Xamarin Essentials

Learn how to efficiently develop Android and iOS apps for deployment using the Xamarin platform

Mark Reynolds
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 6 "The Sharing Game"
- A synopsis of the book’s content
- More information on Xamarin Essentials

About the Author

Mark Reynolds is a software enthusiast who has worked in the industry for nearly 30 years. He began his career with Electronic Data Systems, building and supporting systems for the manufacturing sector. Over the years, Mark has worked with start-ups and Fortune 100 companies across a diverse set of industries. In 1993, he started a consulting practice that focused on delivering training and mentoring services in the areas of software architecture, design, and implementation. With the rise of mobile computing, he has focused his practice on designing, developing, and delivering mobile software solutions.

Mark recently published his first book, Xamarin Mobile Application Development for Android, Packt Publishing. His private consulting practice is based in Allen, TX, USA, where he resides with his wife and son.

I would like to thank my mother, Charlene Reynolds, who I lost this year. She was a great mother, wife, sister, cousin, aunt, and friend, and a great inspiration to everyone she came in contact with. We'll miss her, but we know in her new home; she has overcome the illness she battled here, and we rejoice in that.
Mobile applications have revolutionized the way we communicate and interact with each other, and their development is now beginning to reach a certain level of maturity. To support the development of mobile apps, there are now many tools and environments available.

Xamarin is a toolset that has seen increasing success in recent years and is gaining more and more interest, particularly from development shops that have a significant investment in .NET and C# resources. Xamarin wraps each platform's native APIs with a C# wrapper, allowing the developer to interact with the environment in essentially the same way as any native developer would. As Xamarin apps are developed in C#, a whole new possibility of sharing code across platforms comes into play with all the associated benefits and challenges.

As companies look to adopt Xamarin, new Xamarin developers will be required; where do they come from? In many cases, there will be existing seasoned mobile developers who are already familiar with Android and iOS development.

That's where this book comes in; the idea being to provide a quick path for developers already familiar with Android and/or iOS development so they can get up to speed with Xamarin development. To that end, this book does not focus on the basics of developing Android and iOS apps; rather, we focus on teaching experienced Android and iOS developers how to develop apps using Mono, C#, and the Xamarin suite of tools. Specifically, we focus on the following topics:

- Architecture: This explains how the Xamarin products allow the use of Mono and C# to develop Android and iOS apps
- Tools: This describes the tools provided to support the development of applications
- Code sharing: This explains the types of code that can be shared between Android and iOS apps and the issues that might arise
- Distribution: This explains the special considerations that should be made when distributing Xamarin.Android and Xamarin.iOS apps

It should be noted that sample apps and code snippets are provided where appropriate.

When I first started using C# to develop iOS apps, it just felt a little strange. I was no fan of Objective-C, but when did C# become the cross-platform tool of choice? I always had a lot of respect for what the Mono team accomplished, but I generally had the view that Microsoft would eventually prohibit C# and .NET from being terribly successful on any platform that they did not own. Being a Star Wars fan, and somewhat of a geek, I was reminded of a conversation from Episode III. If you recall a certain scene between Anakin and Palpatine, where Anakin realizes Palpatine knew the dark side of the force; just replace the dark side of the force with Xamarin and you get Palpatine turning to you.
saying: "Xamarin is a pathway to many abilities some consider to be unnatural." That's pretty much the feeling I had; was I selling out to learn a cross-platform set of technologies that would eventually completely tie me to Windows?

Two years later, I feel fairly comfortable answering that question as no! Obviously, we work in a dynamic industry and things can change in an instant, but the technology world is in a different place than it was 10 years ago, and cross-platform C# and .NET seem to play in Microsoft's favor now. So, the strange feeling has been diminished with success, and seeing how the relationship between Microsoft and Xamarin has only gone from strength to strength, I am encouraged.

If you are coming from an Objective-C or Java background, you will likely have the same feelings from time to time, but if you give the tools a chance I think you will be amazed.

I hope that you find this book a valuable resource on your path to becoming a productive mobile application developer with the Xamarin suite of products.

What This Book Covers

Chapter 1, Xamarin and Mono – a Pathway to the Unnatural, provides an overview of the Mono project and the suite of Mono-based commercial products offered by Xamarin.

