PART 3  FROM WORKING CODE TO THE STORE..........................409

13  ■  Running mobile apps on physical devices       411
14  ■  Testing mobile apps using Xamarin UITest       436
15  ■  Using App Center to build, test, and monitor apps 467
16  ■  Deploying apps to beta testers and the stores    505
Now that you’re familiar with the MVVM design pattern and how it can be used to build testable, reusable code, this part of the book expands on that knowledge. It shows how you can build cross-platform apps on iOS and Android, starting with the cross-platform code and moving on to platform-specific UI code, taking a couple of example apps from design through to fully working apps. This part moves up, layer by layer, though MVVM, before diving into the Android and iOS UIs.

Chapter 6 introduces two example apps that will be built throughout the rest of this part. It looks at how to design an app, focusing on what code goes in what layer in the MVVM design pattern. Finally, it covers creating the solutions for the example apps and looks at the project and application properties for a Xamarin mobile app.

Chapter 7 focuses on the model layer. You’ll see how to build simple models and more complex model layers with services and repositories, and you’ll learn how to access SQLite databases and web services. It also introduces unit testing, showing how easy it is to unit-test well-structured code.

Chapter 8 moves up a layer and covers view models. It considers how state and behavior are represented, covering properties, commands, and value conversion. It also shows how to test UI logic using unit testing.

Chapters 9 and 10 cover the view and application layers on Android, recycler views for showing lists of data, and multiscreen navigation. It then shows how to add polish to an app by creating app icons and splash screens.

Chapters 11 and 12 focus on iOS, working on the application and view layers of the first example app, covering view controllers, UI controls, storyboards, and
auto layout and constraints, table views, and multiscreen navigation. It then covers
app icons and launch screens.

After you’ve finished these chapters, you’ll have working iOS and Android apps
that will be made production-ready in the third part of this book.
In the first part of this book we looked at the foundations of an MVVM-based cross-platform mobile app. We looked at what the different layers in MVVM are all about and how to structure your code inside them. We also looked at how threads work in mobile apps and how you can keep your apps responsive when performing long-running actions. Now it’s time to think about how you can start building a cross-platform mobile app.

As developers, we’re often tempted to start a new app by firing up our IDE and clicking File > New Project. But a little planning goes a long way, so before we start...
creating any code, let’s first think about what we’re building and then think about how to build it. Then we can create a new solution and look at how it differs from the kinds of solutions most C# developers are used to.

Throughout this part of the book we’ll be looking at two app examples—one very simple and one slightly more complex—and I’ll use these apps to introduce the ideas and concepts you’ll need to know to build production-quality cross-platform apps using Xamarin and MVVM. In this chapter we’ll talk about designing these apps: you’ll see how to create the solutions for them, and I’ll explain these solutions in detail. The next few chapters will explain how you can build up your Xamarin apps layer by layer, from the model, to the view model, to the view, looking at our two example apps and at other features you can add to your apps in each layer.

6.1 **Introduction to designing a cross-platform app**

The Facebook app is popular on iOS and Android, catering to millions of users worldwide. On both platforms it offers the same functionality, but the way it offers its features is different on each platform. Figure 6.1 shows what these apps look like at the time of writing.

On Android, all the buttons are on the top of the screen. This is because Android has its three navigation buttons at the bottom of the device, either as physical or software buttons. If an app has buttons just above these navigation buttons, it’s too easy to accidentally tap a navigation button when you meant to tap an app button, so Android apps don’t have buttons on the bottom as a good design practice.

iOS, on the other hand, doesn’t have easy-to-tap buttons at the bottom—it has a hardware home button, allowing the iOS Facebook app to have some of its buttons on the bottom. These buttons are used to select different views: your news feed, marketplace, notifications, and settings. They’re tab buttons, each one representing a new tab in a tab control. (Tab controls consist of a number of pages indicated by tab
buttons in a bar, and tapping a tab button shows the page represented by that tab, similar to the tabs on folders in a filing cabinet.

On Android, swiping from right to left or left to right changes tabs; swiping on iOS won’t change tabs, but swiping in the Facebook app will slide the screen out to show the Messenger screen (making it look like the Messenger screen is below the main screen—following the iOS human interface guidelines on depth). This is a simple but important difference—Android has tabs at the top with swiping to change tabs, whereas iOS has tabs at the bottom without swiping.

These differences may seem small, but they’re important. They involve navigation paradigms that are common to each platform, making the apps consistent with other apps on their own platforms, whether they’re apps that come with the OS or from third parties. This means a new user can start using an app and already have an idea how to navigate it.

