Learning to Master Cross-Platform Mobile Development with Xamarin

Brought to you by Falafel Software

Need help with Xamarin? Learn about our services or call us at 831-462-0457

About the Author

Jesse Liberty

Jesse Liberty is a Master Consultant for Falafel Software, an author, and he creates courses for Pluralsight. Jesse hosts the popular 'Yet Another Podcast' and his blog is required reading. He was a Senior Evangelist for Microsoft, a XAML Evangelist for Telerik, a Microsoft MVP, Distinguished Software Engineer at AT&T; Software Architect for PBS and Vice President of Information Technology at Citibank.

Jesse can be followed on twitter at @JesseLiberty

Contents

Part 1: Forms 6

Forms 6
Two Approaches 6
Getting Started 6
Creating the UI 9
App.cs 10
Details Page 10
Run It 11

Part 2: Forms and XAML 11

Views 12
Setting Up Navigation 14

Part 3: Xamarin Data Binding - Hello XAML Programmers! 15

What We'll Build 16
The Book Class 16
Creating the Dummy Data 18
Displaying the Books 19
Book Details 21
Two Way Binding 22

Xamarin iOS Storyboards - Getting Started 22

Getting Started 22
Setting Properties 24
Dismissing the Keyboard 24
Xamarin iOS Storyboards - Events 25

Xamarin Storyboards - Navigation 26

Segues 26
Creating New Controllers 27
Passing Data 27
Displaying the Books 28

Storyboards and Custom Cells 30

Subclassing UITableViewCel 30
Passing in Books 32

Overriding GetCell....33

Learning Xamarin - Building An App 34

The APP 34
The Data Model 34
M-V-VM 37
Views....38
Adding Criteria....40
Details, Details 41

Xamarin Forms: Converting Integers 43

Moving Add Criterion 45
Adding Items 46

Tinkering with our app 48

Importance Of Criteria....49

Learning Xamarin - Adding Persistence with SQLite 50

Adding the Database To The Project....51
Saving Items and Criteria To The Database....53
Wiring It Up....53
iOS Specific....54
Calling Into The Database....55
Restoring Items 56
Not Rocket Science....57

Learning Xamarin - Extending the App & DB Tables 57

Updating the Data 58
Appliances....58
Patterns....62

Learning Xamarin - Adding Tabbed Pages....62

Android Too 63
Xamarin Forms and HTTP....64
Creating the Model 64
Creating the View Model....64
Creating the View....65
Services....65

Xamarin Forms: Web Viewer 68

Wiring It Up....69

Closing 69

Part 1: Forms

This book is a series of chapters on learning cross-platform mobile development with Xamarin.

Forms are particularly appealing as an entry point because they use XAML and C#, which allows me to dig into my muscle memory from days gone by when I coded for Silverlight, WPF and Windows Phone. So that is where I'll start:

Forms

Forms allow you to reuse a great deal of your UI code cross-platform, while still building native UIs for all three platforms with a single shared C# codebase. This is a very good thing.

Two Approaches

There are two fundamental approaches to using Forms and creating your UI. One is to write the UI code in C#, the other is to do so in XAML.

Let's start by creating everything in code…

Getting Started

We begin by firing up Xamarin Studio (or, if you prefer, Visual Studio). Because I'm incredibly lazy I'm going to steal the data that James Montemagno so carefully assembled. To do so, the first thing I'll do is create a Monkey class in the Models folder:

```
public class Monkey
{
 public string Name {get;set;}
 public string Location { get; set; }
 public string Details { get; set; }
}
```

Notice the Details property. This will be used when we set up our master/details pages.

Next, we turn to creating our ViewModel. If you are not familiar with MVVM, now is the time to catch up as MVVM is critical in many of the newest programming technologies, including Angular.js, which I'm writing about in another series.

Here is the contents of MonkeysViewModel which goes in the ViewModels folder:

```
using Monkeys.Models;
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Monkeys.ViewModels
{
 public class MonkeysViewModel
 {
  public ObservableCollection<Monkey> Monkeys { get; set; }

  public MonkeysViewModel()
  {
   Monkeys = new ObservableCollection<Monkey>();
   Monkeys.Add(new Monkey
   {
    Name = “Baboon”,
    Location = “Africa & Asia”,
    Details = “Baboons are African and Arabian Old World
    monkeys belonging to the genus Papio,
    part of the subfamily Cercopithecinae.”
   });

   Monkeys.Add(new Monkey
   {
    Name = “Capuchin Monkey”,
    Location = “Central & South America”,
    Details = “The capuchin monkeys are New World monkeys
    of the subfamily Cebinae. Prior to 2011, the subfamily
    contained only a single genus, Cebus.”
   });

   Monkeys.Add(new Monkey
   {
    Name = “Blue Monkey”,
    Location = “Central and East Africa”,
    Details = “The blue monkey or diademed monkey
    is a species of Old World monkey native to Central and
    East Africa, ranging from the upper Congo River basin
    east to the East African Rift and south to northern
    Angola and Zambia”
   });


   Monkeys.Add(new Monkey
   {
    Name = “Squirrel Monkey”,
    Location = “Central & South America”,
    Details = “The squirrel monkeys are the New World
```

```
 monkeys of the genus Saimiri. They are the only
 genus in the subfamily Saimirinae. The name of
 the genus Saimiri is of Tupi origin, and was
 also used as an English name by early
 researchers."
});

Monkeys.Add(new Monkey
{
 Name = "Golden Lion Tamarin",
 Location = "Brazil",
 Details = "The golden lion tamarin also known as
 the golden marmoset, is a small New World monkey of
 the family Callitrichidae."
});

Monkeys.Add(new Monkey
{
 Name = "Howler Monkey",
 Location = "South America",
 Details = "Howler monkeys are among the largest
 of the New World monkeys. Fifteen species are
 currently recognized. Previously classified in
 the family Cebidae, they are now placed in the
 family Atelidae."
});

Monkeys.Add(new Monkey
{
 Name = "Japanese Macaque",
 Location = "Japan",
 Details = "The Japanese macaque, is a terrestrial
 Old World monkey species native to Japan. They are
 also sometimes known as the snow monkey because
 they live in areas where snow covers the ground
 for months each"
});

Monkeys.Add(new Monkey
{
 Name = "Mandrill",
 Location = "Southern Cameroon, Gabon,
  Equatorial Guinea, and Congo",
 Details = "The mandrill is a primate of the Old
 World monkey family, closely related to the baboons
 and even more closely to the drill. It is found
 in southern Cameroon, Gabon, Equatorial Guinea,
 and Congo."
});

Monkeys.Add(new Monkey
{
 Name = "Proboscis Monkey",
 Location = "Borneo",
```

```
    Details = "The proboscis monkey or long-nosed
    monkey, known as the bekantan in Malay, is a
    reddish-brown arboreal Old World monkey that is
    endemic to the south-east Asian island of Borneo."
   });
  }
 }
}
```

That's a lot to copy, but it is really just an initialization of data for our project.

Creating the UI

As mentioned above, there are two ways to create the UI. In this chapter, we'll create cross-platform user interface in code. Create a new file under Views called MonkeyPage.cs.

In this page our key goal is to create a ListView to display our list of Monkey Names and Locations. We can fill the page with the ListView by setting it to the content of the page.

```
public class MonkeysPage : ContentPage
{
 public MonkeysPage()
 {
  Title = "Monkeys";

  var list = new ListView();
   //...

  Content = list;
 }
}
```

Next, we want to bind our list to the ViewModel; We do this by way of the ItemSource property. Monkeys is an ObservableCollection and so the ListView will be updated when items are added or removed from the collection:

```
var viewModel = new MonkeysViewModel();
list.ItemsSource = viewModel.Monkeys;
```

Display is handled by a DataTemplate. The DataTemplate works as you'd expect if you've used such templates in other environments; That is, you describe how each item should be displayed and that pattern is repeated for every item. In this case, however, we're going to create our DataTemplate entirely in code:

```
var cell = new DataTemplate(typeof(TextCell));

cell.SetBinding(TextCell.TextProperty, "Name");
cell.SetBinding(TextCell.DetailProperty, "Location");

list.ItemTemplate = cell;
```

Here we see the built in binding provided by Xamarin Forms. Again, this will be familiar if you've worked with Silverlight, WPF, Windows, Windows Phone, Angular.js or any MVVM framework.

Finally, we need to tell the ListView how to respond when an item is selected by the user.

```
list.ItemTapped += (sender, args) =>
{
 var monkey = args.Item as Monkey;
 if (monkey == null)
  return;

 Navigation.PushAsync(new DetailsPage(monkey));
 list.SelectedItem = null;
};
```

The key line in this code is

```
Navigation.PushAsync(new DetailsPage(monkey));
```

And that is how page to page navigation is accomplished, passing along the selected monkey item.

App.cs

In App.cs there is a GetMainPage method that dictates how the application will get the first page. Here we'll get the MonkeysPage, but we'll wrap it in a NavigationPage so that we can navigate to it later on.

```
public class App
{
 public static Page GetMainPage()
 {
  var monkeys = new MonkeysPage();
  return new NavigationPage(monkeys);
 }
}
```

Details Page

If the user clicks on one of our Monkeys we want, as you saw above, to navigate to a Details page. Let's create that now. That too goes in the Views folder, and the contents are pretty straight forward:

```
public class DetailsPage : ContentPage
{
 public DetailsPage(Monkey monkey)
 {
  this.Title = monkey.Name;

  var details = new Label{
   Text = monkey.Details
  };

  Content = new ScrollView
  {
   Padding = 20,
```

```
    Content = details
  };
 }
}
```

The one interesting bit of new code is that the Content is set to a ScrollView and within that content is set to the details we obtain from the monkey instance.

Run It

Before going any further, let's run the application – Choose Debug and either deploy to your phone or pick one of the simulator types.

The result is a native application that looks great and responds to your touch. Click on one of the monkeys and you are taken to the details page.

In the next chapter, I'll revisit this project and implement the UI using XAML.

Part 2: Forms and XAML

In the previous chapter I showed how to create a simple cross-platform mobile application using the new **Xamarin Forms.** In this chapter, I will recreate that same application, but using XAML.

> The XAML that is used in Xamarin is very (very!) close to the XAML that was used in Silverlight and is used in WPF, Windows Phones and Windows 8.x. So close, that Xamarin Forms are a natural and happy path for XAML programmers into mobile application development.

Once again, fire up Xamarin Studio (or Visual Studio). We're going to reuse the Monkey class and the MonkeyVewModel that we used in the previous chapter.

When using XAML the data binding will be explicit and the use of a ViewModel will be even more important.

Views

Create a folder Views, and in that folder start by creating a new Forms ContentPage named MonkeyPage.xaml (which will also create MonkeyPage.xaml.cs).

If you know XAML, the contents of this page will look very familiar:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/
forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="Monkeys.Views.MonkeysPage"
       Title="Monkeys">
 <ListView x:Name="list" ItemsSource="{Binding Monkeys}"
       ItemTapped="OnItemSelected">
  <ListView.ItemTemplate>
   <DataTemplate>
    <TextCell Text="{Binding Name}"
      Detail="{Binding Location}">
    </TextCell>
   </DataTemplate>
  </ListView.ItemTemplate>
 </ListView>
</ContentPage>
```

We declare a few namespaces and then we get down to business, creating a ListView and setting its ItemsSource property to bind to the Monkeys collection. You'll see how that gets wired up in just a moment. We also bind an event handler to ItemTapped (the event fired when the user taps on an item).

Inside the ListView we create an ItemTemplate, which describes how each item will be displayed. To show both the name and the location, we use a single TextCell object, and use data binding from each object to the Text and Detail properties, respectively.

In the code-behind file, MonkeyPage.xaml.cs, we start, in the constructor by initializing the BindingContext to the ViewModel:

```
public partial class MonkeysPage
{
 public MonkeysPage()
 {
  InitializeComponent();
  this.BindingContext = new MonkeysViewModel();
 }
```

We also supply the implementation of the event handler:

```
public void OnItemSelected(object sender, ItemTappedEventArgs args)
 {
   var monkey = args.Item as Monkey;
```

```
   if (monkey == null)
     return;
//…
 }
```

I've elided the navigation code for now, we'll return to this shortly.

When the user taps on a monkey, we want to navigate to the details page. So let's create DetailsPage.xaml in Views:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="Monkeys.Views.DetailsPage"
       Title="{Binding Name}">
 <ScrollView Padding="20">
  <Label Text="{Binding Details}" />
 </ScrollView>
</ContentPage>
```

Once again, this is standard XAML. We create a title, which we bind to the Name property of the selected monkey, and we place the bound details in a label which in turn is placed in a ScrollView.