Chapter 2, Demystifying Xamarin.iOS, describes how Mono and the iOS platform coexist and allow developers to build iOS apps using C#.

Chapter 3, Demystifying Xamarin.Android, describes how Mono and the Android platform coexist and allow developers to build Android apps using C#.

Chapter 4, Developing Your First iOS App with Xamarin.iOS, walks you through the process of creating, compiling, running, and debugging a simple iOS app.

Chapter 5, Developing Your First Android App with Xamarin.Android, walks you through the process of creating, compiling, running, and debugging a simple Android app.

Chapter 6, The Sharing Game, presents various approaches of sharing code between Xamarin.iOS and Xamarin.Android apps.

Chapter 7, Sharing with MvvmCross, walks you through the use of the Xamarin.Mobile app, which provides a cross-platform API to access location services, contacts, and the device camera.

Chapter 8, Sharing with Xamarin.Forms, walks you through the basics of using the MvvmCross framework to increase code reuse between platforms.

Chapter 9, Preparing Xamarin.iOS Apps for Distribution, discusses various methods of distributing iOS apps, and walks you through the process of preparing a Xamarin.iOS app for distribution.
Chapter 10, Preparing Xamarin.Android Apps for Distribution, discusses various methods of distributing Android apps, and walks you through the process of preparing a Xamarin.Android app for distribution.
In this chapter, we will discuss one of the most interesting and important aspects of developing with Xamarin: cross-platform code sharing. We will cover the following topics:

- The file linking technique
- Portable Class Libraries
- The pros and cons of each approach

Sharing and reuse

One of the advantages of using Xamarin and C# is the ability to share code across your mobile apps as well as other .NET solutions. The reuse of code can provide significant productivity and reliability advantages as well as reduce many of the long-term maintenance headaches that come with long-lived apps. That's great, but anyone who has been involved in software development for a long period of time understands that reuse is not free and not simple to achieve.

There are practical aspects of reuse; the question is, "Physically, how do I package my code for reuse?" For this, we can use one of the following three methods:

- Share source code that can be compiled into multiple projects
- Share **Dynamic-Link Library (DLL)** that can be referenced by multiple projects
- Share code as a service that can be accessed remotely by multiple clients
There are also more strategic aspects; again the question arises, "How can I organize my code so that I can reuse more of it?" To solve this problem, we have the following options:

- Create a layered approach so that data access logic and business validation is separated out of the user interface logic
- Utilize interfaces and frameworks that abstract platform-specific services away from the reusable layers

In this chapter, we will touch on both these aspects of reuse, but primarily focus on the practical side of reuse. Specifically, we will present two different approaches to bundle up the code for reuse.

So, what parts of our code should we try and reuse? In the work we have done on the NationalParks apps so far, one obvious set of code stands out for reuse: the persistence code, which is the logic that loads parks from a JSON file and saves them back to the same file. In Chapter 5, Developing Your First Android App with Xamarin.Android, we moved towards a reusable solution by creating the NationalParkData singleton. In this chapter, we will demonstrate two different methods for sharing the NationalParkData singleton across both our projects as well as other .NET projects that might need it.

**Old school source file linking**

File linking refers to a technique where source code files are linked or referenced by a Xamarin project and are compiled when a build is run on the project along with the rest of the source code in the project. When using file linking, a separate DLL is not created for the files you are sharing, rather the code is compiled into the same DLL produced for the project that the file is linked to; in our case, either NationalParks.iOS.dll or NationalParks.Droid.dll.

**Creating a shared library project**

We will start by creating a new Library project to house the reusable code. To create a Library project, perform the following steps:

1. Add a new library project with the name NationalParks.Data to the NationalParks solution. You can find the Library project template in the New Project dialog box under C# | Library, as shown in the following screenshot:
2. Remove MyClass.cs from the new project. When removing the file, selecting **Delete** will remove the file from being referenced by the project and delete the underlying file from the filesystem.

3. Set the **Target Framework** option to **Mono/.NET 4.5** in the **Project Options** dialog box under **Build** | **General**.