With Xamarin you can build an app for multiple platforms in one Visual Studio solution with lots of shared code, but you have to be careful with your UIs. You shouldn’t always build identical apps on both platforms; instead, tailor the user interface and user experience on each platform. You’ll want to maximize code reuse, but not necessarily reuse the same UIs.

Luckily, some of this complexity is abstracted away from you—if you create tabs in your apps using out-of-the-box tab controls, you’ll get the appropriate behavior, as shown in figure 6.2. The problems come when you need something that doesn’t come from an out-of-the-box control.

Google has released a Gmail app for both Android and iOS (figure 6.3). On iOS, to write a new email you tap a button on the top toolbar. This is pretty standard for iOS apps, using the top toolbar for action buttons. On Android it does something different—instead of having a button on the top toolbar, it has a floating action button (FAB). This is a round button near the bottom right of the screen (a little bit up from the bottom to avoid the risk of accidentally tapping a navigation button), and it uses a
shadow effect (referred to as elevation in Android-speak) so it looks like it’s floating above the page. This is a standard Android UI paradigm—a lot of Android apps use a FAB for the most common action on a particular screen.

These differences aren’t provided for free by out-of-the-box controls. Instead, they’re different controls added in different ways. On iOS the developers had to explicitly add a toolbar button, and on Android they had to add the FAB.

As developers of cross-platform apps, we have to keep these differences in mind. It’s easy as a consumer to use one platform and get used to the way it works, but to be a successful cross-platform developer, you’ll need to get used to both platforms so you can always think of your UIs in terms of each platform.

**Mobile apps are constantly changing**

At the time of writing, GMail uses the iOS toolbar button and Android FAB. This may change in future releases because Google is really trying to push its own design standard. It’s worth getting a few apps on different platforms and looking at their differences to get an understanding of how apps can provide the same functionality in different ways.

Android has also released a bottom-navigation component, to provide iOS-like tabs at the bottom. This is pretty recent, but it may mean that Android apps will start to support bottom navigation.
With this in mind, let’s now start to think about the design for a couple of apps. One will be a single-screen square root calculator. The other will be a multiscreen counter app (an app that you can use to count different things, like how many cups of coffee you’ve had). We’ll take both these apps through from design, to looking at the code structure, to creating a new solution and structuring the code.

6.2 Designing the UI and user flows

For the rest of this and the upcoming chapters, we’re going to focus our attention on a couple of app examples. One is a simple square root calculator (like the example from chapter 2), which we’ll call SquareRt, in keeping with the current trend for naming things by taking normal words and losing vowels. The other will be a counter app supporting multiple counters, and we’ll call this one Countr. Let’s look at these in turn and consider the design of their UIs and user flows. Later you’ll see how this maps to the architecture of each app.

When I refer to the **UI**, I’m referring to the user interface presented to the user. By **user flows**, I’m referring to the user’s experience—the actions the user can take to flow through the app, the interactions they have with the UI, and the results of these interactions on screen.

6.2.1 SquareRt—a simple app for calculating square roots

The aim of this app is to let the user enter a number and then to calculate its square root. It’s a fairly simple task, so we don’t need a complex UI. This is also the kind of app that could be the same on both iOS and Android.

---

**Apple requires high-quality apps**

This is a simple app example for illustration, and not something that you should ever build and try to submit to the app stores. Google has a fairly lax attitude toward the quality of apps that can be submitted, whereas Apple is fairly draconian (although both are strict about offensive material or copyright violations). If your app doesn’t do anything of value (such as just calculating a square root), it’s very likely to be rejected from the app store. According to Apple’s App Store guidelines, there are “lots of serious developers who don’t want their quality apps to be surrounded by amateur hour.”


---

Before we can start cutting code, we need to think about what to build. Thinking about the UI is a good way to divide up the code. We’re using MVVM after all, so we need to consider the model layer, the views, and their corresponding view models. Once we know what UI we need, we can start to define our views, and then the view models.
A good way to define the UI is to think about the user flows—what actions the user will want to take, and what the results of these actions will be. Once you have these actions, you can start to map them to the UI and define what your UI should look like. Let’s draw a simple flowchart of the only user flow through the SquareRt app, shown in figure 6.4.

This flow is very simple—the user can only use this app for one thing. They need a way to enter a number, something in the app needs to calculate the square root, and then the square root is presented to the user. The UI for this is relatively easy to imagine—you need a way to enter the number, a way to kick off the calculation, and a way to show the results. Figure 6.5 shows some options.