All we need to do in DetailsPage.xaml.cs is set the BindingContext to the Monkey that gets passed in:

```
public partial class DetailsPage
{
 public DetailsPage(Monkey monkey)
 {
  InitializeComponent();
  this.BindingContext = monkey;
 }
}
```

Setting Up Navigation

We need to set the start page, which we do in App.cs. Because we want to provide navigation, a topic to which we'll return in future chapters, we wrap that first page in a NavigationPage:

```
public class App
{
  public static Page GetMainPage ()
  {
    var monkeyPage = new MonkeysPage ();
    return new NavigationPage (monkeyPage);
  }
}
```

Finally, we return to MonkeyPage.xaml.cs and complete the OnItemSelected method, using the Navigation capabilities of the NavigationPage.

```
public void OnItemSelected(object sender, ItemTappedEventArgs args)
{
 var monkey = args.Item as Monkey;
 if (monkey == null)
  return;

 Navigation.PushAsync(new DetailsPage(monkey));
 list.SelectedItem = null;
}
```

I'll test this on my new Nexus 5 Android phone, purchased specifically for Xamarin cross-platform development (I already had an iPhone and a Windows Phone). You can see that it works identically to the code-only approach followed in the previous chapter.

While either approach works very well with forms, the XAML approach has the advantage of being declarative, and thus, theoretically, easier to maintain and easier to use with tools. Unfortunately, for the moment, there is no designer, and so all the XAML has to be written by hand.

If you are experienced with XAML and writing it by hand is not daunting, then I personally recommend this approach. I find the data binding to be much more explicit and the organization of the files makes sense to me. But the code-only approach is perfectly valid and may be easier for those who are comfortable with C# but new to XAML.

There is much more to forms, and I'll be tackling somewhat more advanced topics in coming chapters.

Xamarin Forms make creating cross platform UI's incredibly easy, especially for LOB applications. I'll be returning to Xamarin Forms many times in the rest of this book.

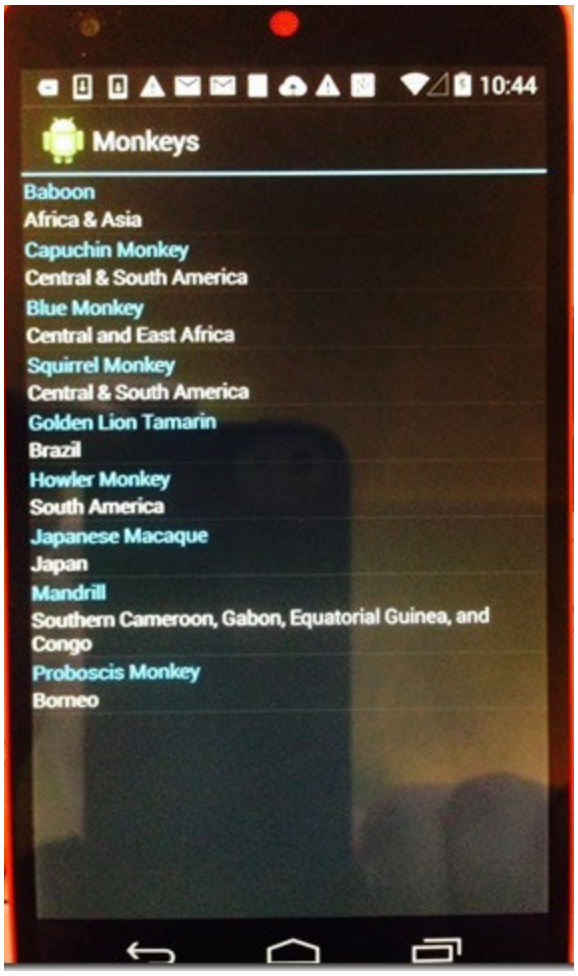
Part 3: Xamarin Data Binding - Hello XAML Programmers!

Xamarin happily supports two-way binding, and the mechanism will be intensely familiar to XAML programmers. The more I work with Xamarin, the more I think this is where Silverlight went to heaven.

To illustrate this, let's create a simple Forms application that provides two way binding of information about books.

> A short aside... One of the interesting and frustrating things when learning a new technology is that you read a blog post or watch a video and it all makes sense. Then it is time to create something new and... <poof> all you thought you knew is gone.

Let's struggle through this application together, starting relatively simply and adding features as we go, over several chapters.



What We'll Build

We begin with a spec: I'd like to keep track of the books that I've loaned out to others. I read a lot, my wife reads a lot, and we have thousands of books, many of which are leant to friends in large shopping bags (the books are in the bags, not the friends) . It would be good to be able to quickly enter the name of the book and who I loaned it to and then have it keep track of all my leant out books, and whether they've been returned.

We'll start small in the "Get it working and keep it working" design philosophy.

The Book Class

To start, we need a class for the Books themselves. There is a lot of data we may want in such a class, but we can strip it down to the essentials. We know, however, that we want these Book items to appear in a list, and to be updated when they are updated in the UI. Thus, we know, right out of the gate, that we want the Book class to implement INotifyPropertyChanged.

Just as it was in Silverlight, WPF, etc., INotifyPropertyChanged is the essential interface for supporting data binding. The interface itself is dead simple, it requires only one event:

```
public event PropertyChangedEventHandler PropertyChanged;
```

The pattern that is used in support of this interface is to eschew automatic properties, create backing fields, and in the setter to check to make sure that you have a new value. If so, then assign it and then to call a helper method, OnPropertyChanged.

The job of OnPropertyChanged is to call PropertyChanged if anyone has registered with it. This way, every time the underlying value changes, the UI can be updated.

Here is the code for our Book class:

```
using System;

using System.ComponentModel;

using System.Runtime.CompilerServices;

namespace Booktracker

{

    public class Book : INotifyPropertyChanged
    {
        private string _name;
```

```
public string Name {
    get { return _name; }
    set {
        if (value.Equals (_name, StringComparison.Ordinal))
            return;
        _name = value;
        OnPropertyChanged ();
    }
}
private string _author;
public string Author {
    get { return _author; }
    set {
        if (value.Equals (_author, StringComparison.Ordinal))
            return;
        _author = value;
        OnPropertyChanged ();
    }
}
private string _borrower;
public string Borrower {
    get { return _borrower; }
    set {
        if (value.Equals (_borrower, StringComparison.Ordinal))
            return;
        _borrower = value;
        OnPropertyChanged ();
    }
}
private DateTime _dateBorrowed;
public DateTime DateBorrowed {
    get { return _dateBorrowed; }
    set {
        if (value.Equals ((DateTime)_dateBorrowed))
            return;
        _dateBorrowed = value;
        OnPropertyChanged ();
    }
}
private bool _returned;
public bool Returned {
    get { return _returned; }
    set {
        if (value.Equals (_returned))
            return;
        _returned = value;
        OnPropertyChanged ();
```

```
            }
        }
        private Uri _imageUri;
        public Uri ImageUri {
            get { return _imageUri; }
            set {
                if (value.Equals (_imageUri))
                    return;
                _imageUri = value;
                OnPropertyChanged ();
            }
        }


        public event PropertyChangedEventHandler PropertyChanged;


        void OnPropertyChanged ([CallerMemberName] string propertyName = null)
        {
            var handler = PropertyChanged;
            if (handler != null) {
                handler (this, new PropertyChangedEventArgs (propertyName));
            }
        }
    }
}
```

Creating the Dummy Data

Eventually this data will be persisted, but for now we'll just generate it each time, in App.cs:

```
public static List<Book> BorrowedBooks = new List<Book> {
            new Book {
                Name = "War and Peace":
                Author = "Tolstoy":
                Borrower = "Karp":
                DateBorrowed = new DateTime (2014, 1, 15):
                Returned = false
            }:
```

```
        new Book {
            Name = "Anatham":
            Author = "Stephenson":
            Borrower = "Weiss":
            DateBorrowed = new DateTime (2014, 2, 1):
            Returned = false
        }:
        new Book {
            Name = "Pro Windows 8":
            Author = "Liberty":
            Borrower = "Hurwitz":
            DateBorrowed = new DateTime (2014, 5, 1):
            Returned = false
        }:
        new Book {
            Name = "Far From the Tree":
            Author = "Solomon":
            Borrower = "Belove":
            DateBorrowed = new DateTime (2014, 3, 15):
            Returned = false
        }:
        new Book {
            Name = "Ulysses":
            Author = "Joyce":
            Borrower = "Tadros":
            DateBorrowed = new DateTime (2014, 5, 15):
            Returned = false
        }
    };
```

Simple. Now we're ready to think about how to display this. For now, to get started and to keep things as easy as possible I thought I'd create a List on one page and when the user touches a book I'd bring her to the details page.

Displaying the Books

I very much wanted to create the UI in XAML (see the previous chapter) and doing so is very straight-forward if you are used to XAML programming. If not, the Matryoshka Doll effect can make you a little crazy.

What we want to do is to create a ListView and teach that ListView how to display each item in the list. To do so we create an ItemTemplate which begins with a DataTemplate. Inside the DataTemplate we place a ViewCell and within that the ViewCell.View which will contain the actual controls to be used for each item. We'll use a StackPanel to stack two controls, one above the other for each item, and Labels to display (and bind to) the text for the Name and Author properties of the items.

All of this is stuffed inside the ContentPage View so that it will be properly displayed as one page of the application. Here's the code. Don't freak out, if you work your way down through the levels it all begins to make sense:

```
<?xml version="1.0" encoding="UTF-8"?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"

             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:Booktracker;assembly=Booktracker"
             x:Class="Booktracker.MainPage"
             Title="Borrowed Books">
         <ListView x:Name="listView"
                  IsVisible="true"
                  ItemsSource="{x:Static local:App.BorrowedBooks}">
                      <ListView.ItemTemplate>
                          <DataTemplate>
                              <ViewCell>
                                  <ViewCell.View>
                                      <StackLayout Orientation="Vertical" Horizon-
talOptions="CenterAndExpand">
                                                  <Label Text="{Binding Name}" Hori-
zontalOptions="Center" />
                                                  <Label Text="{Binding Author}"
Font="10" HorizontalOptions="Center"/>
                                              </StackLayout>
                                  </ViewCell.View>
                              </ViewCell>
                          </DataTemplate>
                      </ListView.ItemTemplate>
                  </ListView>
</ContentPage>
```

Having described the entire page in markup, there is little for the code-behind to do.Its main job is to wire up and implement the event handler for when an item is selected,

```
01.public partial class MainPage : ContentPage

02.{

03.    public MainPage ()

04.    {
```

```
05.         InitializeComponent ();
06.         listView.ItemSelected += (sender, e) => {
07.             var selectedBook = (Book)e.SelectedItem;
08.             var bookDetailsPage = new BookDetails (selectedBook);
09.             Navigation.PushAsync (bookDetailsPage);
10.         };
11.     }
```

The key lines are 7 and 8 where we determine which book was selected, and pass that book to the BookDetails page. Line 9 uses the Navigation infrastructure (about which more in later chapters) to navigate to the BookDetails page.