5. Open NationalPark.cs and NationalParkData.cs and change the namespace to NationalParks.Data.

6. Add a public string **DataDir** property to NationalParkData and use it in the **GetFilename()** method, as follows:

```csharp
public string DataDir { get; set; }

protected string GetFilename()
{
    return Path.Combine (DataDir, "NationalParks.json");
}
```
7. Move the logic to load the parks data from the constructor to a new method named `Load()`, as shown in the following code snippet:

```csharp
public void Load()
{
    if (File.Exists (GetFilename())) {
        string serializedParks =
            File.ReadAllText (GetFilename());
        _parks = JsonConvert.DeserializeObject
            <List<NationalPark>> (serializedParks);
    }
    else
        _parks = new List<NationalPark> ();
}
```

8. Compile `NationalParks.Data`. You will receive compile errors due to unresolved references to Json.NET. Unfortunately, we cannot simply add a reference to the component version of Json.NET that we previously downloaded from the Xamarin component store because this version is built to be used with the Xamarin.iOS and Xamarin.Android profiles and is not binary compatible with Mono/.NET 4.5 library projects.

9. Add the Json.NET library to the project using NuGet. Select `NationalParks.Data`, right-click on it, and navigate to **Add** | **Add Packages**. Enter Json.NET in the search field, check the Json.NET entry in the list, and select **Add Packages**. The following screenshot shows the **Add Packages** dialog box:

![Add Packages](image-url)
10. Compile `NationalParks.Data`; you should receive no compile errors this time.

**Updating `NationalParks.Droid` to use shared files**

Now that we have the `NationalParksData` singleton in a separate project, we are now ready to reuse it.

To update `NationalParks.Droid` in order to use the shared solution, perform the following steps:

1. Select `NationalPark.cs` and `NationalParksData.cs` in the Solution pad, right-click on it, select Remove, and then select Delete. This will remove the selected files from the project and physically delete them from the project folder.

2. In `NationalParks.Droid`, add a folder named `NationalParks.Data`. This folder will not contain any files, but will simply be used within the project structure to organize links to the shared files.

3. Select the `NationalParks.Data` folder, right-click on it, and navigate to Add | Add Files to add the existing files to the project.

4. In the Add Files dialog box, navigate to the `NationalParks.Data` project folder, select `NationalPark.cs` and `NationalParkData.cs`, and click on Open.

5. In the Add File to Folder dialog box, select Add a link to file, check the Use the same action for all selected files option, and click on OK. Expand the `NationalParks.Data` folder to see that two file links are added. The following screenshot shows the Add File to Folder dialog box:
6. Add a using clause to the NationalParks.Data namespace and remove any using directives for Newtonsoft.Json in MainActivity, DetailActivity, EditActivity, and NationalParksAdapter.

7. In MainActivity.onCreate(), set the NationalParksData.DataDir property and call the Load() method prior to creating the ListView adapter:

   NationalParksData.Instance.DataDir =
   System.Environment.GetFolderPath (System.Environment.SpecialFolder.MyDocuments);
   NationalParksData.Instance.Load();

8. Compile and run the app. You should see no noticeable behavior changes, but we are now using the serialization and storage logic in a shareable way.

**Updating NationalParks.iOS to use shared files**

Now, let's move on to update NationalParks.iOS. We have a little more work to do here because if you can recall, we had the file handling logic spread out in several areas.

To update NationalParks.iOS in order to use the shared solution, perform the following steps:

1. Remove NationalPark.cs from the project.
2. Add a folder named NationalParks.Data in the NationalParks.Droid project.
3. Add file links to NationalPark.cs and NationalParksData.cs.
5. In MasterViewController.viewDidLoad(), set the DataDir property before creating the data source for UITableView:

   NationalParksData.Instance.DataDir =
   Environment.CurrentDirectory;
   NationalParksData.Instance.Load();

6. In the DataSource class, remove the Parks collection and remove the loading action of the Parks collection in the constructor.
7. Update the methods in DataSource to reference the Parks collection property in NationalParksData.

8. Remove the Parks property from DataSource and update MasterViewController.PreppeareForSegue() to use the Parks property in NationalParksData.


10. In SetNavData(), remove the Parks collection argument, corresponding private variable, and then update the navigation logic in MasterViewController.