The first two UIs in figure 6.5 have a text box where the user can enter the number, and a button that kicks off the calculation. The third removes the button—from a user perspective, if they’re entering a number, they obviously want to see the square root, so the app could calculate it automatically every time the value in the text box changes. This is a good option to consider—the less the user has to do, the better the experience.

We’re going to use the third UI in this book (figure 6.6), but it’s a good exercise to think about the other UIs, or to consider other designs of your own, as we delve deeper into designing this app.
6.2.2 **Countr—an app for counting multiple things**

The SquareRt app is a very simple example, but most apps are a lot more complicated. Our second example is an app called Countr that allows the user to define multiple counters, and to increment them whenever they want, such as to track the number of cups of coffee they’ve had, or the number of times they’ve been out for a run. This app will need to show multiple counters, will need the ability to add or delete a counter, and will need a simple way to increment each counter. Figure 6.7 shows these user flows.

![Figure 6.7 The user flows for the Countr app—showing, adding, deleting, and incrementing counters](image)

Showing lists is a very popular thing to do in mobile apps. Think about the apps you use most often—probably most of them deal with lists or grids of data. Email apps show a list of emails, Facebook shows a list of posts, Twitter shows a list of tweets, messaging apps like WhatsApp show lists of messages. In all these apps you have a scrollable list of data. You read what’s on the page and then “push” the items up by swiping up on the screen to see what’s below. This is a popular paradigm, so we’ll use it for our Countr app, with the main part of the UI showing a list of counters.

As you’ve already seen, Gmail on iOS and Android have different ways for the user to create a new email—a toolbar button on iOS and a FAB on Android. We’ll follow this convention in our app with the iOS version having a toolbar button to add a new counter, and the Android version having a FAB to do the same thing.

Often apps with lists use swiping to delete—email apps allow you to swipe an email to the left to display a Delete button below the email, which you can tap to delete. This paradigm would be good for deleting counters.

Another thing you see with lists is buttons against each item, allowing you to perform some action, such as retweeting a tweet in the Twitter app or liking a post in
Facebook. Again, this is a popular paradigm, so we’ll use this in our Countr app to increment a counter.

Unlike with our simple SquareRt app, the Countr app will have different UIs on iOS and Android, at least when it comes to adding counters. This is something you always have to keep in mind—Xamarin allows you to build cross-platform apps, but you should always build the UIs in a way that’s right for each platform. Cross-platform core, and a platform-specific UI. Don’t be tempted to build one UI for both platforms—if it goes against the standard UI of one platform, it will only confuse users.

Figure 6.8 shows the different UIs we can use for this app on both iOS and Android.

![Figure 6.8](image)

**Figure 6.8** iOS and Android UIs for the Countr app with different UI conventions on each platform, such as an Add button in the toolbar on iOS, but a FAB on Android

### 6.2.3 Defining user flows and UIs

We’ve just looked at the user flows for our two example apps. But the hard part can be defining these flows, so how do you go about doing it? I like to use these steps:

1. Start by thinking about the high-level actions that the user will want to use your app for, such as counting something.
2. From these high-level actions, think about how they can get the app in a state where they can perform these actions, such as adding a counter so that they can count something, and showing all the counters to see what they can count.
3. Think about the ancillary tasks they might want to perform around this state, such as deleting a counter.
4. Think about the steps the user takes to perform each task or action, such as viewing counters, and the end results, such as seeing the counters.
5. Think about the general tasks your app needs to do in order to go from the starting point to the end result, such as loading counters from some kind of storage.

By following these steps, you should be able to build some simple flowcharts for your app, like the ones you’ve seen already. The flowchart should start with the high-level
action the user’s trying to achieve, starting from the place in your app where the user will likely be when they kick off these actions. Then it should go through one or more steps to achieve this action, either user-based steps (something the user has to do) or system steps (something the app does). Finally, it ends with a result that may or may not involve the user.

Figure 6.9 shows a simple example. The user wants to see counters, and the end result is that the counters are shown. The step required to get there is to load counters from storage.

Once you have these flows, it’s easy to start mapping them to a UI. Your UI needs to provide a way to kick off each flow and provide the result. In this example, the UI needs a way to kick off loading the counters from some kind of storage when it’s opened and then showing all the counters. This means you need a UI with a control that can show a collection of data, and the normal way to do this is using a vertically scrolling list control. When you’re thinking about how to represent tasks and results on your app’s UI, take a look at how other apps do it—sometimes there are standard ways, like lists, that you can use to make your app easy to use. After all, if you’re using a popular UI paradigm, your users will probably already be used to it, so they’ll be comfortable in your app.