Book Details

BookDetails is also written in XAML, only this time we want to stack up all the fields with labels to identify them, and we want them to be editable. To do this, we'll use Entry controls for text entry, a DatePicker to pick the date the book was borrowed, and a Switcher to toggle true or false for whether or not the book has been returned:

```
<?xml version="1.0" encoding="UTF-8"?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms" xmlns:x="http://schemas.micro-
soft.com/winfx/2009/xaml" x:Class="Booktracker.BookDetails">

    <ContentPage.Content>
        <StackLayout Orientation="Vertical" HorizontalOptions="StartAndExpand">
                <StackLayout Orientation="Horizontal">
                    <Label Text="Book Name "  HorizontalOptions="FillAndExpand"/>
                    <Entry Text="{Binding Name}" HorizontalOptions="EndAndExpand" />
                </StackLayout>
                <StackLayout Orientation="Horizontal">
                    <Label Text="Author " HorizontalOptions="FillAndExpand" />
                    <Entry Text="{Binding Author}" HorizontalOptions="EndAndExpand" />
                </StackLayout>
                <StackLayout Orientation="Horizontal">
                    <Label Text="Borrowed by " HorizontalOptions="FillAndExpand" />
                    <Entry Text="{Binding Borrower}" HorizontalOptions="EndAndExpand"
/>
                </StackLayout>
            <StackLayout Orientation="Horizontal" >
                    <Label Text="Date Borrowed" HorizontalOptions="FillAndExpand" />
                    <DatePicker Date="{Binding DateBorrowed}" HorizontalOptions="En-
dAndExpand"/>
                </StackLayout>
```

```
                <StackLayout Orientation="Horizontal" >
                    <Label Text="Returned?" HorizontalOptions="FillAndExpand"/>
                    <Switch IsToggled="{Binding Returned}" HorizontalOptions="EndAn-
dExpand"/>
                </StackLayout>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

The code-behind, BookDetails.xaml.cs is responsible for creating the binding between the page and the selected Book that is passed in from the first page:

```
public partial class BookDetails : ContentPage

{
    public BookDetails (Book selectedBook)
    {
        InitializeComponent ();
        this.BindingContext = selectedBook;
    }
}
```

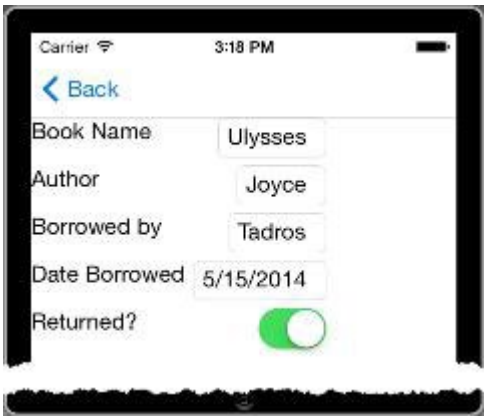
Two Way Binding

Because Xamarin supports two-way binding out of the box, any changes you make in the UI will be reflected in the underlying data, even though at the moment that data is just residing in memory.

You can see that with forms you can quickly create native applications that look great (well, it would look great if I had a designer working with me!)

Xamarin iOS Storyboards - Getting Started

The first three chapters in this book were all about Xamarin Forms. That is because Forms are terrific, convenient and new. But they are not the only game in town.

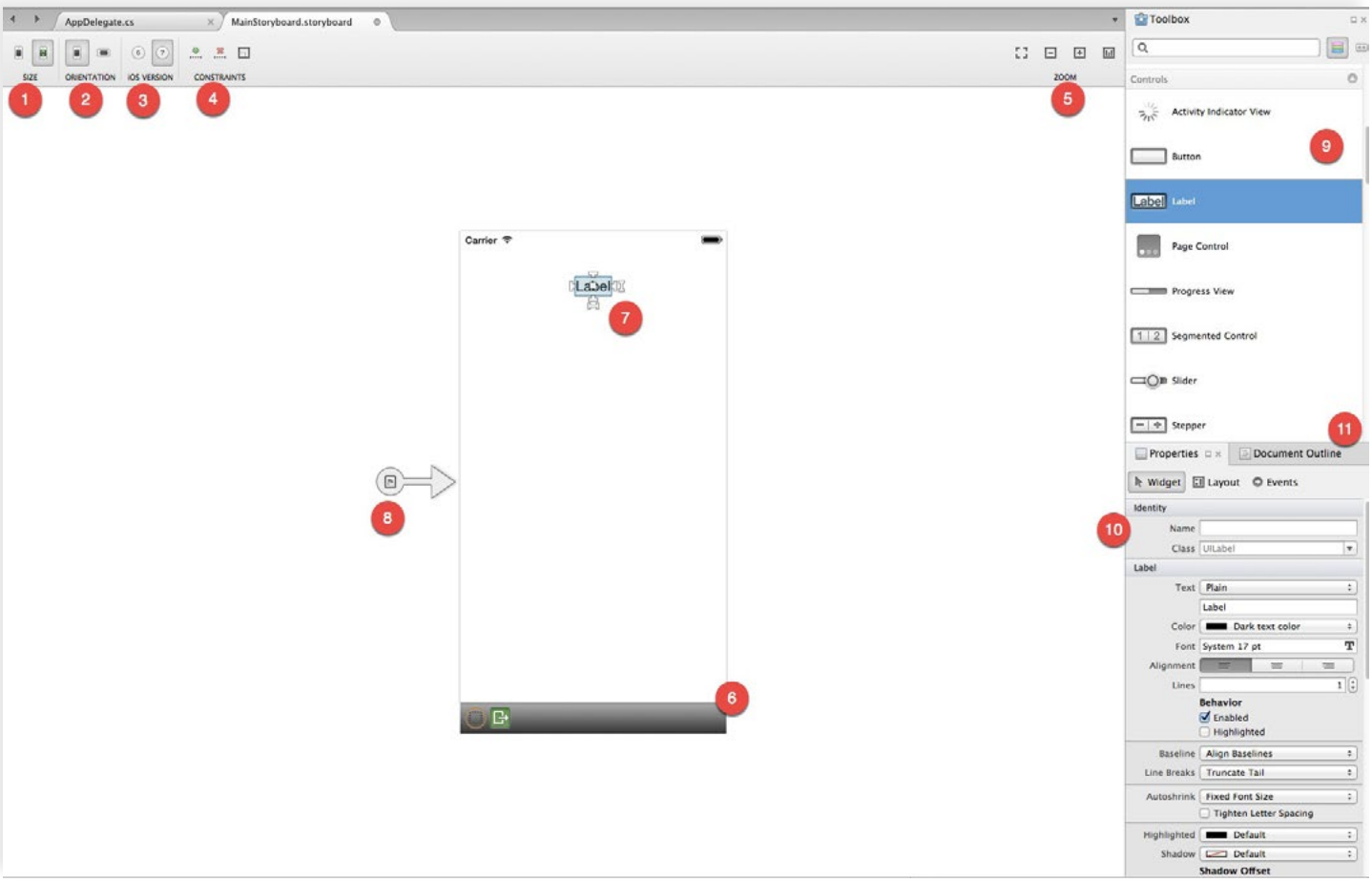


It is possible to write iOS applications using native controls and reap the benefit of a visual designer. Let's take a look at working with the Xamarin Storyboard.

Getting Started

Create a new project and pick IOS->iPhone->Single View Application, as you have done in the past. I named mine StoryboardSample.

Notice that your application includes a MainStoryboard.storyboard file. Double click on it to open the IOS designer.



The designer has three main areas and a number of toolbar buttons. I've numbered various parts of the designer as follows:

1. Pick the size iPhone you are targeting
2. Pick the orientation you are targeting
3. Select iOS version 6 or 7
4. Add or remove constraints
5. Zoom in or out on the design surface
6. The dark gray bar can be used to drag the page on the surface
7. I've put a label on the form which I'm ready to edit
8. This is for navigation and we'll come back to it
9. The toolbox has standard views but also layout views
10. The properties window is where you set the (surprise!) properties for the selected view
11. Notice the Document Outline tab which shows you all the views on the designer

Click on the first constraints button (Add Recommended Constraints).

Setting Properties

When you click on the label it sprouts resizing handles which you can use to set the width and height of the label. As you stretch the label horizontally, guides will appear to help you size it appropriately.

Set the Text property of the label to say Title.

Drag a Text Field onto the design surface and resize it to the same width as the label. Give this widget a Name: BookTitle

Add a label and Text Field for Author (we'll assume a single author for now). Name the Text Field Book-Author

Add a label and DatePicker for the Purchase Date and name the date picker PurchaseDate

Finally, add a Button set its text to Add and set its name to AddBook

Run the application in the simulator or on your device. After you fill in a book and author, it should look similar to this image:

Dismissing the Keyboard

While it looks good, there are a number of practical problems, the most immediate of which is that the keyboard is not going away after you enter the author, and so there is no way to enter the PurchaseDate.

To fix that, we'll need to write a little code, specifically we need to wire into the ShouldReturn event. This allows us to resign focus from the text field by calling UITextField.ResignFirstResponder().

Double click on the associated controller (in my case, StoryboardSampleViewController). Find the View-DidLoad() method.

Add the following code right after the call to the base implementation of ViewDidLoad():

```
BookAuthor.ShouldReturn += delegate {
    BookAuthor.ResignFirstResponder ();
    return true;
};
```

Now run the program again and you'll find that if you hit enter after entering the Author name, the keyboard will disappear. That should allow you to pick a date and press the Add button. Alas, the Add button doesn't do anything. We need to wire up that button and that will be the next chapter.

Xamarin iOS Storyboards - Events

In the previous chapter, I left off with a page with a button which did nothing. The key question is how to handle the associated event. This brief chapter will answer that question.

It turns out that handling the event is frighteningly familiar to .NET programers. In fact, you can probably guess at the code and come pretty close.

First, we return to the ViewDidLoad method and we add an event handler for the TouchUpInside event for the button. TouchUpInsde means that this event will fire when the button is released after touching and releasing inside the boundaries of the button. It is more natural for an event to fire on release than on, e.g., TouchDownInside.

The first thing we'll do in our event handler is to gather the user's input from the three controls on the page:

```
AddBook.TouchUpInside += (sender, e) => {
                var title = BookTitle.Text;
                var author = BookAuthor.Text;
                var tempDate = BookPurchaseDate.Date;
```

It turns out that the date you get back is of type NSDate and we'd like a System.DateTime to make formatting easier. Fortunately, the conversion is automatic:

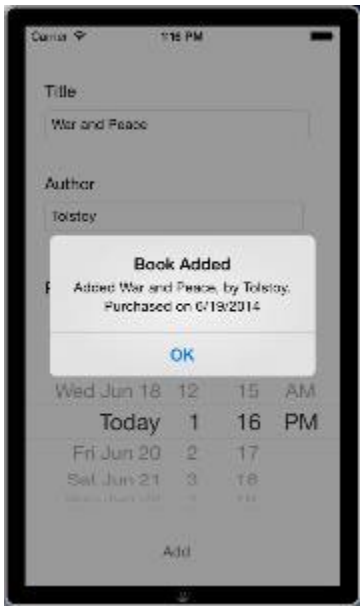
```
System.DateTime purchaseDate = tempDate;
```

We're now in a position to show an alert with the user's selection. Here's the complete event handler for context,

```
AddBook.TouchUpInside += (sender, e) => {
                var title = BookTitle.Text;
                var author = BookAuthor.Text;
                var tempDate = BookPurchaseDate.Date;
                System.DateTime purchaseDate = tempDate;
                var alert = new UIAlertView () {
                    Title = "Book Added":
                    Message = "Added " +
```

```
            title + ", by " +
            author + ". Purchased on " +
            purchaseDate.ToShortDateString ()
    };
        alert.AddButton ("OK");
        alert.Show ();
    };
```

The result is shown in the image:



Xamarin Storyboards - Navigation

In the previous chapter, I left off with binding data. Now we take a look at page navigation and passing that data around.

Navigation with Storyboards is accomplished with Segues. This process is so easy it is almost frightening. Passing data is almost as easy.

Segues

To get started, return to the application we left off previously, and reopen MainStoryboard.storyboard. Drag a Navigation controller onto the design surface and notice that it brings a second controller with it. It may be necessary to use the zoom feature to make enough room to see all three controllers. Delete the second controller you just dragged onto the design surface, so that you end up with the Navigation controller and the original View Controller.

Notice that there is an arrow pointing to the ViewController. This is the segue. Click and drag it onto the Navigation controller. Now that first segue points to the Navigation Controller. Next, click and drag from anywhere on the Navigation Controller onto the View Controller and in the pop up menu that appears when you let go of the mouse, click on Root. You've just made the View Controller the root controller for

Navigation.

Let's now add a new controller of type TableViewController. Place it to the right of View Controller for convenience, and click-drag from the Review button to the new Controller. In the pop up window, click Push.

Notice that the top of the new ViewController now has a navigation control to get back to the first Controller.

Creating New Controllers

Name your Table View Controller BookListController and you'll see BookListController.cs appear in the list of files. Nice. Also, name the TableView itself (inside the Controller in the storyboard) BookListTableView.

Passing Data

The basic navigation already works (run it and try) but you're not passing data. To do so, open StoryBoardSampleViewController and add a collection to hold the books:

```
private List<Book> books = new
List<Book> ();
```

Declare the same collection in BookListController.

To support this, we'll need a Book class:

```
public class Book

{

        public string Name { get;
set; }
        public string Author {
get; set; }
        public DateTime DatePur-
chased { get; set; }
}
```

Here is an excerpt from the Xamarin documentation on the different types of segues

- Push - A push Segue adds the View Controller to the navigation stack. It assumes the View Controller originating the push is part of the same Navigation Controller as the View Controller that is being added to the stack. This does the same thing as pushViewController, and is generally used when there is some relation between the data on the screens. Using the Push Segue gives you the luxury of having a Navigation bar with a back button and title added to each View on the stack, allowing drill down navigation through the View Hierarchy.

- Modal - A Modal Segue create a relationship between any two View Controllers in your Project, with the option of an animated transition being shown. The child View Controller will completely obscure the Parent View Controller when brought into view. Unlike a Push Segue, which adds a back button for us; when using a modal segue DismissViewController must be used in order to return to the previous View Controller.

- Custom - Any custom Segue can be created as a subclass of UIStoryboardSegue.

- Unwind - An unwind Segue can be used to navigate back through a push or modal segue - for example by dismissing the modally-presented view controller. In addition to this, you can unwind through not only one, but a series of push and modal segues and go back multiple steps in your navigation hierarchy with a single unwind action. To understand how to use an unwind segue in the iOS, read the Creating Unwind Segues recipe.

- Sourceless - A Sourceless Segue indicates the Scene containing the Initial View Controller and therefore which View the user will see first.

We'll be notified when it is time to transition to the new page. To make this work, override Prepare-ForSegue, make an instance of our new BookListController and pass our books collection to its books collection.

```
public override void PrepareForSegue (UIStoryboardSegue segue, NSObject sender)

{
    base.PrepareForSegue (segue, sender);
    var bookListController = segue.DestinationViewController as BookListController;
    if (bookListController != null) {
        bookListController.books = books;
    }
}
```

Displaying the Books

We have a small problem. The BookListTableView's Source property expects a class derived from UI-TableViewSource and that, in turn expects an array of strings, not a List<Book>. Easily fixed.

Create a new class BookListTableSource that derives from UITableViewSource and give it an array of strings. We need to pass in the array of strings (which we'll take care of in a moment) and then override RowsInSection (to tell it how many rows it has) and GetCell. These are handled pretty much the same for every application.

```
public class BookListTableSource : UITableViewSource

{
    string[] tableItems;
    string cellIdentifier = "TableCell";
    public BookListTableSource (string[] items)
    {
        tableItems = items;
    }
    public override int RowsInSection (UITableView tableview, int section)
    {
        return tableItems.Length;
    }
    public override UITableViewCell GetCell (UITableView tableView, MonoTouch.Foundation.NSIndexPath indexPath)
    {
        UITableViewCell cell = tableView.DequeueReusableCell (cellIdentifier);
        // if there are no cells to reuse, create a new one
        if (cell == null)
            cell = new UITableViewCell (UITableViewCellStyle.Default, cellIdentifier);
        cell.TextLabel.Text = tableItems [indexPath.Row];
        return cell;
    }
}
```

Really you can just copy and paste this code and then later modify it for your own variations.

As for converting the List<Book> to an array of strings, we'll do that in BookListController where we assign our new BookListTableSource to our BookListTableView's Source property in ViewDidLoad:

```
public override void ViewDidLoad ()
{
    base.ViewDidLoad ();
    BookListTableView.Source = new BookListTableSource (ConvertBooksToStringArray ());
}
```

The conversion happens in our private helper method, ConvertBooksToStringArray:

```
private string[] ConvertBooksToStringArray ()
{
    string[] output = new string[books.Count];
    int counter = 0;
    foreach (var book in books) {
        string entry = book.Name + " by " + book.Author + ". Purchased on " + book.Date-
Purchased;
        output [counter++] = entry;
    }
    return output;
}
```

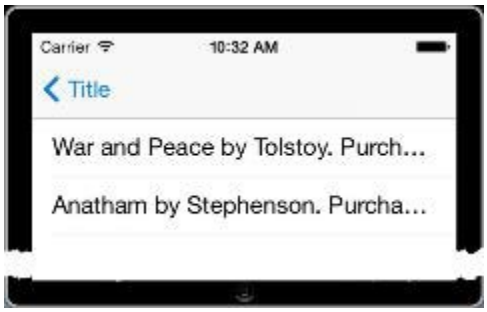
<whew!>. You are now ready to start adding books.

Run the application and add a couple books, remembering to click Add after each one. When you are ready, click Review and you should navigate to the new page which will display a list of the books you've picked,

It worked! (Always surprises me). Notice, however, that your text is too long to be displayed. In the next chapter we'll look at creating a custom cell view so that you can adjust the layout of each entry.

Storyboards and Custom Cells

In earlier chapters we navigated to a details page, but the display of the details left a lot to be desired. We had a single string of information and not enough room to display it in the UITableViewCell in the BookListTableView.



In this chapter, we'll take a look at customizing the UITableViewCell. We return to the same code, but we'll add a new class, BookCell, that will derive from UITableViewCell. This cell will be used to describe how we want each cell in the table to display its data.

Subclassing UITableViewCel

We start by creating three member variables of type UILabel (though we certainly could create other types of View objects as well):

```
public class BookCell : UITableViewCell

{
    UILabel BookName;
    UILabel AuthorName;
    UILabel PurchaseDate;
```

Next, a good deal of our work is done in the constructor, where we start by invoking the base constructor:

```
public BookCell (NSString cellID)
    : base (UITableViewCellStyle.Default, cellID)
{
```

Let's set the selection style and the background color. Might as well jazz up the colors some to make this really stand out:


```
SelectionStyle = UITableViewCellSelectionStyle.Gray;

ContentView.BackgroundColor = UIColor.FromRGB (218, 255,
127);
```

We're now ready to fill the three UILabels we'll be adding to the cell. For each we'll set a font, a text color and a background color. We can also set the text alignment and other properties at this point.

```
BookName = new UILabel () {
                Font = UIFont.FromName ("Cochin-BoldItalic", 22f):
                TextColor = UIColor.FromRGB (127, 51, 0):
                BackgroundColor = UIColor.Clear
            };
            AuthorName = new UILabel () {
                Font = UIFont.FromName ("AmericanTypewriter", 12f):
                TextColor = UIColor.FromRGB (38, 127, 0):
                BackgroundColor = UIColor.Clear
            };
            PurchaseDate = new UILabel () {
                Font = UIFont.FromName ("AmericanTypewriter", 12f):
                TextColor = UIColor.FromRGB (38, 127, 0):
                BackgroundColor = UIColor.Clear
            };
```

Finally, we add the three controls to the ContentView,

```
    ContentView.Add (BookName);
    ContentView.Add (AuthorName);
    ContentView.Add (PurchaseDate);
}
```

There are two additional methods to implement. We need an UpdateCell method to assign values to the Text properties of the lables, and we need to override LayoutSubviews to establish how we want the labels laid out in the cell. Let's look at UpdateCell first:

```
public void UpdateCell (string bookName, string authorName, string purchaseDate)

{
```

```
    BookName.Text = bookName;
    AuthorName.Text = authorName;
    PurchaseDate.Text = purchaseDate;
}
```

This is pretty straightforward. We've elected to pass in three strings. We could have passed in a Book object and extracted the strings from that, but I chose to do that wok in BookListTableSource's GetCell method, as you'll see shortly.

The override of LayoutSubviews sets the Frame property of each Label control to a Rectangle, passing in the x and y coordinates as well as the width and height:

```
public override void LayoutSubviews ()

{

    base.LayoutSubviews ();
    BookName.Frame = new RectangleF (0, 0, 200, 20);
    AuthorName.Frame = new RectangleF (0, 20, 100, 20);
    PurchaseDate.Frame = new RectangleF (200, 20, 100, 20);
}
```

Passing in Books

Returning to BookListController, we can eliminate ConvertBooksToStringArray, and pass an array of Books rather than an array of strings (remember that the member books is a List<Book>),

```
public override void ViewDidLoad ()

{

    base.ViewDidLoad ();
    BookListTableView.Source =
        new BookListTableSource (books.ToArray ());
}
```

Overriding GetCell

We now return to our BookListTableSource and override GetCell. Since BookCell is a UITableViewCell we can instantiate a BookCell where we previously instantiated its base class.

Note that tableItems[IndexPath.Row] is now a Book object, not a string, we must extrac the property we want for each label.

```
public override UITableViewCell GetCell (UITableView tableView, MonoTouch.Foundation.NSIn-
dexPath indexPath)

{
    BookCell cell = tableView.DequeueReusableCell              (cellIdentifier) as
BookCell;
    // if there are no cells to reuse, create a new one
    if (cell == null)
        cell = new BookCell (cellIdentifier);
    cell.UpdateCell (tableItems [indexPath.Row].Name:
        tableItems [indexPath.Row].Author:
        tableItems [indexPath.Row].                DatePurchased.ToShortDat-
eString ());
    return cell;
}
```

Of course, all the data we've been working with is in memory. In an upcoming chapter we'll take a look at data persistence.

Learning Xamarin - Building An App

I've set out on an adventure; not only to learn Xamarin inside out, but to build an app, in public. I don't know if I'll finish it, I don't know if it will work, and I don't know just how much risk I'm taking writing this as I go, but life is risk.

The APP

The concept for the app came from my older daughter, Robin, who has recently accompanied us to buy both a new refrigerator and a new dishwasher (oh, the joys of home ownership).

Each time we went shopping, and attempted to figure out what features were important to us, how important each feature was relative to the others, and how each potential choice stacked up with regard to those features. Then we'd run around trying to remember which item had the features we wanted most (think of price as a feature).

Robin's idea was to create an app that lets you put in arbitrary criteria for any item you want to buy, weight the various criteria, and evaluate contending models based on those criteria.

My idea was to build this app incrementally, and worry about the aesthetics of the layout last, when I could go to our crack designer, Matt and plead for help.

I decided to put the hard parts off, and start with data in memory, and to seed the View Model with data to get me started.

The Data Model

At least initially, I really only need two classes. The first is Item to represent the individual appliance (e.g., the Maytag 6300),

[Note all the appliances in this book are made up]

```
public class Item : INotifyPropertyChanged

{
    private string _name;
    public string Name {
        get{ return _name; }
        set {
            if (value != _name) {
                _name = value;
                NotifyPropertyChanged ();
            }
        }
    }
    private string _location;
    public string Location {
        get{ return _location; }
```

```
        set {
            if (value != _location) {
                _location = value;
                NotifyPropertyChanged ();
            }
        }
    }
    private ObservableCollection<Criterion> _criteria;
    public ObservableCollection<Criterion>  Criteria {
        get{ return _criteria; }
        set {
            if (value != _criteria) {
                _criteria = value;
                NotifyPropertyChanged ();
                ResetNotifications ();
            }
        }
    }
```

This simple class has three properties:

1. The name of the specific appliance
2. The location of that appliance in the store
3. A collection of criteria that we'll use to judge these appliances

The second class in the data model is of course, Criterion. Here we need the name of the criterion, a description to explain the particular rating system used for this criterion, and the rating and its relative importance.

