Cross-platform UI Development with Xamarin.Forms

Xamarin provides developers with tools to develop effective mobile solutions using the .NET framework. Part of the toolchain is an IDE called Xamarin Studio. Even if you use the Xamarin toolchain, you will still need three different user interface construction sets.

This book will show you, with fully-coded examples, how to use both the Xamarin toolchain and the Xamarin.Forms library to code for three platforms at once. It goes from the concept and design of an application to its execution. You will be introduced to common application features such as push notifications, UI, maps, databases, and web services. Next, you will learn to plan the UI using Xamarin.Forms for cross-mobile platform development, and move on to creating custom buttons, extending the UI, and connecting to social sites such as Facebook and Twitter.

What you will learn from this book

- Create a responsive UI, modified to suit the target platform
- Understand the basics of designing an application, and the considerations needed for target platforms
- Construct a complete app using a single codebase
- Design attractive user interfaces
- Bind information to the backend code to generate a reactive application
- Design an effective Portable Class Library (PCL)
- Include a Windows Mobile application within your standard Xamarin.Forms application
- Extend your applications using the Xamarin.Forms Labs library

Who this book is written for

This book is intended for mobile software developers who are fed up with having three different code sets for the same application. If you want to put your code on all mobile platforms with minimum fuss, and just want to develop but haven’t got the time to be digging too far into a particular platform, this is the book for you. Some basic knowledge of C# is assumed.


Cross-platform UI Development with Xamarin.Forms

Create fully operational applications and deploy them to major mobile platforms using Xamarin.Forms

Paul F. Johnson

$ 44.99 US
£ 28.99 UK
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'In the Beginning…'
- A synopsis of the book’s content
- More information on Cross-platform UI Development with Xamarin.Forms
Many years ago, from the fountains of Mount Olympus came forth upon the planet a man, a mystery, and an enigma. Over many years, he grew; he developed amazing biceps, an intellect to rival the greatest minds in the universe, a personality larger than a fair-sized moon, and a smile that would melt the hearts of the iciest of witches. He fought in wars, raged battles against injustice, and was generally an all-round amazing type of guy.

This is not his story.

You see, while he could do all of this really cool stuff, he couldn't work his mobile, and worse, he was clueless about how to make his own apps. Then stepped forth a nice chap from Liverpool, and with a bit of patience, he showed him how to do it.

This is his story.

What made this Scouser worthy of helping the man from Olympus? The simple answer is experience. You see, he was there at the outset of the home computer boom of the early 1980s. He developed code in BASIC, Z80, the 6502 and ARM assembler, C, C++, C#, Pascal, and FORTRAN. He has won awards for programming and is a published author with Packt Publishing.

Add these together and you can see why he was a worthy teacher. That, and he makes a killer cup of coffee!

Paul (for that is his name) is 44, lives with his wife, dog, cats, and son and drinks way too much coffee! You can normally find Paul on the Facebook Xamarin Developers group, where he is an admin. He is currently in the planning stage for a follow up of this book, but this time, he is concentrating on using XAML instead of pure C# to develop Xamarin.Forms applications. This will combine his lifetime love of Dr. Who with his other love that is to create fun code.

He is currently in the middle of buying enough coffee to fuel him through it - Brazil is on high alert!
Preface

One of the pleasures of working in the mobile arena is the speed of development and the improvement in the toolchains. Being at the forefront of allowing the "write once, deploy many times" paradigm, Xamarin never sits still in making life easier for the developer.

Although having C# as the basis of writing code made the business logic (consider anything not to do with the user interface as the business logic) of an app much easier to cater to, it did leave a terrible chunk missing: to employ specialized developers for the UI. Then, the game changed.

There was a new technology out there, an uncharted territory, and that territory was Xamarin Forms. Instead of just one platform, with Xamarin Forms, we have three platforms. The dream of one language for all mobile devices comes another step closer. With Xamarin Forms, the mythical 100 percent code-sharing nirvana for the user interface and program is within reach.

What this book covers

Chapter 1, In the Beginning…, talks about planning your app from the backend to the frontend.

Chapter 2, Let's Get the Party Started, discusses views and gadgets. It teaches you how to add a Forms project to your current project in addition to the application life cycle, inversion of control, web views, and maps.

Chapter 3, Making It Look Pretty and Logging In, focusses on the UI abstraction. It shows you how to style your UI, custom renderers, triggers, and positioning.