The Countr example demonstrates that you can create cross-platform apps that have the same user flows, but with different UIs. I can’t stress enough how important it is to always consider the differences between the UI paradigms on iOS and Android. Using Xamarin, you can build cross-platform apps, but that doesn’t mean your apps have to be exactly the same on both platforms. It’s worth spending time getting to know how each platform works so that when you design your apps, you can keep these differences in mind—even if the difference is as simple as using a FAB on Android and a toolbar button on iOS.

There are many opinions about how to build a mobile app with a good user experience—the topic is worthy of a book in its own right (such as *Usability Matters* by Matt Lacey, Manning, May 2018), but as a simple starting point I recommend looking at your app the way we’ve looked at our two examples. Start by considering the user flows—think about the interactions your user will have with your app. Then think about how you can map those interactions to the UI in a way that makes sense on each platform. It’s also good to think about how the user can achieve each flow in as few steps as possible, in a way that makes sense on each platform.

We’ve now designed the user flows and sketched out the UIs, so let’s look at converting these into an architecture that follows the MVVM design pattern.
6.3 Architecting the app

Now that we’ve worked out what the UI should be, it’s time to start thinking about the architecture. As you’ve seen in the preceding chapters, there are three layers and two thread types to think of. You have to consider what goes in the model layer, what goes in the view-model layer, and what goes in the view layer. And for the code in the model and view-model layers, you need to consider what code needs to run on the UI thread and what can run in the background.

### 6.3.1 Which layer?

As you start thinking about the structure of your code, you need to consider which layers the different parts of the code go in. Think back to the layer diagrams from previous chapters, as shown in figure 6.10. Remember that the code responsible for UI interactions encompasses the view layer, the view-model layer, and the binding—the view is the platform-specific UI widgets, and the view model is the cross-platform UI logic bound to the view.

**SQUARERT**

For SQUARERT, we want as much code as possible in the model and view-model layers. This is cross-platform code that’s shared between the Android and iOS apps, and we only want to write it once and reuse it on both platforms.

Let’s see how the SQUARERT app code can be divided up between layers. We can take the user flow we’ve defined and map it across the layers. This is shown in figure 6.11.

What we see from this exercise is that we need one view with a control that a user can enter text into and one control to show the result. We also need a corresponding view model that can bind to those controls, and a model layer that can do the calculation. This very quickly leads to three classes that are the main structure of our app, as shown in figure 6.12.

**COUNTR**

Let’s repeat the same exercise with our Countr app, mapping the flows we’ve already defined into our three MVVM layers. This is shown in figures 6.13 and 6.14.
Figure 6.11 User flows can map to the MVVM layers, with user interactions spanning the view and view-model layers.

Figure 6.12 Once you’ve mapped user flows to the MVVM layers, you can map classes to these as well.

Figure 6.13 Mapping showing and adding counters to the view, view-model, and model layers
Again, just like with SquareRt, you can see a pattern of classes in these layers. We need to display a list of counters that can be manipulated (such as adding and deleting them), so we need a view and view model for the UI, as well as some kind of service class in the model layer that stores and retrieves the counters from some kind of storage. We also need something in the UI to add a new counter and enter its details, so we need a view and view model for this new counter. This gives us the classes shown in figure 6.15.

**Dividing apps into layers**

Once you’ve worked out your user flows, it should be obvious which parts of the flow involve the UI and which parts don’t. Any direct interaction with the user needs some kind of UI, and anything else doesn’t. This means it’s relatively simple to map user flows to the MVVM layers. Anything that involves the UI lives in the view and view-model layers (the UI in the view layer, and the state and behavior in view models), and the core business logic lives in the model layer. Your flow could switch between layers as many times as necessary—user does x, app does y in the background, it asks the user to confirm on the UI thread, does z in the background, and then shows the user a result.

The process of adding a counter needs something in the UI layer that the user can interact with to start the add flow, such as an Add button in the view and a command to handle it in a view model. Next, the user needs to give the counter a name, so there needs to be some kind of UI, such as a new screen with a text box where the user can enter the name, and a corresponding view model to get this name. Then the app needs to create the counter and store it somewhere, and this is handled in the model layer. Finally, the user needs to see the new counter in the list, so the UI for showing
the list of counters needs to be updated, which means an update in a list of counters stored in a view model, which is reflected in a view.

6.3.2 Which thread?

We’ve divided our example apps up into layers to match the MVVM design pattern, so the next thing to do is think about multithreading. As you saw in the previous chapter, it’s important that our app remains responsive, so we should start thinking now about what thread our code runs on.