```
public class Criterion : INotifyPropertyChanged

{
    private string _name;
    public string Name {
        get { return _name; }
        set {
            if (value != _name) {
                _name = value;
                NotifyPropertyChanged ();
            }
        }
    }
    private string _description;
    public string Description {
        get { return _description; }
```

```
        set {
            if (value != _description) {
                _description = value;
                NotifyPropertyChanged ();
            }
        }
    }
    private string _rating;
    public string Rating {
        get { return _rating; }
        set {
            if (value != _rating) {
                _rating = value;
                NotifyPropertyChanged ();
            }
        }
    }
    private string _importance;
    public string Importance {
        get { return _importance; }
        set {
            if (value != _importance) {
                _importance = value;
                NotifyPropertyChanged ();
            }
        }
    }


    public event PropertyChangedEventHandler PropertyChanged;
    private void NotifyPropertyChanged ([CallerMemberName] String propertyName =
“”)
    {
        if (PropertyChanged != null) {
            PropertyChanged (this, new PropertyChangedEventArgs (propertyName));
        }
    }
}
```

Notice that rating and importance are strings. This makes data binding to the view much easier, but makes me a little crazy. I may change this.

(Databinding in forms only works with strings. One way or another, at least for now, I’ll be doing some data conversions).

M-V-VM

I have created folders for my model (which contains the two classes shown above), my ViewModel and my Views. While iOS programming is traditionally MVC, the two patterns are very similar, and forms, which use XAML, seem to map better to MVVM. In my ViewModel, I just have starter data so that the application has something to display,

```
public class ItemsViewModel
{
    public ObservableCollection<Item> Items { get; set; }
    public ItemsViewModel ()
    {
        Items = new ObservableCollection<Item> ();
        Items.Add (new Item {
            Name = “Samsung 5200”:
            Location = “3rd Row, 2nd from end”:
            TotalRating = “96”:
            Criteria = new ObservableCollection<Criterion> () {
                new Criterion {
                    Name = “Noise”:
                    Description = “Higher is quieter”:
                    Rating = “4”:
                    Importance = “4”
                }:
                new Criterion {
                    Name = “Power Spray”:
                    Description = “Good for caseroles on bottom”:
                    Rating = “0”:
                    Importance = “3”
                }:
                new Criterion {
                    Name = “Cost”:
                    Description = “Higher number is cheaper”:
                    Rating = “3”:
                    Importance = “4”
                }:
                new Criterion {
                    Name = “Quality”:
                    Description = “Higher is better”:
                    Rating = “4”:
                    Importance = “5”
                }
            }
        });
        Items.Add (new Item {
            Name = “Maytag Premium”:
            Location = “2nd row, middle”:
```

```
                TotalRating = “102”:
                Criteria = new ObservableCollection<Criterion> () {
//...
```

Two things to notice. First, I’ve spared you the entire file as it is just more of the same. Second, each Item contains an ObservableCollection of Criterion called Criteria. This will get interesting when we preserve the Items to a Sqlite database (which does not, I’m afraid, support Foreign Keys terribly well).

Views

So far I’ve created three views

1. ItemPage displays the name and location of an item and its overall score
2. DetailsPage displays the criteria for a selected item
3. AddCriterionPage allows you to add new criteria for all the items

Item page also has a link to AddCriterionPage so that the user can add an arbitrary number of criteria by which they evaluate the appliances:

```
<?xml version=”1.0” encoding=”UTF-8”?>

<ContentPage xmlns=”http://xamarin.com/schemas/2014/forms” xmlns:x=”http://schemas.micro-
soft.com/winfx/2009/xaml” x:Class=”EvalUate.ItemPage” Title=”Dishwashers”>

    <ContentPage.Content>
<StackLayout Orientation=”Vertical”>
    <Button Text=”Add Criterion” Clicked=”OnAddCriterionClicked”/>
        <ListView x:Name=”list” ItemsSource=”{Binding Items}” ItemTapped=”OnItemSelect-
ed”>
            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <ViewCell.View>
                            <StackLayout Orientation=”Vertical” HorizontalOp-
tions=”StartAndExpand”>
                                <Label Text=”{Binding Name}” HorizontalOptions=”Fill-
AndExpand” />
                                <StackLayout Orientation=”Horizontal” HorizontalOp-
tions=”StartAndExpand”>
                                    <Label Text=”{Binding Location}” HorizontalOp-
tions=”FillAndExpand” Font=”10” />
                                    <Label Text=”Total Rating: “ HorizontalOp-
tions=”EndAndExpand” Font=”10” />
                                    <Label Text=”{Binding TotalRating}” HorizontalOp-
```

```
tions=”End” Font=”10” />
                                    </StackLayout>
                                </StackLayout>
                            </ViewCell.View>
                        </ViewCell>
                    </DataTemplate>
                </ListView.ItemTemplate>
            </ListView>
</StackLayout>
    </ContentPage.Content>
</ContentPage>
```

The key thing to notice here is that these forms are laid out with XAML, and so will be very familiar to anyone with Silverlight, WPF, Windows 8 or Windows Phone experience. See my introduction to Xamarin forms.

Key to the use of XAML in this and other forms, is DataBinding. Here we are binding to properties on the Item object.

Notice that above the ListView is a Button with a Clicked event. We see the implementation of the event handler in the code behind; that is, in ItemPage.xaml.cs,

```
 public void OnAddCriterionClicked (object o, EventArgs e)

{
    Navigation.PushAsync (new AddCriterionPage (vm.Items));
}
```

That code navigates to the AddCriterionPage page. Before we look at that, the ItemPage.xaml.cs file has an event handler for when an item is selected as well as a constructor that sets up the initial data binding context:

```
ItemsViewModel vm;

public ItemPage ()

{
    InitializeComponent ();
    vm = new ItemsViewModel ();
    this.BindingContext = vm;
}
public void OnItemSelected (object sender, ItemTappedEventArgs e)

{
    var item = e.Item as Item;
    if (item == null) {
        return;
```

```
    }
    Navigation.PushAsync (new DetailsPage (item));
    list.SelectedItem = null;
}
```

Adding Criteria

The AddCriterionPage is pretty simple; it asks the user to name the criterion and to give it a description:


```
<?xml version="1.0" encoding="UTF-8"?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/
forms" xmlns:x="http://schemas.microsoft.com/winfx/2009/
xaml" x:Class="EvalUate.AddCriterionPage">

    <ContentPage.Content>
        <StackLayout Orientation="Vertical" Horizon-
talOptions="StartAndExpand">
            <Label Text="Name" HorizontalOptions="-
FillAndExpand" />
            <Entry Text="{Binding Name}" WidthRe-
quest="400" HorizontalOptions="FillAndExpand" />
            <Label Text="Description" HorizontalOptions="FillAndExpand" />
            <Entry Text="{Binding Description}" WidthRequest="400" HorizontalOp-
tions="FillAndExpand" />
            <Button Text="Save" HorizontalOptions="CenterAndExpand" Clicked="OnSave-
NewCriterion" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

This page also uses data binding and events, both of which depend on the codebehind,

```
public partial class AddCriterionPage : ContentPage

{

    Criterion criterion;
    ObservableCollection<Item> items;
    public AddCriterionPage (ObservableCollection<Item> items)
    {
        InitializeComponent ();
        this.items = items;
        criterion = new Criterion ();
        this.BindingContext = criterion;
    }
```

This part of the code establishes the new Criterion object to which we'll be binding. What remains is to initialize the Rating and Importance values and then to add the new Criterion object to each item in the ViewModel's items collection (which was passed in to the constructor):

```
public void OnSaveNewCriterion (object o, EventArgs e)

{
    criterion.Rating = "0";
    criterion.Importance = "0";
    foreach (var item in items) {
        var newCriterion = new Criterion ();
        newCriterion.Name = criterion.Name;
        newCriterion.Description = criterion.Description;
        newCriterion.Rating = criterion.Rating;
        newCriterion.Importance = criterion.Importance;
        item.Criteria.Add (newCriterion);
        item.ResetNotifications ();
    }
    Navigation.PopAsync ();
}


Navigation.PopAsync ();
```

That final line ensures that after this work is done we return to the page that called us.

Details, Details

The third and remaining page (for now) is the DetailsPage, which displays the various criteria for the selected item.

The Details page lays the criteria out in a grid, using classic XAML, including attached properties such as Grid.Row and Grid.Column.

The indentation makes it almost impossible to read here, so I've broken it up, Here is the indentation,

```
<?xml version="1.0" encoding="UTF-8"?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms" xmlns:x="http://schemas.micro-
soft.com/winfx/2009/xaml" x:Class="EvalUate.DetailsPage" Title="{Binding Name}">

    <ContentPage.Content>
            <ScrollView>
                <ListView x:Name="CriteriaList" ItemsSource="{Binding Criteria}">
                    <ListView.ItemTemplate>
                        <DataTemplate>
```

```
                            <ViewCell>
                                <ViewCell.View>
                                </ViewCell.View>
                            </ViewCell>
                        </DataTemplate>
                    </ListView.ItemTemplate>
                </ListView>
            </ScrollView>
    </ContentPage.Content>
</ContentPage>
```

Here is the code that goes inside the ViewCell.View,


```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width=”100” />
        <ColumnDefinition Width=”100” />
        <ColumnDefinition Width=”100” />
    </Grid.ColumnDefinitions>
    <Label Text=”{Binding Name}” Grid.Row=”0” Grid.Col-
umn=”0” />
    <StackLayout Orientation=”Horizontal” Grid.Row=”0”
Grid.Column=”1”>
        <Label Text=”Rated: “ />
        <Entry Text=”{Binding Rating}” />
    </StackLayout>
    <StackLayout Orientation=”Horizontal” Grid.Row=”0” Grid.Column=”2”>
        <Label Text=”Value: “ />
        <Entry Text=”{Binding Importance}” />
    </StackLayout>
</Grid>
```

The code behind in this case, is trivial,

```
public partial class DetailsPage : ContentPage

{

    public DetailsPage (Item item)
    {
        InitializeComponent ();
        this.BindingContext = item;
    }
}
```

The Details page allows the user to fill in values both for the importance and the rating for each criterion.

The Total Score (shown on the Item page) is computed by doubling the Rating and multiplying by the Value, and then summing those products. (Feel free to change the algorithm!)

That's as far as I've gotten so far. You'll see the rest of the changes as this book progresses.

Xamarin Forms: Converting Integers

In the last chapter, I complained that Forms do not (yet) automatically convert strings to integers (or vice versa) and thus was "forced" to declare my integer values as strings. In this chapter, I work around that problem by implementing an IValueConverter.

In yesterday's code, for example, the Criterion's Rating and Importance were declared as strings.