Chapter 4, Making Your Application Portable, discusses the considerations for a PCL. It provides information on how to move code from a nonPCL to a PCL.
Chapter 5, *Data, Generics, and Making Sense of Information*, talks about everything you ever wanted to know about generics, but were too afraid to ask. This involves reflection and how to use LINQ to make the manipulation of data much simpler.

Chapter 6, *A View to a Kill*, takes the UI to the next level. This chapter demonstrates how to bring the Android application to Xamarin Forms in order to produce a universal app.

Chapter 7, *Connect Me to Your Other Services*, focusses on Azure, REST, and WCF.

Chapter 8, *What a Bind!* talks about the power behind a data-driven application.

Chapter 9, *Addressing the Issue*, discusses how to use the device address book.

Chapter 10, *This is the World Calling…*, focuses on how to use the device hardware with your Xamarin forms application using GPS and maps.

Chapter 11, *A Portable Settings Class*, talks about the different strategies to implement a settings system that will work on all platforms.

Chapter 12, *Xamarin Forms Labs*, takes a quick glimpse at how to extend the UI through the Xamarin Forms Labs community project.

Chapter 13, *Social Media into the Mix*, teaches you how to add Twitter and Facebook to your app.

Chapter 14, *Bringing It All Together*, brings all the parts of the app together, enabling you to create your own app.
By virtue of using a mobile device, you will have used a messenger app of some description or the other (as well as any number of other apps). As time has moved on, so have the apps. For the humble messenger software, you can now add sound, video, and audio, and have it extend over many messages (a message was traditionally 160 characters, as it was only used by telecom operatives to send small messages informing others of their progress or a fault in the system). How did the apps metamorphose to the apps we have today? The answer is simple—planned changes.

In this chapter, we will cover the following topics:

- Application planning
- Application setup
- Creating the correct provisioning profile for iOS
- The database model

**Application planning**

In the 1990s, there was a piece of software for the Acorn RISC OS range of machines called SBase. The software was one of the most stable, intuitive, and above all simple-to-use applications. I was surprised many years later to find out that this was not down to unit testing, A-B testing, or anything like that. The reason was that before a single line of code had been written, absolutely everything was written by hand first. Not code (this would not be a good plan), but a detailed storyboard and flow.

After a month or so of planning, the work of coding began. However, as the author knew what would go where, the final development time was much less than the time it would have taken, had he just sat down and started coding.
The design templates

There are quite a number of different design templates freely available for download (a quick search on https://images.google.com/ will show them). The following template will be used in this book:

For each new view, a new design will be drawn. While the app will never be a one-to-one representation of the design, it should be close and give an idea of how all sections would work together.

It is also a good idea to use piece of flowcharting software when creating a flow. Again, there are plenty of pieces available, but through trial and error, one of the best that I have found is called the yEd Graph Editor (http://www.yworks.com/en/products/yfiles/yed/).

When creating any piece of software, it's also a good plan to have a shopping list of the features that you would like to see in the final application.

A messenger – a shopping list of features

The shopping list should feature everything that you want in the app rather than everything that you think should be in the app. If you have a large budget, then the shopping list can be longer. Remember, the longer the list, the larger (and more complex) the application.

The basics of any messenger application UI comprises of the following lists:

- Contacts
- Add and remove contacts
- Message composition, add sound, and pictures
• Message summary
• Message display or text to speech
• Message delete
• Login/register/forgotten password
• Forced synchronization

The following are a few details about the basics:

• Contacts can be from Facebook or your contacts, and should also provide a summary of the contact
• Messages need to be entered via a keyboard or speech to text (STT)
• Messages need to be able to include pictures (not video) and sound
• Message notifications (new messages) are required
• Forced synchronization causes the app to download all messages and contacts from the server

**Speech to text and text to speech**

Speech to text is a vital addition to most applications to facilitate those who are visually impaired. These facilities are available on all mobile platforms, and the inclusion will greatly improve the scope and range of users who can take advantage of your new app.

Speech to text is not the same as adding sound to a message; adding sound is different.

**Other considerations**

While Android is a very open platform that allows users to create their own apps for messaging, iOS is very prescriptive over what you can and cannot do. For example, to create a text message, you must use their text message service and software. This is fine, except that it ties you in strongly to what Apple provides, and doesn't give the end user any different an experience; so why should they bother using what you have over what Apple supplies?