There’s a very simple rule to follow here—if the UI lags or is unresponsive for more than about 200 ms, the user will notice a perceptible lag. More than this and it feels like the app has locked up, and it can take only a few seconds before a user is fed up waiting and kills your app. You should always run any action that could take more than about 100 ms on a background thread.

Of course, this isn’t always easy to judge, especially when you’re in the process of developing your app. Most developers have high-powered versions of the latest and greatest devices, but most users don’t. What takes 50 ms on your top-of-the-range iPhone 7 might take 500 ms on an old iPhone 4s. Making a web call might be almost instantaneous when calling a development web service running on your development machine and accessing it over WiFi, but it might take multiple seconds in the real world using 3G.

Here are some good basic guidelines:

1. Always test on a poor device—you can pick up older devices for not very much money, either through clearance sales or secondhand from sites like eBay or Craigslist. It’s worth having a device with the lowest specs you want to support for testing.
2. Always assume anything involving the network will run slowly, so always make network calls on a background thread.
3. When storing anything locally, or retrieving anything from a database such as SQLite (a popular mobile database that comes as part of iOS and Android) or from a file, always do this in the background.
4. If you’re not sure, do it in the background.

A good way to work out which code needs to run on the UI thread or a background thread is to take each task and ask yourself a few questions: does it involve the UI, does it need external resources, is it slow? Figure 6.16 shows a flowchart for this.

If you map your user flows by following this flowchart, you should be able to easily work out in which thread your code should be run.

**Android has a tool that can help check your code**  
Android has a strict mode that you can enable to get feedback on your code, to see if it’s running in the correct thread. You can use this to get feedback about whether an action is taking too long on the UI thread. There are more details on **Strict Mode** in the Android developer’s reference: [http://mng.bz/nDI5](http://mng.bz/nDI5).
When using third-party code, such as NuGet packages or Xamarin components, it’s always good to check whether the code has any async methods. For example, if you use HttpClient from the System.Net.Http namespace in the .NET Framework, you’ll see it has methods like GetAsync, which returns a Task<HttpResponseMessage>. Whenever an async method is exposed, you can usually be sure that internally it will create a Task to run long-running actions on a background thread. You can await this from the UI thread, and your app should remain responsive because the HttpClient handles the threading for you. Obviously, there are no guarantees, so it’s good to check the behavior first.

Let’s now think about the threading requirements of our example apps. We’ll take each user flow and consider what needs to happen to achieve it. From there, we can work out if each part of the flow needs to run on a background thread.

**SquareRt**
The user flow for the SquareRt app has three parts—two that involve the UI (getting the initial number and showing the result), and one that involves a calculation. Although this is a relatively complex calculation, it’s pretty quick—it will run in fractions of a millisecond, so we don’t have to think about background threads at all. Everything can run on the UI thread (figure 6.17).

**Countr**
Unlike SquareRt, Countr does a bit more than just a single calculation. It includes a simple calculation—incrementing a counter, which could be run on the UI thread.
without any issues—but “storage” is the key word in these user flows when thinking about threading. We haven’t discussed the storage of data yet (we’ll look at storage in the next chapter), but any kind of storage involves making a call to something potentially slow. If you write to a SQLite database, or a file, or a web service, it’s good practice to do it in a background thread to shield your UI from anything that might make it unresponsive.

**FILESYSTEM ACCESS CAN BE SLOW** It’s often assumed that saving files by writing to flash memory in a mobile app is fast, because the hardware involved is pretty quick. Although this is generally true, the flash memory has a filesystem on top of it that may not be the best at handling concurrency. If it’s busy performing a large file operation, such as saving a downloaded update, your fast disk access might wait a short time before running, making the save take longer than expected.

In the user flows, you can do anything that involves the UI on the UI thread, and anything that involves storage on a background thread. Even though the calculation to increment a counter is fast, we still need to think about doing it on a background thread because the result of the calculation will need to be stored. Figure 6.18 shows how the incrementing of a counter would be handled across the UI thread and a background thread.

**Figure 6.18** The Countr app needs to access storage, so it’s better to do this on a background thread before coming back to the UI thread to display the results.

### 6.3.3 Mapping code to layers and threads

We’ve looked at how to map user flows onto MVVM layers and threads, and this is a good exercise to go through as you start out building cross-platform mobile apps. Figure 6.19 shows a layout you can use to help map your user flows and code. Print out or photocopy a few copies, or grab the SVG version from this book’s Git repository to use with your favorite drawing tool if you’d like to save paper.