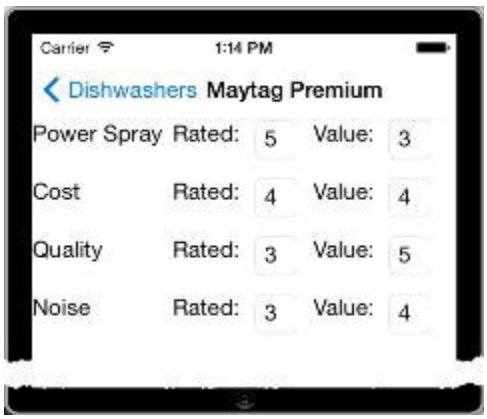
Today, we reset them to the semantically correct int:

```
private int _rating;
public int Rating {
    get { return _rating; }
    set {
        if (value != _rating) {
            _rating = value;
            NotifyPropertyChanged ();
        }
    }
}
```

We do the same with the Total in Item. Displaying the total in Item is not a problem, as we can use the StringFormat in the XAML to format the integer as a string. Here is an excerpt from ItemPage.xaml:

```
<Label Text="Total Rating: " HorizontalOptions="EndAndExpand" Font="10" />
<Label Text="{Binding TotalRating, StringFormat='{0}'}" HorizontalOptions="End" Font="10"
/>
```

Not so lucky with the Entry widget, however, which we use in DetailsPage.xaml. Here we need to create a class that implements IValueConverter. We create a new class, IntConverter.cs:




```
public class IntConverter : IValueConverter
{
    public object Convert (
                object value,
               Type targetType,
```

```
                object parameter,
                CultureInfo culture)
    {
        int theInt = (int)value;
        return theInt.ToString ();
    }
    public object ConvertBack (
                    object value,
                    Type targetType,
                     object parameter,
                     CultureInfo culture)
    {
        string strValue = value as string;
        if (string.IsNullOrEmpty (strValue))
            return 0;
        int resultInt;
        if (int.TryParse (strValue, out resultInt)) {
            return resultInt;
        }
        return 0;
    }
}
```

We can now incorporate this into our XAML by first declaring the appropriate namespace and then adding the converter as a resource:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<ContentPage xmlns=”http://xamarin.com/schemas/2014/forms”
xmlns:x=”http://schemas.microsoft.com/winfx/2009/xaml”
xmlns:local=”clr-namespace:EvalUate;assembly=EvalUate”
x:Class=”EvalUate.DetailsPage”
Title=”{Binding Name}”>
<ContentPage.Resources>
    <ResourceDictionary>
        <local:IntConverter x:Key=”intConverter”/>>
    </ResourceDictionary>
</ContentPage.Resources>
```

With that in place, we can use the valueConverter to convert the integer values to strings in the UI,

```
<Entry Text=”{Binding Path=Rating,
    Converter={StaticResource intConverter}}”
    Keyboard=”Numeric” />
```

We do the same, of course for Importance.

With that, (and a bit of fixing up the ViewModel) we have stored our values as integers (as we should) but displayed them as strings (as the Xamarin Form insists).

Extending the Forms Application

In this chapter we will extend the application in a couple of ways. First, it really bothered me to have "Add criterion" on the items page; it seemed much more reasonable to have it on the Details (criteria) page, and so we'll move it.

Second, currently we have no way of adding new items (e.g., a new Dishwasher) so we'll add that ability.

The process of making these changes will make evident that once you get the pattern, adding pages and navigating among them is pretty straightforward.

Moving Add Criterion

To move the "Add Criterion" button from the Item page to the Details page we just add a StackLayout within the ScrollView in the Details page, and add the button above the ListView:

```
<ContentPage.Content>
    <ScrollView>
        <StackLayout Orientation="Vertical">
            <Button Text="Add Criterion" Clicked="OnAddCriterionClicked" />
            <ListView x:Name="CriteriaList" ItemsSource="{Binding Criteria}">
```

We then implement OnAddCriteriaClicked in the DetailsPage.xaml.cs file. However, we'll need access to the ViewModel's items collection in the AddCriterion page. The Items page has a reference to the View-Model, but there is no reason to pass the entire vm to the Details page, we can just pass the collection of items.

```
public partial class DetailsPage : ContentPage

{
    ObservableCollection<Item> items;
    public DetailsPage (          Item item, ObservableCollection<Item> items)
    {
        InitializeComponent ();
        this.items = items;
        this.BindingContext = item;
    }
```

This requires that we modify the Item page to pass in the items collection:

```
public void OnItemSelected (object sender, ItemTappedEventArgs e)

{
    var item = e.Item as Item;
    if (item == null) {
        return;
    }
    Navigation.PushAsync (new DetailsPage (item, vm.Items));
    list.SelectedItem = null;
}
```

No changes are necessary in the AddCriterionPage – it doesn't care where it is called from and will "pop" back to whatever page invoked it.

Adding Items

Adding Items follows very much the same logic. First, we add a button to the Items page itself:

```
<?xml version="1.0" encoding="UTF-8"?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms" xmlns:x="http://schemas.micro-
soft.com/winfx/2009/xaml" x:Class="EvalUate.ItemPage" Title="Dishwashers">

    <ContentPage.Content>
        <StackLayout Orientation="Vertical">
                <Button Text="Add Item" Clicked="OnAddItemClicked" />
            <ListView x:Name="list" ItemsSource="{Binding Items}" ItemTapped="OnItem-
Selected">
```

Then we add the event handler inside ItemPage.xaml.cs,

```
public void OnAddItemClicked (object o, EventArgs e)

{
    Navigation.PushAsync (new AddItemPage (vm.Items));
}
```

Once again, we must pass in the Items collection so that we can add the new Item to it.

The UI for AddItemPage is very simple, we just need the name (Model?) of the new item and its location in the store:

```
<?xml version="1.0" encoding="UTF-8"?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms" xmlns:x="http://schemas.micro-
soft.com/winfx/2009/xaml" x:Class="EvalUate.AddItemPage">
    <ContentPage.Content>
        <StackLayout Orientation="Vertical" HorizontalOptions="StartAndExpand">
            <Label Text="Name" HorizontalOptions="FillAndExpand" />
            <Entry Text="{Binding Name}" WidthRequest="400" HorizontalOptions="Fill-
AndExpand" />
            <Label Text="Location" HorizontalOptions="FillAndExpand" />
            <Entry Text="{Binding Location}" WidthRequest="400" HorizontalOptions="-
FillAndExpand" />
            <Button Text="Save" HorizontalOptions="CenterAndExpand" Clicked="OnSaveNe-
wItem" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

The Save button invokes the OnSaveNewItem event handler in the code behind. Let's take apart the code-behind line by line. We begin by declaring member variables to hold both the new item and the collection of items that will be passed in. The constructor initializes them and sets the binding context,

```
public partial class AddItemPage : ContentPage

{
    Item item;
    ObservableCollection<Item> items;
    public AddItemPage (ObservableCollection<Item> items)
    {
        InitializeComponent ();
        this.items = items;
```

```
        item = new Item ();
        this.BindingContext = item;
    }
```

When the user clicks Save we want to initialize the new item and initialize its Criteria collection. The trick here is that we want to set the Name, Description and Importance of each of the Criteria to the already established values from other Items.

```
public void OnSaveNewItem (object o, EventArgs e)

{
    item.TotalRating = 0;
    item.Criteria = new ObservableCollection<Criterion> ();
    if (items.Count > 0) {
        var existingItem = items [0];
        foreach (var criterion in existingItem.Criteria) {
            var newCriterion = new Criterion ();
            newCriterion.Name = criterion.Name;
            newCriterion.Description = criterion.Description;
            newCriterion.Importance = criterion.Importance;
            newCriterion.Rating = 0;
            item.ResetNotifications ();
            item.Criteria.Add (newCriterion);
        }
    }
```

Once that is done, we can add the new item to the Items collection and pop back to the calling page:

```
    items.Add (item);
    Navigation.PopAsync ();
}
```

There is a great deal that is similar between adding an Item and adding a Criterion, and there is a great deal that is similar in the Pages we use to gather the information.

You can see this in action here.

Tinkering with our app

In the previous chapter, I modified the application that we've been working with from the start of this book. In this chapter, I'm going to tinker a bit more, though the really big fix is to add persistence. That, I'm afraid, will wait a little while longer.

What I want to do now is to ensure that the criteria are the same across items (and turn them into labels rather than values that can be filled in). And I want to get rid of the pre-created data and make sure that

my first item and first criterion work as expected.

To do this, I need to make a few modifications to the pages, but for this short chapter, I'm not changing the basic structure of the program.

Importance Of Criteria

You will remember that for any given item there are a set of criteria. That set is the same for all the items we're comparing, and each criterion has a rating (the value you assign to that criterion for that item) and an importance (how highly you value that criterion in general). It makes no sense for the importance to vary among the items, so we should set that when we create the new criterion.

Here's the XAML for our modified AddCriterionPage.xaml file:

```
<?xml version="1.0" encoding="UTF-8"?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:-
Class="EvalUate.AddCriterionPage" xmlns:local="clr-name-
space:EvalUate;assembly=EvalUate" >
    <ContentPage.Resources>
        <ResourceDictionary>
            <local:IntConverter x:Key="intConverter" />
    </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout Orientation="Vertical" HorizontalOp-
tions="StartAndExpand">
            <Label Text="Name" HorizontalOptions="FillAnd-
Expand" />
            <Entry Text="{Binding Name}" WidthRequest="300"
                  HorizontalOptions="Start" />
            <Label Text="Description" HorizontalOptions="-
FillAndExpand" />
            <Entry Text="{Binding Description}" WidthRequest="300"
                   HorizontalOptions="Start" />
            <Label Text="Importance" HorizontalOptions="FillAndExpand" />
            <Entry Text="{Binding Path=Importance:
                Converter={StaticResource intConverter}}"
                WidthRequest="50" HorizontalOptions="Start"
                Keyboard=»Numeric»/>
            <Button Text=»Save» HorizontalOptions=»CenterAndExpand»
                Clicked=»OnSaveNewCriterion» />
        </StackLayout>
    </ContentPage.Content>
```

```
</ContentPage>
```

With that in place, we can change the Entry for Importance on the Details page to be a label:

```
<StackLayout Orientation=»Horizontal» Grid.Row=»0» Grid.Column=»2»>
    <Label Text=»Value: « />
    <Label Text=»{Binding Path=Importance, Converter={StaticResource intConverter}}» />
</StackLayout>
```

We can now turn to the ItemsViewModel and just comment out the initialization of the items collection, thus giving us a blank page when we start up.

We can click on Add Item and add a new item (e.g., a Whirlpool X45) and a new location (e.g., 3rd row, 2nd from left).

That new item will not have a score, but we can click Add Criterion and add, e.g., power, or quiet, or color or whatever is important to us, and set the importance of each of the criteria.

We can then add additional items and they'll get the same criteria and the same importance, and we can fill in our rating and get a score for each.

Unfortunately, when we turn off the application and turn it back on, all the data is lost. So we really have to do something about that soon.

Learning Xamarin - Adding Persistence with SQLite

Virtually every interesting application deals with data, and very often you want that data to survive your session with the app, so that you can return to it next time. To accomplish this, you must write your data somewhere, and with mobile apps you have a number of options. For small amounts of data you may well decide to write to the file system (yes, there is one!) But for larger amounts of data, a database is very convenient, and happily SQLite comes preinstalled on iOS and Android.

Even better, nearly all the code you write for SQLite is shareable across platforms; the only platform-specific code is configuring the db connection and setting the location of the db file.

There is a difference, however, if you are running a PCL or a Shared Project application. For this discussion we'll look only at PCL and iOS.

Note that Xamarin.com has a very good article named Working With A Local Database, and they have an associated sample, but the two are (for now) out of sync with one another. I'm sure that they will fix that soon.

Adding the Database To The Project

To get started, we return to the project we've been building in the recent chapters in this series.

Begin by creating a new folder, Data, which will contain an Interface and our key database file. Let's start with the interface: ISQLite:

```
public interface ISQLite

{
    SQLiteConnection GetConnection ();
}
```

That was easy. The key file is EvalUateDatabase.cs. Let's go through it method by method.

We begin by declaring an instance of SQLiteConnection and an object that we can use for creating locks, since SQLite.net does not handle asynchronous programs very well.

```
using System;

using Xamarin.Forms;

using SQLite.Net;

using System.Collections.Generic;

using System.Linq;

namespace EvalUate

{
    public class EvalUateDatabase
    {
        SQLiteConnection database;
        static object locker = new object ();
```

Our first method is the constructor where we are passed in a SQLiteConnection object that we stash away in the database member. We then create the two tables we'll need, based on the classes in our Model:

```
public EvalUateDatabase (SQLiteConnection conn)

{
    database = conn;
    database.CreateTable<Item> ();
    database.CreateTable<Criterion> ();
```

```
}
```

Now, it is critical to note that SQLite.net does not support foreign keys, and it is none too happy with persisting complex objects. Thus, to make life easier, I'm going to add an attribute to my Item class instructing SQLite to ignore the Criteria collection:

```
private ObservableCollection<Criterion> _criteria;

[Ignore]

public ObservableCollection<Criterion>  Criteria {
    get{ return _criteria; }
    set {
        if (value != _criteria) {
            _criteria = value;
            NotifyPropertyChanged ();
            ResetNotifications ();
        }
    }
}
```

We'll have to recreate the Criteria collection when we obtain the Item from the database. To assist with that, we'll modify the Criterion object to know what Item it is associated with:

```
public class Criterion : INotifyPropertyChanged

{
    [PrimaryKey, AutoIncrement]
    public int ID { get; set; }
    public int ItemID { get; set; }
```

Notice also that the ID property of Criterion is marked both as a PrimaryKey and an AutoIncrement. You'll want to do the same in Item as well.

```
public class Item : INotifyPropertyChanged

{
    [PrimaryKey, AutoIncrement]
    public int ID { get; set; }
```

Let's return to EvalUateDatabase.cs

It has only two other methods, but they are the heart of the work we're doing right now.