To remove this requirement, a web service can be used. The use of the web service gives a number of advantages:

• No requirement to use the built-in messaging software
• Offline storage
• Notifications

---

[3]
In the Beginning…

- Large amount of storage
- Routing facilities
- Ability to download messages to any device

This final point is of great importance for this application.

Cross-platform considerations

As the name of this book includes Xamarin, and you are reading this, you will be aware that we are using C# for the core language. This will already be giving a massive advantage in terms of code sharing (for example, all the data services, SQL, and core code will be the same, irrespective of the platform), and also by using one of the more recent additions to Xamarin. It also means that the user interface can be shared by using Xamarin.Forms.

We can also swap between devices. On my desk, I have to handle a range of Android devices (tablets and phones), a couple of Windows Phones, a couple of iPhones, and an iPad. At any given time, I may have two of the phones on me, and want to use the app on either of the devices. Not only will the data structure need to be the same but ideally, a similar user interface is needed. This is not to say that each platform version cannot make use of some specific tweak on each platform, but the overall experience needs to be the same.

A secondary aspect to having a cross-platform application is to have a common settings file. All three platforms store user data in a different (and largely incompatible) way meaning that for personal data and app settings, a common profile needs to be established. This can be achieved either online, offline, or both, each having their own advantages and disadvantages:

<table>
<thead>
<tr>
<th>Method</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
</table>
| Offline | Locally stored  
App can be used out of network coverage | File can be corrupted  
Not portable between devices |
| Online  | Permanently available  
Unlikely to become corrupted by app failure  
Can be used by any device | Requires connection (updates may therefore not always be in sync) |
| Both    | Portable  
App can be used when not in network range  
Unlikely to be corrupted  
Can be used on any device | Synchronization |
Obviously, from a portability and usability point of view, having both as options is the best way forward; however, synchronizing changes may be an issue: which is the newer settings file?

**Language considerations**

It is no longer acceptable to have apps run in a single language. While it was the norm to say, "Hey! I'm English, so everyone can understand English" a number of years ago, there is no valid reason why that now needs to be the case, as all devices are able to localize given the correct language file.

The issue though is that to have every language covered would cost a fortune—many (human) language translators charge between £50 and £100 per hour (at the time of writing), so unless you're part of a larger organization with deep pockets, this can be ruled out.

**Online translation services**

The next consideration would be to use the likes of Google Translate for the language files used within one version of the app. This is not a bad idea but has a simple problem: though the likes of Google Translate are getting much better, it fails to recognize the context of a sentence so the translation may be incorrect. That said, if the translation is for a message 3 of 10 (stored as message %0 of %1 in the localization file), the returned translation to German would be correct.

This allows translations to any language covered by Google (or the translation service you choose to use).

The only remaining issues are storage of the translated messages, and having them as usable across all platforms.

Consider how Android and iOS store localized strings:

- **iOS**: The text for translation is stored within the language file like the following code:
  
  "Common.Save"="Save";

- **Android**: This is more in line with a standard XML format, as shown in the following code:
  
  `<string name="Common.Save">Save</string>`

Android is closer to a standard XML format while iOS as a Dictionary<string,string>. The two are not easily translatable and really, there isn't a simple, cross-platform way to translate text.
However, there is a way around this issue, which also means that should Google add further languages to their system, further language files need no longer be created. The only caveat is that it requires at least one language file to store the translation data in a database on the device and on the server (for others to use).

While this may be initially slow for grabbing the translation file from the server, it won’t be slow later on.

**The internal data structure**

Windows Phone, iOS, and Android all come with an implementation of SQLite (though Windows Phone does require the package to be installed via nuget). This means that we can have the same data structure for all platforms apart from being able to match up to the server with minimal issues. The code for accessing the databases will be covered in *Chapter 5, Data, Generics and Making Sense of Information*; here we will discuss the data table classes.

Any database can be considered as a series of disparate objects that may or may not be linked. Each of these objects is an entry in a data table.

Consider the following example of a book, author, and publisher:

The Newton-le-Willows Community Library has a hundred books on Android development. For argument’s sake, they have only one edition and a single author for each book. Each book has the following information stored on it: the ISBN, price, author, publication date, and publisher.

The ISBN number is unique to each book, so it is never replicated making it perfect as what is known as a **Primary Key**. The author may have written any number of books (not only on Android, but possibly in Chemistry and the UK Armed Forces between 1960 and 1965, for example). A name is also not unique—I went to school with three people who had the same name as me.