You should start by thinking about the distinct actions your user would want to perform with your app, as we’ve discussed. Then break them apart into separate steps—what would a user do step by step, and what would the app do step by step. Draw these flows on the diagram, thinking about which layer each should go into—UI interactions go on the view layer with the corresponding view model, and steps the app takes
To help work out which layer or thread to use, try mapping your user flows on this diagram.

<table>
<thead>
<tr>
<th>View Model</th>
<th>UI Thread</th>
<th>Background Thread</th>
<th>Model</th>
<th>UI Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>View</td>
<td>UI Thread</td>
<td>Background Thread</td>
<td>Model</td>
<td>UI Thread</td>
</tr>
<tr>
<td>View Model</td>
<td>UI Thread</td>
<td>Background Thread</td>
<td>Model</td>
<td>UI Thread</td>
</tr>
<tr>
<td>View</td>
<td>UI Thread</td>
<td>Background Thread</td>
<td>Model</td>
<td>UI Thread</td>
</tr>
</tbody>
</table>
Creating the solutions

6.4 Creating the solutions

Once you’ve created a rough app architecture based on the sort of classes you want, what layer those classes represent, and what thread your code should run on, it’s time to fire up your IDE and create the actual solutions. In the rest of this chapter, we’re going to create solutions much like we built the Hello Cross-Platform World example in earlier chapters. Then we’ll look at some new concepts that are important in mobile app development—app property files, SDK versions, and linking.

Everything in the remainder of this chapter is relevant to both SquareRt and Countr, and over the next few chapters we’ll start writing the code to turn the new solutions into fully working apps. We’ll just be creating the solution for SquareRt here, but everything we’ll discuss is relevant to both apps, so repeat the process for Countr when you’re done with SquareRt.

The first thing to do is to create a new solution. We’ll be using the same Visual Studio extension we used in chapter 2 and creating the same project type. Name your project SquareRt (or come up with your own name, of course). In Visual Studio for Mac, create an MvvmCross Single Page Native Application from the Other > .NET section. On Windows, choose MvvmCross Single Page Native Application under Visual C# > MvvmCross, and delete the Universal Windows and WPF projects.

Now that you have your solution, let’s take a look at some of the ways that the projects in this solution will differ from what you’ve seen before in C# projects, such as desktop or ASP.NET web applications. Mobile apps are different from other C# applications—they run on devices with limited hardware and with an OS that changes dramatically every year. This means your apps need to be very aware of the OS version and what APIs are available, as well as be as small as possible. They also expose a whole raft of properties to the OS and app stores via a file in the app package that provides information about your app. Let’s start by looking at how these app properties are set.
6.5 Application properties

When you build and ship your mobile app, you have to bundle some information inside your app. This is used by both the relevant app store and the OS to get information about your app, such as its name, icon, supported OS version, and app version number. Both iOS and Android ship an XML file containing this information.

We’ll look at a number of these properties here, but not the app icon. Mobile apps have to run on devices of all shapes and sizes, so there are some complications when it comes to images, and we’ll look at these for Android in chapter 9 and iOS in chapter 11.

6.5.1 Android manifest

The AndroidManifest.xml file in the Properties folder of the solution is shipped with your app package and provides information about your app to the Google Play Store and the Android OS. This includes which SDK version you’re targeting (more on this later in the chapter), what permissions your app needs, the app’s name, its version, ID, and icon. In a native Java Android application built using Android Studio from Google, the manifest will also contain information about the classes that make up your app.

Luckily, as Xamarin developers, we don’t need to worry about explicitly adding this information to the manifest XML file. Instead we mark the relevant classes with attributes, and these get added automatically to the copy of the manifest file that’s packaged inside the compiled application at build time. Again, this will be covered in chapter 9, but you may have seen this already in the FirstView activity in the FirstView.cs file in the Views folder of the Android app—this class was marked with an Activity attribute ([Activity(Label = "...")]), indicating that at build time it should be added to the manifest as an activity with a particular label.

Although this is an XML file, there’s really no need to manually edit the XML. Visual Studio comes with an editor for this: you can access it in Visual Studio for Mac from the Project Options dialog or by double-clicking the AndroidManifest.xml file in the Properties folder (figure 6.20); on Windows you can access it from the project Properties tab (right-click the app and select Properties). If you open it by double-clicking the file in the Solution Pad in Visual Studio for Mac, it opens in a tab with two subtabs—one to edit the file using the same editor as the Project Options dialog, and the other showing the raw XML (figure 6.20). In Visual Studio, double-clicking the file in the Solution Explorer opens a tab with just the raw XML.