Saving Items and Criteria To The Database

The first is used to save Items to the database, and with the item all its criteria. In each case, we check to see if the object as an ID of 0 (in which case it is new). If so we Insert it, otherwise we update it.

```
public int SaveItem (Item item)

{
    lock (locker) {
        foreach (Criterion c in item.Criteria) {
            c.ItemID = item.ID;
            if (c.ID != 0) {
                database.Update (c);
            } else {
                database.Insert (c);
            }
        }


        if (item.ID != 0) {
            database.Update (item);
            return item.ID;
        } else {
            return database.Insert (item);
        }
    }
}
```

You can see that we iterate through the criteria associated with the item and add each to the database, and then we add the item itself.

Note: The nature of the work flow means that when we're adding the criteria the item will already have an ID, otherwise it would make more sense to add the item first, get its ID and then use that to add the criteria.

Wiring It Up

None of this would do us much good if we didn't connect our UI to these methods. We do so first by turning to App.cs where we establish the connection and the database as static and thus available to the rest of the application.

```
public class App

{
    public static Page GetMainPage ()
    {
        var itemPage = new ItemPage ();
        return new NavigationPage (itemPage);
    }
    static SQLite.Net.SQLiteConnection conn;
    static EvalUateDatabase database;
    public static void SetDatabaseConnection (SQLite.Net.SQLiteConnection connection)
    {
        conn = connection;
        database = new EvalUateDatabase (conn);
    }
    public static EvalUateDatabase Database {
        get { return database; }
    }
}
```

iOS Specific

Everything we've done so far has been in shared code, but we do need to tell the application where to put the database, and that is different in iOS than in Android. The hack here is to create an empty database in your resources directory, and then override FinishedLaunching to copy the file into place:

```
public override bool FinishedLaunching (UIApplication app, NSDictionary options)

{
    Forms.Init ();
    window = new UIWindow (UIScreen.MainScreen.Bounds);
    var sqliteFilename = "EvalUateSQLite.db3";
    string documentsPath = Environment.GetFolderPath (Environment.SpecialFolder.Personal); // Documents folder
    string libraryPath = Path.Combine (documentsPath, "..", "Library"); // Library folder
    var path = Path.Combine (libraryPath, sqliteFilename);
    Console.WriteLine (path);
    if (!File.Exists (path)) {
        File.Copy (sqliteFilename, path);
    }
    var plat = new SQLite.Net.Platform.XamarinIOS.SQLitePlatformIOS ();
    var conn = new SQLite.Net.SQLiteConnection (plat, path);
```

```
    App.SetDatabaseConnection (conn);
    window.RootViewController = App.GetMainPage ().CreateViewController ();
    window.MakeKeyAndVisible ();

    return true;
}
```

Note, I want to be able to find the file and examine it using SQLiteBrowser, and so I temporarily changed the library path as follows:

```
string libraryPath = "/Users/jesseliberty/Projects";
```

Calling Into The Database

Returning to the shared code, we need to save items with their criteria, and we need to restore from the database on start up.

The first and most obvious place to save to the database is in OnSaveNewItem in AddItemPage.xaml.cs:

```
public void OnSaveNewItem (object o, EventArgs e)

{
    item.TotalRating = 0;
    item.Criteria = new ObservableCollection<Criterion> ();
    if (items.Count > 0) {
        var existingItem = items [0];
        if (existingItem != null && existingItem.Criteria.Count > 0) {
            foreach (var criterion in existingItem.Criteria) {
                var newCriterion = new Criterion ();
                newCriterion.Name = criterion.Name;
                newCriterion.Description = criterion.Description;
                newCriterion.Importance = criterion.Importance;
                newCriterion.Rating = 0;
                item.ResetNotifications ();
                item.Criteria.Add (newCriterion);
            }
        }
    }
    items.Add (item);
    App.Database.SaveItem (item);
    Navigation.PopAsync ();
}
```

Notice that this is really just a one line change. Next, we modify AddCriterionPage. You may remember that we pass all the items into this page and add the new criterion to each item. We modify that just slightly to save the item and reset its notifications on adding the criteria:

```
item.Criteria.Add (newCriterion);

App.Database.SaveItem (item);

item.ResetNotifications ();
```

Interestingly, we need to save the item again when we finish on the DetailsPage. After all, we've updated the values for that item's criteria.

```
public void OnSaveNewRatings (object o, EventArgs e)

{
    App.Database.SaveItem (currentItem);
    currentItem.ResetNotifications ();
    Navigation.PopAsync ();
}
```

Restoring Items

When we restart the application we want to go to the database to get the items and criteria that were previously saved, which we then stash away in the ViewModel:

```
public partial class ItemPage : ContentPage

{
    ItemsViewModel vm;
    public ItemPage ()
    {
        InitializeComponent ();
        var items = App.Database.GetItems ();
        vm = new ItemsViewModel ();
        vm.Items = new System.Collections.ObjectModel.ObservableCollection<Item>
(items);
        this.BindingContext = vm;
    }
```

That restores the entire object graph. It does so by invoking the third and final method in EvalUateDatabase.cs: GetItems(). Its job is to get all the items and for each item, all its criteria. Here we cheat. We look in the Criteria table for those criteria which have the current Item's ID – a poor man's foreign key.

```
public IEnumerable<Item> GetItems ()

{
    List<Item> items;
    lock (locker) {
        items = (from i in database.Table<Item> ()
                 select i).ToList ();

        foreach (Item i in items) {
            i.Criteria =
                newSystem.Collections.ObjectModel.ObservableCollection<Criterion>
                ((from c in database.Table<Criterion> ()
                  where c.ItemID == i.ID
                  select c).ToList ());
        }
        return items;
    }
}
```

Notice that we are converting the list of criteria to an ObservableCollection<Criterion> as that is what the Item holds, so that we're notified when the list changes. The data shows that each Criterion tracks not only its own data but which Item it belongs to.

Note that I've intentionally denormalized this database to simplify coding in the absence of foreign key constraints.

Not Rocket Science

I don't think anyone would ever accuse Xamarin of making adding persistence to a database easy, but it certainly isn't terribly difficult and can be done incrementally. More important, once you've done it a couple times, it becomes reasonably routine.

Learning Xamarin - Extending the App & DB Tables

The application that we've been working with so far is almost done.

Now we'll add a new first page to allow the user to track more than one appliance type. This will require minor changes throughout the application and a change to the database.

The key concern would be that we already have data in the database and now I not only want to add a table, but I want to add a property to one of the existing (and populated!) tables.

It turns out that SQLite.net makes this a total non-issue as it handles the update perfectly with no effort on my part.

Updating the Data

The goal is to be able to have different appliances that you are tracking. Each appliance (e.g., dishwasher, refrigerator) will have multiple items (e.g., different models) and each individual item will have details (e.g., how it rates on the various criteria).

The first significant change is to the definition of Item. I'm going to add an ApplianceId property so that I can find all the items associated with that appliance:

```
public class Item : INotifyPropertyChanged
{
    [PrimaryKey, AutoIncrement]
    public int ID { get; set; }
    public int ApplianceId { get; set; }
```

SQLite will see the change and update the table accordingly,

We can then use that column when displaying the items to find only the items associated with the selected appliance.

We are, of course, going to have to add appliances, and that implies two new pages: a new first page that lists the appliances, and a page to add new appliances.

And all of that implies that we need a new class to represent the appliance.

Appliances

The appliance class is dead simple:

```
    public class Appliance : INotifyPropertyChanged
    {
        [PrimaryKey, AutoIncrement]
        public int ID { get; set; }
        private string _name;
        public string Name {
            get { return _name; }
            set {
                if (value != _name) {
                    _name = value;
                    NotifyPropertyChanged ();
                }
            }
        }
//...
```

The class is just an ID and a Name property and the code (elided) to implement INotifyPropertyChanged.

AddAppliancePage.xaml is very similar to AddCritrion or AddItem:

```
<?xml version=”1.0” encoding=”UTF-8”?>

<ContentPage xmlns=”http://xamarin.com/schemas/2014/forms” xmlns:x=”http://schemas.micro-
soft.com/winfx/2009/xaml” x:Class=”EvalUate.AddAppliancePage”>

    <ContentPage.Content>
        <StackLayout Orientation=”Vertical” HorizontalOptions=”StartAndExpand”>
            <Label Text=”Name” HorizontalOptions=”FillAndExpand” />
            <Entry Text=”{Binding Name}” WidthRequest=”400” HorizontalOptions=”FillAnd-
Expand” />
            <Button Text=”Save” HorizontalOptions=”CenterAndExpand” Clicked=”OnSaveNe-
wItem” />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

The code behind is nearly identical to the related classes:

```
public partial class AddAppliancePage : ContentPage

{

    Appliance appliance;
    ObservableCollection<Appliance> appliances;
    public AddAppliancePage (ObservableCollection<Appliance> appliances)
    {
        InitializeComponent ();
        this.appliances = appliances;
        appliance = new Appliance ();
        this.BindingContext = appliance;
    }
    public void OnSaveNewItem (object o, EventArgs e)
    {
        appliances.Add (appliance);
        App.Database.SaveAppliance (appliance);
        Navigation.PopAsync ();
    }
}
```

The new first page, Appliance, looks a lot like the Item page:

```
<?xml version=”1.0” encoding=”UTF-8”?>

<ContentPage xmlns=”http://xamarin.com/schemas/2014/forms” xmlns:x=”http://schemas.micro-
soft.com/winfx/2009/xaml” x:Class=”EvalUate.AppliancePage” Title=”EvalUate”>

    <ContentPage.Content>
        <StackLayout Orientation=”Vertical”>
```

```
                <Button Text="Add Appliance" Clicked="OnAddApplianceClicked" />
            <ListView x:Name="list" ItemsSource="{Binding Appliances}" ItemTapped="On-
ApplianceSelected">
                <ListView.ItemTemplate>
                    <DataTemplate>
                        <ViewCell>
                            <ViewCell.View>
                                <StackLayout Orientation="Vertical" HorizontalOp-
tions="StartAndExpand">
                                    <Label Text="{Binding Name}" HorizontalOptions="-
FillAndExpand" />
                                </StackLayout>
                            </ViewCell.View>
                        </ViewCell>
                    </DataTemplate>
                </ListView.ItemTemplate>
            </ListView>
        </StackLayout>
    </ContentPage.Content></ContentPage>
```

And its code behind follows the same pattern as Items did:

```
public partial class AppliancePage : ContentPage

{
    ItemsViewModel vm;
    public AppliancePage ()
    {
        InitializeComponent ();
        var appliances = App.Database.GetAppliances ();
        vm = new ItemsViewModel ();
        vm.Appliances = new System.Collections.ObjectModel.ObservableCollec-
tion<Appliance> (appliances);
        this.BindingContext = vm;
    }
    public void OnApplianceSelected (object sender, ItemTappedEventArgs e)
    {
        var appliance = e.Item as Appliance;
        if (appliance == null) {
            return;
        }
        Navigation.PushAsync (new ItemPage (appliance));
        list.SelectedItem = null;
    }
```

```
    public void OnAddApplianceClicked (object o, EventArgs e)
    {
        Navigation.PushAsync (new AddAppliancePage (vm.Appliances));
    }
}
```

The key thing to notice here is that we pass the selected appliance to the ItemPage so that it can make the connection.

Notice also that we call GetAppliances to get the appliances list from the database, and that we keep that list in the ViewModel along with the list of items.

GetAppliances is a stripped down copy of GetItems, and SaveAppliances is very similar to SaveItems:

```
public class EvalUateDatabase

{
    SQLiteConnection database;
    static object locker = new object ();
    public EvalUateDatabase (SQLiteConnection conn)
    {
        database = conn;
        database.CreateTable<Item> ();
        database.CreateTable<Criterion> ();
        database.CreateTable<Appliance> ();
    }


    public IEnumerable<Appliance> GetAppliances ()
    {
        List<Appliance> appliances;
        lock (locker) {
            appliances = (from i in database.Table<Appliance> ()
                          select i).ToList ();
        }
        return appliances;
    }
    public int SaveAppliance (Appliance appliance)
    {
        lock (locker) {
            if (appliance.ID != 0) {
                database.Update (appliance);
                return appliance.ID;
            } else {
                return database.Insert (appliance);
            }
        }
    }
}
```

Don't forget to call CreateTable for the ApplianceTable in your constructor.

GetItems actually has to change slightly, as described above:

```
public IEnumerable<Item> GetItems (int applianceId)

{
    List<Item> items;
    lock (locker) {
        items = (from i in database.Table<Item> ()
                 where i.ApplianceId == applianceId
                 select i).ToList ();
```

Patterns

One of the nice things about extending the application in this way is that you can really see that there is a pattern in how we deal with adding, saving and retrieving data.

Notice also that *all the modifications happened in the common code; nothing specific to iOS had to change at all.*

Learning Xamarin - Adding Tabbed Pages

Until now, our application has used simple "push" navigation, creating a stack that the user can pop out of to return to the calling screen.

It would be convenient, however, to add tabs to the opening screen where the user can access the help file and the license information.

In the figure to the right, you can see that there are three tabs, and the currently selected tab is "help."

To accomplish this, we'll add three new pages:

- MainPage will hold the tabs
- HelpPage will hold the help text
- LicensePage will hold the license information

All the magic for tabs, such as it is, is in MainPage.xaml

Here is the XAML for MainPage. Notice that the <ContentPage> tag has been replaced with a <TabbedPage> tag, and <ContentPage.Content> is replaced by <TabbedPage.Children>:

```
<?xml version="1.0" encoding="UTF-8"?>

<TabbedPage

xmlns="http://xamarin.com/schemas/2014/forms"

xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"

xmlns:me="clr-namespace:EvalUate;assembly=EvalUate"

x:Class="EvalUate.MainPage"

Title="EvalUate">

    <TabbedPage.Children>
        <me:AppliancePage />
        <me:HelpPage />
        <me:LicensePage />
    </TabbedPage.Children>
</TabbedPage>
```

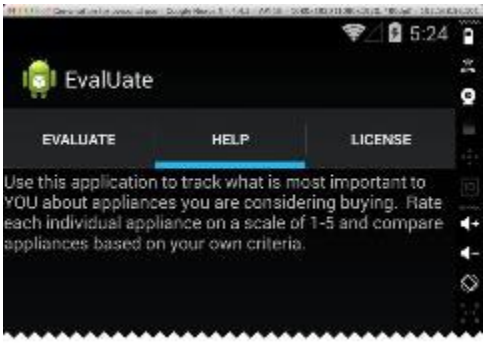
The only thing to know about the code behind is that MainPage must inherit from TabbedPage:

```
public partial class MainPage : TabbedPage

{

    public MainPage ()
    {
        InitializeComponent ();
    }
}
```

HelpPage and LicensePage are normal pages, just like all our others. You navigate among the (opening) AppliancePage, the HelpPage and the LicensePage using the tabs, but once you navigate from AppliancePage to ItemPage the tabs are no longer available.

Android Too

What is particularly nice is that since we're using Forms, there is zero additional work to get this to work on Android, where the tabs are on the top (that is, the platform-specific look and feel is automatic).



Xamarin Forms and HTTP

Let's say we're building an application, and we want to list the titles and publication dates from an RSS feed.

We can do this using a ListView, but we have to get the feed, and to do that we need a library; specifically the NuGet package Microsoft.Http.Client Libraries.

To create our application, we'll start with a new Forms application by firing up Xamarin Studio and selecting New Solution -> C# -> Mobile Apps -> Blank App (Xamarin.Forms.Portable).

Let's add structure to the app by adding these folders:

- Models
- Services
- View Models
- Views

Creating the Model

Our Model is very simple, just one class: BlogEntry:

```
public class BlogEntry

{
    public string Title { get; set; }
    public string Link { get; set; }
    public string Description { get; set; }
    public string PubDate { get; set; }
}
```

Of these properties, for now we'll only be concerned with Title and PubDate.

Creating the View Model

The View Model (VM) will mediate between the model and the view. Again, ours is very simple. Create a file named BlogsViewModel.

```
public class BlogsViewModel

{
    public ObservableCollection<BlogEntry> Blogs { get; set; }
}
```

Creating the View

Right-click on the Views folder and choose Add->New File -> Forms -> Forms Content Page XAML. Name the new page BlogsPage.

In the XAML we'll add a ListView and modify the Cell to display both the Title and the PubDate:

```
<?xml version="1.0" encoding="UTF-8"?>

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms" xmlns:x="http://schemas.micro-
soft.com/winfx/2009/xaml" x:Class="Falafel2GoV2.BlogsPage">

    <ContentPage.Content>
        <StackLayout Orientation="Vertical" HorizontalOptions="StartAndExpand">
            <ListView x:Name="list" ItemsSource="{Binding Blogs}">
                <ListView.ItemTemplate>
                    <DataTemplate>
                        <ViewCell>
                            <ViewCell.View>
                                <StackLayout Orientation="Vertical" HorizontalOp-
tions="StartAndExpand">
<!--                                <Label Text="{Binding Link}" />-->

                                    <Label Text="{Binding Title}" Font="12" />
                                    <Label Text="{Binding PubDate}" Font="10" />
                                </StackLayout>
                            </ViewCell.View>
                        </ViewCell>
                    </DataTemplate>
                </ListView.ItemTemplate>
            </ListView>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

The list itself binds to Blogs; the collection declared in the View Model. We'll make the connection to the VM as the binding context in the code behind. Similarly, the two Labels bind their Text properties to properties on each BlogEntry object in the list.

Services

The code behind for the view depends on our Services files. There are two files. The first, Urls.cs is pretty simple:

```
public static class Urls

{

    public enum UrlType
    {
```

```
        Blogs:
        Tweets:
        Website:
        Videos}
    ;
    public static string GetUrl (UrlType urlToRetrieve)
    {
        switch (urlToRetrieve) {
        case UrlType.Blogs:
            return “http://blog.falafel.com/Feeds/all-blogs”;
        default:
            return “”;
        }
    }
}
```

For now, the only URL we care about is the RSS feed for the Falafel blogs. Later we can add other URLs for other types of data, as hinted at by the enum.

The second, and more important class, is the DataLoader class:

```
public class DataLoader

{
    public async Task<List<BlogEntry>> GetRecentBlogs ()
    {
        using (var httpClient = new HttpClient ()) {
            Stream data =
                await httpClient.GetStreamAsync (
                    Urls.GetUrl (Urls.UrlType.Blogs));
            List<BlogEntry> blogs = await Task.Run (() =>
                GetBlogList (data));
            return blogs;
        }
    }
    private List<BlogEntry> GetBlogList (Stream blogRSSData)
    {
        IEnumerable<XElement> items = XElement.Load (
            blogRSSData).Descendants (“item”);
        var blogList = (from blog in items
                        select new BlogEntry () {
            Link = blog.Element (“link”).Value:
            Title = blog.Element (“title”).Value:
            Description = blog.Element (“description”).Value:
```

```
            PubDate = blog.Element ("pubDate").Value
        }).ToList<BlogEntry> ();
        return blogList;
    }
}
```

You can see that this consists of a key public method, GetRecentBlogs and a helper method, GetBlogList. The first uses the HttpClient library referred to earlier. To add this to your project, right click on the project and select Add -> Add Packages and search for Microsoft.Http.Client. Check the Microsoft.Http.Client libraries and click Add Package. Hey! Presto! It is added to your project, along with the appropriate references.

Notice that this runs asynchronously. We'll want to remember that in a moment when we return to the view to add the call to this method.

Back in BlogPage.xaml.cs you'll create a constructor and a private method GetBlogs. The constructor calls GetBlogs and GetBlogs calls the loader asynchronously.

```
public partial class BlogsPage : ContentPage

{
    private BlogsViewModel vm;
    public BlogsPage ()
    {
        InitializeComponent ();
        vm = new BlogsViewModel ();
        GetBlogs ();
    }
    private async void GetBlogs ()
    {
        var loader = new DataLoader ();
        List<BlogEntry> recentBlogs = await loader.GetRecentBlogs ();
        vm.Blogs = new System.Collections.ObjectModel.ObservableCollection<BlogEntry>
(recentBlogs);
        this.BindingContext = vm;
    }
}
```

That's it, your page will come up and a few moments later it will populate with the titles and the publication date.

There are a few essential things to take care of, such as showing an indeterminate progress indicator while loading the data and adding links to the blog titles.

Xamarin Forms: Web Viewer

In the last chapter we looked at using HTTP with Web Forms and creating a list of blog posts from an RSS feed. Now we'll add the ability to tap on one entry in the list and see that blog post in a viewer on the phone.

The image shown here, incidentally, is an exact reflection of what is running on my iPhone thanks to a very handy utility named Reflector. You run Reflector and then use AirPlay on the phone to mirror what is displayed on your phone onto the screen. This addition to our program is shockingly easy. Return to the source from before, and add a new Form page, BlogViewerPage.

Unlike previously, we'll create the entire page in code this time, rather than in XAML. You can ignore BlogViewrPage.xaml (just leave it as is) and open BlogViewrPage.xaml.cs. Here we'll add all our new code in the constructor.

After initializing the components, we create a label to act as a header for the page:

```
public BlogViewerPage (BlogEntry blogEntry)

{
    InitializeComponent ();
    Label header = new Label {
        Text = blogEntry.Title:
        Font = Font.BoldSystemFontOfSize (30):
        HorizontalOptions = LayoutOptions.Center
    };
```

We then add a web view, setting the web view's source to the link property of the BlogEntry that we passed into the constructor:

```
WebView webView = new WebView {
    Source = new UrlWebViewSource {
        Url = blogEntry.Link:
    }:
    VerticalOptions = LayoutOptions.FillAndExpand
};
```

Now all that we have to do is to add those two items to a new StackLayout and, while we're at it, we'll set some padding on the page:

```
this.Padding = new Thickness (10, Device.OnPlatform (20, 0, 0), 10, 5);

// Build the page.
```

```
this.Content = new StackLayout {
    Children = {
        header:
        webView
    }
};
```

Wiring It Up

Our final task is to write the code that responds to the tap on the RSS entry and passes the BlogEntry object to this new page. Return to BlogPage.xaml and add an event in the XAML:

```
<ListView x:Name=”list” ItemsSource=”{Binding Blogs}” ItemTapped=”OnBlogSelected”>
```

Now open BlogPage.xaml.cs and add the event handler, navigating to the new page and passing in the selected BlogEntry.

```
private void OnBlogSelected (object o, ItemTappedEventArgs e)

{
    var blogEntry = e.Item as BlogEntry;
    if (blogEntry != null) {
        Navigation.PushAsync (                new BlogViewerPage (blogEntry));
    }
}
```

That’s all it takes, now when you tap on an RSS entry, the Blog entry is shown (as you can see in the figure at the start of this chapter).

Closing

I hope you have found this eBook helpful as you pursue your own applications in Xamarin. It is a truly remarkable platform, and only getting better. If you need help with such an application, let me know. Falafel is a great resource to turn to for any mobile development needs. Of course, if you need a little more help on the side, there is always Falafel University! Check out the next page for more details.

Falafel University

Yearlong access to a live online learning experience for developers, testers, designers and web managers. Take each class as many times as you need. With each class comes all the materials you need to be successful at no extra charge, and we're releasing new material all the time.

Learn more now

or, call us at 831-462-0457