The publisher too, will have a table containing information such as their address.

In terms of a database structure, the publisher and the author will have what is known as a **one-to-many relationship** with the book (the meaning is literally that—one publisher or author to many books) going **FROM** the publisher or author table TO the book. Going **FROM** the book TO the author or publisher shows a many-to-one relationship.

To complete this triangle, there will be a relationship between the publisher and the author. As the publisher is a single entity using multiple authors, it will have a one-to-many relationship with the authors.
The ISBN number will have a *one-to-one relationship* with the publisher and the author.

**The messenger data structure**

For each table, the key data stored needs to be defined. We have, in our application, a simple set up:

- The message
- Contacts
- Backend storage
- The language

Each of these will need defining.

**The message**

A message contains a number of pieces of information

- A unique identifier
- Who it is from (a contact)
- Date and time sent
- Attachments (though this will be from a separate table, linked by ID)
- The previous message Id

The previous message Id is important for message threading. The first message in any thread will have a value of -1. It is then just a case of moving through the message identifiers to correctly thread a message.

Attachments can be either an audio file or an image. It is never a good idea to store these on a device within the database, as it leads to massive slowdown of the application (SQLite was never designed to store BLOB (*Binary Large OBject*) data types).

**Contacts**

Contacts are anyone (including you). A contact can be anyone invited to share in the app with you; it can be read from your mobile device directory or obtained via the likes of Facebook. Each contact will have the following information:

- Contact ID
- Name
In the Beginning…

- E-mail
- Phone number
- Last-used date
- Username

Each message is linked to another message as well as to a contact.

Backend storage
While it is fine to have storage on the device, there needs to be an external server somewhere. Currently, there are two main players when it comes to external servers if you are unable to host things yourself. They are Microsoft Azure and Amazon. Both offer the same sort of facilities, but given that Xamarin uses the .NET framework, it makes slightly more sense to use Azure.

The configuration and setting up of an Azure service is covered extensively in Chapter 14, Bringing It All Together.

Making sure the messages get through
Part of ensuring that messages get through as soon as they are sent is to use a push notification. Essentially, this is sent out from the server to the application. Unfortunately, the procedure for getting this to work is not the same for Android and iOS.

Setting up for Android
For Android, push notifications are a simple affair, as shown in the following steps:

1. Register the application with Google and obtain an API key.
2. Once you have this, go into the Azure portal, click on Mobile Services, select the service name, and then select Push.
3. To obtain the key, go to https://console.developers.google.com/start, and click on Create Project. Enter a new project name. After sometime, the project will appear on the console.
4. Next, select Add an API and switch on the Google Cloud Messaging for Android option.
5. Now select the credentials and create a new Public API access key. You will see the following displayed. Over here, select **Server Key**:

![Create a new key]

6. After this point, you will need to just create a new key by selecting **Create**:

![Create a server key and configure allowed IPs]

**This key should be kept secret on your server.**

Every API request is generated by software running on a machine that you control. Per-user limits will be enforced using the address found in each request's userIp parameter, (if specified). If the userIp parameter is missing, your machine's IP address will be used instead. Learn more

**ACCEPT REQUESTS FROM THESE SERVER IP ADDRESSES**

One IP address or subnet per line. Example: 192.168.0.1, 172.16.0.0/16, 2001:db8::1 or 2001:db8::/64

[Create] [Cancel]
7. Once the API key has been created, copy and paste it onto the Azure push services page (shown in the following screenshot):

8. Once complete, select Save and Android is set up, server side, to send a push notification.

**Setting up for iOS**

For push notifications to work on iOS, you will need a valid iOS developer account with Apple (currently this costs $99 per year). Assuming you have such an account, log in.

If you already have a developer profile set up, skip to the Creating and configuring for push notifications section.

**Creating your iOS profile**

The certificate is your digital signature that can be used for any application, for any given team. The use of certificates enables developers to work on different teams and still have a valid signing certificate. Creating a certificate is simple enough:

1. Click on your Applications folder and navigate to Utilities | KeyChain Access.
2. From the Menu bar, select Certificate Assistant | Request a Certificate From a Certificate Authority... (as shown in the following screenshot):
3. Once this has been clicked, you will be presented with the following window:

![Certificate Assistant Window]

4. Fill in the **User Email Address** using the address that the Apple developer account is registered with.
5. Click the **Saved to disk** radio button.