11. Open EditViewController and add a using directive for NationalParksData.

12. In SetNavData(), remove the Parks collection argument, corresponding private variable, and then update the navigation logic in MasterViewController and DetailViewController so that no Parks collection is passed in.

13. Remove the SaveParks() method.

14. In DoneClicked(), replace the logic that adds the park to the collection and saves the collection with a call to NationalParksData.Instance.Save(), as follows:

   ```csharp
   private void DoneClicked (object sender, EventArgs e)
   {
      ToPark ();
      NationalParksData.Instance.Save (_park);
      NavigationController.PopViewControllerAnimated (true);
   }
   ```

15. In DeleteClicked(), replace the logic that removes the park from the collection and saves the collection with a call to NationalParks.Instance.Delete(), as follows:

   ```csharp
   partial void DeleteClicked (UIButton sender)
   {
      NationalParksData.Instance.Delete (_park);
      NavigationController.PopToRootViewController(true);
   }
   ```

16. Compile and run the app. As with NationalParks.Droid, you should see no noticeable behavior changes.
Portable Class Libraries

Portable Class Libraries (PCL) are libraries that conform to a Microsoft standard and can be shared in a binary format across many different platforms such as Windows 7 desktop, Windows 8 desktop, Windows 8 phone, Xbox 360, and Mono. The big advantage with a PCL is that you can share a single binary for all these platforms and avoid distributing source code. However, there are some significant challenges.

One issue we face straightaway is the fact that our code uses APIs that are not supported across all the platforms; specifically File.Exists(), File.ReadAllText(), and File.WriteAllText(). It seems surprising, but most of System.IO is not common across all of the .NET profiles; so, the file I/O logic can be difficult to deal with within the shared code. In our case, there are only three methods and we can easily abstract this logic away from the shared code by creating an IO interface. Each platform that uses our shared solution will be responsible for providing an implementation of the IO interface.

Creating NationalParks.PortableData

The first step is to create the Portable Class Library to house our shared solution. To create NationalParks.PortableData, perform the following steps:

1. Add a new Portable Class Library project to the NationalParks solution. The project template can be found under C# | Portable Library.
2. Remove MyClass.cs from the newly created project.
4. Add a reference to the Json.NET Portable Class Library.
5. Create the IFileHandler interface and add three methods that abstract the three IO methods we need. It will be best to make the read and write methods asynchronous returning Task<>, because many of the platforms only support asynchronous IO. This will simplify implementing the interface on these platforms. The following code demonstrates the required action:

```csharp
public interface IFileHandler
{
    bool FileExists (string filename);
    Task<string> ReadAllText (string filename);
    Task WriteAllText (string filename, string content);
}
```
6. Add a public IFileHandler property to NationalParksData and change all the logic to use this property rather than using System.IO.File, as follows:

```csharp
public IFileHandler FileHandler { get; set; }
...
public async Task Load()
{
    if (FileHandler.FileExists(GetFilename())) {
        string serializedParks =
            await FileHandler.ReadAllText (GetFilename());
        Parks = JsonConvert.DeserializeObject
            <List<NationalPark>> (serializedParks);
    }
    ...
}
...
public Task Save(NationalPark park)
{
   ...
    return FileHandler.WriteAllText (GetFilename (), serializedParks);
}
public Task Delete(NationalPark park)
{
   ...
    return FileHandler.WriteAllText (GetFilename (), serializedParks);
}
```

Implementing IFileHandler

We now need to create an implementation of IFileHandler that can be used by both our projects. We will share the file handler implementation using the file linking method from the previous sections.

To implement IFileHandler, perform the following steps:

1. In the NationalParks solution, create a new Library project named NationalParks.IO and set the Target framework option to Mono/.NET 4.5. This will serve as a shared project for our file handler implementation.

2. Remove the MyClass.cs file created by default and add a reference to NationalParks.PortableData. This will give us access to the IFileHandler interface we intend to implement.
The Sharing Game

3. Create a class named `FileHandler` in `NationalParks.IO`. Add a `using` directive for the `NationalParks.PortableData` namespace and specify that the class implements the `IFileHandler` interface.