There are several items of interest to us now:

- Application name
- Package name
- Version number
- Version name
- Required permissions
The AndroidManifest.xml file lives in the Properties folder.

![Image of AndroidManifest.xml in Visual Studio](image)

**Figure 6.20** The Android Manifest file in the Solution Pad and in the editor

### Application Name

The application name is the name of your application both in the Google Play Store and on your device. You’ll notice in figure 6.20 that this is set to `@string/ApplicationName`, which is a resource reference. You’ll see these a lot in Android apps—rather than hard-code a value, you reference a resource. These resources are in the Resources folder, which contains a subfolder called values containing an XML file, string.xml. If you open this file, you’ll see the following.

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="VersionName">1.0</string>
  <string name="ApplicationName">SquareRt</string>
</resources>
```

If you want to rename the app, you can edit the value of `ApplicationName` in this XML file, and the app will automatically be renamed next time you compile.
We’ll cover these more in chapter 9, but the main reason for using resource files for storing values is so that you can localize your app easily. In this example we only have one strings.xml file, but you can have multiple resources based on the locale of the app’s user—one file for U.S. English users, one for Chinese users, one for French users, and so on. By containing all your strings in one file, it’s easy to get your app translated—there’s only one XML file to translate. Localization is outside the scope of this book, but you can find more information in Android’s “Localizing with Resources” API guide: http://mng.bz/fb7Y.

**Package name**
The package name is the unique name for your application package, and it’s used to identify your app on the Google Play Store. When you want to push app updates to the store, this package name identifies which app is being updated. The normal form for this name is to use your company or personal domain name reversed, suffixed with an identifier for your app. For example, if I were creating this app for publication, I’d have a package name of `io.jimbobbennett.squarert`. Once your app has been published to the store, you can’t change this value, or the Play store will think you’re publishing a different app.

Unlike the application name, there’s no need to define this in the strings.xml file because this will never change to match a locale. Once this name is set and your app is published, you can’t change it, so be sure to set it correctly before publishing.

**Version number**
This is the version number of your package (also referred to in the XML as the version code). This number is used by the Google Play Store and the device to track upgrades—if the package on the store has a higher version number than the one on the device, the app can be upgraded. You can only push to the store an updated package with a higher version number than the one already in the store—it doesn’t have to be 1 higher, you can increment it by however much you want, it just needs to be higher.

There’s a limit on this number—it’s a large limit (2,100,000,000 to be exact), but it can easily be reached if you use automated build tools that increment this number or set it to values based on the current time or source code revision number. Once it’s reached, you can no longer upgrade your app, so be careful with large numbers.

**Version name**
This is your internal version name for your app, and it’s a string that can be set in the manifest or in the string.xml resource file, as shown earlier in listing 6.1. Most developers use a multipart version so that they can internally track releases. You can set this to whatever is relevant to you.

**Required permissions**
When you’re building mobile apps, you can closely integrate your app with the features of the device as well as with other apps that come built into the OS, such as a camera, contacts, and calendars. Obviously, there are privacy concerns with this—you
wouldn’t want a malicious app accessing your private details and uploading them to a server somewhere, so both Android and iOS require your app to request permissions from the user before you can do certain things. Android has a changing permission model—on older OS versions, you’d specify the permissions you needed in the application manifest, and at install time the user would be prompted to give your app these permissions (and, at update time, if your app needed new permissions, the user would be asked to confirm them before updating). In newer versions of the Android OS, you can also ask for some permissions at runtime, explicitly popping up a dialog asking the user’s consent before doing something.

The required permissions section allows you to request permissions up front by ticking the boxes against the permissions that your app requires. These permissions are shown to the user at install time, and the user has to agree to them before your app can be installed. It’s always good to request as few permissions as you need, or users might refuse to install your app. Certainly any permissions that don’t appear relevant will cause a user to pause before installing. For example, our SquareRt app doesn’t need permissions to access the user’s contacts or photos, and seeing a request for these permissions would definitely make a user refuse to install the app!

### 6.5.2 iOS info.plist

Like Android, iOS also has a properties file that ships with the app package, called info.plist (plist is short for property list). The info.plist file can be edited using a built-in editor by double-clicking the file (figure 6.21), or on Visual Studio on Windows from the iOS App project Properties tab.