6. When these have been filled in, select **Continue**. After a few moments, you will be asked where to save the certificate. Give the certificate a meaningful name and save. You can now minimize the Keychain Access application.

You now need to upload the certificate. Again, this is simple enough. Log into your Apple developer console (https://developer.apple.com/):

1. Click on **Certificates, Identifiers and Profiles**, and select **Certificates**.
2. Click on the **Add** icon.
3. You will be presented with the options shown in the following screenshot:
4. You will need to create at least an iOS App Development and an App Store and Ad Hoc certificate.

5. The process for creating either of these certificates is the same. Select the certificate type, and click on Continue.

6. You will be presented with an information page that covers how to create your certificate (performed in the preceding steps). Click on Continue.

7. The next page allows you to upload the certificate you previously created to the Apple Developer portal. Click Choose File, navigate to where you saved the certificate, and click on OK to upload.

8. Once the file has been selected, click on the Generate button. After a while, the certificate can be downloaded. Download and double click on the certificate to install.

9. You can check whether the certificate has been installed by checking the Keychain Access application.
Creating and configuring for push notifications

We now have to create a profile for our application. On the developer portal, select Identifiers.

1. Select the + button to create a new app.
2. Fill in the App ID description (this is just a name for the app).
3. Add an explicit app ID. This is recommended to be a reverse, domain-style string (for example, com.packt-pub.messenger).
4. Towards the bottom of the page, you will need to select Push Notifications.
5. Once complete, you will be presented with something similar to the following screenshot:

![Confirm your App ID.

To complete the registration of this App ID, make sure your App ID information is correct, and click the submit button.

App ID Description: Messenger
Identifier: com.packt-pub.messenger
App Groups: Disabled
Associated Domains: Disabled
Data Protection: Disabled
Game Center: @Enabled
HealthKit: Disabled
HomeKit: Disabled
Wireless Accessory Configuration: Disabled
iCloud: Disabled
In-App Purchase: @Enabled
Inter-App Audio: Disabled
Apple Pay (Beta): Disabled
Passbook: Disabled
Push Notifications: @Enabled
VPN Configuration & Control: Disabled

[Submit]
6. Once you are happy with all the selections, click on **Submit**.

7. Next, a certificate has to be generated for the notifications. If you're used to creating a certificate, generate two—one for development and one for distribution. If you're not, follow the instructions used for generating your initial certificate.

8. Select the app created from the App IDs page on the developer site (as seen in the following screenshot):
10. Go to the section marked Push Notifications.

![Push Notifications](image)

Apple Push Notification service SSL Certificates
To configure push notifications for this iOS App ID, a Client SSL Certificate that allows your notification server to connect to the Apple Push Notification Service is required. Each iOS App ID requires its own Client SSL Certificate. Manage and generate your certificates below.

- Development SSL Certificate
  - Create certificate to use for this App ID.

- Production SSL Certificate
  - Create certificate to use for this App ID.

11. Starting with the Development SSL certificate, click on Create Certificate and follow the on-screen instructions. When requested, select the development certificate created in step 7.
12. Once generated, download the certificates and double-click to install.

Exporting the certificate for Azure
The final step for creating the push notifications for Azure is to export what is known as a .p12 file. The steps for exporting the certificate are given as follows:

1. To access this, start the Keychain Tool, and locate the freshly installed development push notification entry (highlighted in the following screenshot):
2. Select **File | Export** and give the file a name. Then click **Save**, as shown in the following screenshot:
3. This needs to be imported into Azure. As with Android, go to **Mobile services** | **Push**, and move to the **Apple** section:

![Apple Push Notification Settings](image)

4. Click on upload, and upload the `.p12` file you have just exported from the keychain tool. Ensure that **Sandbox** is selected when requested. When complete, you will see the following screen:

![Apple Push Notification Settings](image)

Azure, iOS and Android push notifications are now set correctly on the server side.

**Adding packages to your application**

For the application to be able to access the Azure services, it requires an additional package. You can manually create it, but it is easier to use the NuGet package. I will cover this in detail in *Chapter 3, Making It Pretty and Logging In.*

**Summary**

We’ve covered quite a lot of information in this chapter, from enabling push notifications to registering your app with Google and Apple, and by now you should have at least an appreciation for what is involved in even the simplest of applications.

In the next chapter, we will start looking at the user interface and at using Xamarin forms for the application UI.
Where to buy this book

You can buy Cross-platform UI Development with Xamarin.Forms 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.