4. Use the `Implement` interface menu item under `Refactor` to create stub implementations for each method on the interface.

5. Implement each of the stub methods. The following code demonstrates the required action:

   ```csharp
   #region IFileHandler implementation
   public bool FileExists (string filename)
   {
     return File.Exists (filename);
   }

   public async Task<string> ReadAllText (string filename)
   {
     using (StreamReader reader = File.OpenText(filename))
     {
       return await reader.ReadToEndAsync();
     }
   }

   public async Task WriteAllText (string filename, string content)
   {
     using (StreamWriter writer = File.CreateText (filename))
     {
       await writer.WriteAsync (content);
     }
   }
   #endregion

Updating NationalParks.Droid to use PCL

Now, it's time to update `NationalParks.Droid` in order to use our new PCL.

To update `NationalParks.Droid` in order to use `NationalParks.PortableData`, perform the following steps:

1. In the `NationalParks.Droid` project, remove the `NationalParks.Data` folder, create a new folder named `NationalParks.IO`, and add a reference to `NationalParks.PortableData`.

2. In the `NationalParks.IO` folder, add `Link` to the `FileHandler` class.


---
4. In `MainActivity.OnCreate()`, initialize the `FileHandler` property with an instance of `FileHandler`, place an `await` instance on the call to `Load()`, and move the assignment of `NationalParksAdapter` before the call to `Load()`, as shown in the following code snippet:

```csharp
_adapter = new NationalParksAdapter (this);
NationalParksData.Instance.FileHandler =
    new FileHandler ();
NationalParksData.Instance.DataDir =
    System.Environment.GetFolderPath (System.Environment.SpecialFolder.MyDocuments);
await NationalParksData.Instance.Load();
```

5. Now that we are loading data asynchronously, the `OnPause()` method will likely be called before the asynchronous return of `OnCreate()`. Thus we need to add a null check for the logic in `OnPause()` that calls `NotifyDataSetChanged()`, as follows:

```csharp
protected override void OnResume ()
{
    base.OnResume ();
    if (_adapter != null)
        _adapter.NotifyDataSetChanged ();
}
```


7. Compile and run the app.

**Updating NationalParks.iOS to use PCL**

Now, it's time to update `NationalParks.iOS`. For the most part, we go through essentially the same steps.

To update `NationalParks.iOS` in order to use `NationalParks.PortableData`, perform the following steps:

1. In the `NationalParks.Droid` project, remove the `NationalParks.Data` folder, create a new folder named `NationalParks.IO`, and add a reference to `NationalParks.PortableData`.
2. In the `NationalParks.IO` folder, add `Link` to the `FileHandler` class.
4. In `MasterViewController.ViewDidLoad()`, initialize the `FileHandler` property with an instance of `FileHandler`, place an `await` instance on the call to `Load()`, and place a call to `TableView.ReloadData()` after the assignment of the data source, as shown in the following code snippet:

```csharp
NationalParksData.Instance.FileHandler = new FileHandler();
await NationalParksData.Instance.Load();
TableView.Source = dataSource = new DataSource(this);
TableView.ReloadData();
```


6. Compile and run the app.

**The pros and cons of the code-sharing techniques**

Now that we have some experience with two practical methods for sharing code across Xamarin.iOS and Xamarin.Android apps, let's look at some pros and cons. The following table summarizes some of the pros and cons of each approach:

<table>
<thead>
<tr>
<th></th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
</table>
| **File linking**      | • This allows for a broader use of .NET APIs, assuming that these APIs are supported by all the platforms that will use the shared code. If you are only targeting Xamarin.iOS and Xamarin.Android, this works pretty well. | • This requires source code to be shared.  
• These API dependency issues might not be known until shared code has been compiled for each target platform. |
| **Portable Class Library** | • This ensures platform API compatibility.  
• This allows for distribution of binary code. | • This limits the namespaces and APIs available for use in your code. |
Summary

In this chapter, we reviewed two practical approaches to share code across Xamarin projects as well as other .NET solutions. In the next chapter, we will investigate MvvmCross, a framework that simplifies implementing the Model-View-ViewModel design pattern, increasing the amount of shared code across platforms.
Where to buy this book

You can buy Xamarin Essentials from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.