or 400 entities. It proceeds in three stages:

1. Tell DevForce where to find the assembly with the BookShelf model in the static constructor.

2. Create an in-memory EntityCacheState object from the ECS file.
3. Populate the design-time EntityManager from the in-memory EntityCacheState object.

```
static DesignEntityManagerProvider()
{
    // Register the model's assembly name among probed assemblies
    // Must be called BEFORE the first EM creation else
    // GetDesignEntityCacheState fails w/ deserialization exception
    IdeaBlade.Core.IdeaBladeConfig.Instance.ProbeAssemblyNames
        .Add(typeof(BookClubEntities).Assembly.FullName);
}

/// <summary>
/// Get the design-time <see cref="EntityCacheState"/> from resource file
/// </summary>
private static EntityCacheState GetDesignEntityCacheState()
{
    const string filename =
                    @"/BookShelf;component/DesignServices/DesignCacheState.dat";

    var res = Application.GetResourceStream(new Uri(filename, UriKind.Relative));

    return _designEntityCacheState = EntityCacheState.Restore(res.Stream);
}

private static EntityCacheState _designEntityCacheState;

/// <summary>
/// Populate the manager from design-time entities held in
/// an in-memory <see cref="EntityCacheState"/>.
/// </summary>
private static void PopulateFromEntityCacheState(EntityManager context)
{
    var ecs = _designEntityCacheState ?? GetDesignEntityCacheState();
    context.CacheStateManager.RestoreCacheState(ecs, RestoreStrategy.Normal);
}
```

Here's the BookView as seen in Blend, showing the design data sourced from the ECS file.



Exclude the Design Time Support from the Release Build

Both the original RIA BookShelf and this DevForce version include design-time components and concerns in the Silverlight application project. The ECS approach adds a resource to the assembly that adds dead weight to the production release … and the XAP.

That resource weighs about 23KB before compression, less than 1% of the XAP file's hefty 3.7MB. None of this matters now but that XAP will likely have to go on a diet before release and we should remove everything that isn't needed in production.

> Image files, mostly book covers, account for 2.4 MB. A real bookshelf application would retrieve images from the server, not the application assembly.
>
> The compressed BookShelf assembly is 92KB. Its compressed ECS file is 8KB - not much to worry about now. But, in a real world application, it could be much larger and worth excluding.

One could remove unwanted development scaffolding with techniques such as relocating the material to a design-only assembly, adding compiler directives, and adding conditional exclusion logic to the project file's xml – these are beyond the scope of this document.

Summary

Thanks for following along with our conversion of Microsoft's BookShelf MVVM example to DevForce.

We promised more simplicity and more robust functionality.

Robustness enhancements include:

- Error handling added to the BookDataService API
- The Edit and Submit buttons are disabled after logout
- Cancelling the Book Editor rolls back changes to the current book
- No more multiple confirmation dialogs that freeze the UI
- The Book Editor is independent of the BookView and the BookViewModel
- Book Editor launched directly, without Messenger involvement
- Security rules applied in the ViewModels, not hidden in xaml resource binding
- Less (sometimes nothing) in the View code-behind files
- No DomainService to fail or maintain
- Automated tests pave the way for future improvements with fewer regressions
- EntityCacheState simplifies creation, maintenance, and use of design-time data

We said "simplicity" could be measured in reduced lines of code and fewer classes. But adding robustness (e.g., error handling) , testability (e.g., depending on interfaces), features (e.g., rollback) takes more code. How did we do?

It's fair to exclude the Test and DataLoader projects which has no bearing on the complexity of the application while potentially improving our confidence in application quality.

BookViewModel is a good proxy for lines of code. We've worked the BookViewModel over many times, adding features while tidying up. Yet it is twenty-some lines shorter than the original RIA Services version. We dropped the huge RIA-generated BookShelf.Web.g.cs file and three DomainServices class files in the web project. On the other hand, we have a net gain of eight helper classes among the two "Services" folders. They're small and they help decouple the ViewModels from unwanted dependencies but adding classes inevitably increases complexity. It's something to keep our eye on.

We hope you feel it's been worthwhile. We hope that you have appreciated the explanations and reasoning behind each design decision even if you disagree with some of them.

While there's much more we could do, we trust you'll find plenty to chew on right here.

Please send issues or questions to support@Ideablade.com