The info.plist file in the iOS app contains the application settings that the OS needs to know about, such as the name, icons, supported orientations (portrait or landscape). This is an XML file that you can modify directly if you’re so inclined, but Visual Studio has an editor that makes it easy to manipulate. The editor has three tabs: Application, which allows you to change the main application settings; Advanced, which configures things like the document types your app supports for extensions or URL types so your app can handle being launched for links; and Source, which is a key/value type editor for editing raw values without having to interact with the XML. The XML syntax isn’t simple—you have to define nodes based on types, so you need to know what type to use for what value. This editor is a great help, and even the Source tab makes it easy for you by using the right types automatically.

Unlike Android, permissions aren’t requested here. Instead, some permissions are granted by default (such as internet access) and others are requested at runtime.

These are the fields we care about:

- Application name
- Bundle identifier
- Version
- Build
- Devices
- Device orientations
APPLICATION NAME

As the name suggests, this is the application name that will be shown in the app store, and by default it’s used as the name on the iOS SpringBoard. Unlike Android, you can’t reference a string resource for this; it has to be set in the info.plist file on the Application tab. You can, however, localize it. See the “iOS Localization” Xamarin documentation for more information: http://mng.bz/6B91.

On the iOS SpringBoard, your app really doesn’t have much space to display a name, and if the name is too long, it’ll be truncated and end with ellipses. You can usually get 12 characters or so, but your app name might be longer.

Luckily the display name shown for your app on the SpringBoard can be different from the one shown on the store, so you can shorten it to fit. To change this, flip to the Source tab and you’ll see a couple of values—Bundle Display Name and Bundle Name. Bundle Name is the name of your app on the app store, and Bundle Display Name is the name on the SpringBoard. You could set your bundle name to “SquareRt Square Root Calculator”, and set the display name to “SquareRt”. Be aware, though, that if you update the Application Name on the Application tab, both of these values will be updated to match.
**Bundle Identifier**

The bundle identifier is a unique identifier for your app, and it’s essentially the same as the Android package name—even down to the convention of using a reversed domain name suffixed with the app name. Once it’s set on the app store, the bundle identifier can’t be changed, and ideally you won’t want to ever change it. Your app is signed with a certificate and a private key based on this bundle identifier, so if you change the identifier, the signing profiles won’t work and will need to be re-created. We’ll walk through doing this in chapter 13, but be warned, it’s not a nice process!

**Version**

Version is a string representation of a three-part version number (usually something like major.minor.revision, such as 1.0.4) and it must be incremented each time you update the app on the store. This is the public version number that’s shown on the store.

**Build**

Confusingly, iOS has a second one-, two-, or three-part version number that’s used to define the build number. This is an internal build number, so it’s not shown in iTunes, but it is used to determine if the app has been updated—just like the version code on Android. For example, if you’re working on a release you want to publicly call 2.5.3, you’d submit an app with the version set to 2.5.3 and the build set to anything you like, such as 1. If this version gets rejected by Apple, you’d fix the issue and then upload a build with the same version (2.5.3) but a different build (such as 2) so that the public version stays the same but iTunes will know that you’ve submitted a new version.

Although this build number can consist of up to three parts, it’s usually simpler to use a single build number and increment it with every build.

**Devices**

This field allows you to chose the type of device you want to target—iPhone, iPad, or both. This means that you can target a particular platform—for example, if you’re building an app that only makes sense on a larger device, you can limit it just to iPad.

**Device Orientations**

All mobile devices can be easily rotated, and a good app should work well in all orientations or it should be locked to one orientation (something you see a lot in games—they only work in landscape). By ticking the different boxes, you can choose which orientations to support.

### 6.6 SDK versions

Every year at Apple’s WWDC (Worldwide Developers Conference) the senior VPs at Apple unveil the new and awesome features of the next version of iOS—the operating system for iPhones, iPads, and iPod Touches. The iPhone has been around for ten years and has seen eleven different versions of the OS in that time, going from a game-changing phone to a pocket supercomputer. The same is true for Android—fourteen versions of the OS in eight years. Currently the operating systems have a major update at least every year (more often for minor updates), and each update
brings a whole range of new APIs that you can use and deprecates older ones. Each OS release comes with a newer version of the SDK providing these APIs, so new OS releases are often referred to as new SDKs. This is different from the previous OS models that C# developers would be used to—new Windows versions come out every few years, and updates to the .NET Framework are also few and far between (although this is a model that’s changing, with Windows 10 having regular updates).

As a developer, you want to use the latest features where possible, but you still want to support older devices potentially running older OS versions. Supporting older OS versions is less of a concern on iOS, where within weeks of a new OS being released, the majority of users update, but it’s a big concern on Android. When Google released Android, it was open source, so device manufacturers added their own features to the OS before passing it to the carriers who also sometimes added their own features. This means that when Google releases a new version of Android, or even a security patch, not every device can install the update straightaway. Instead they have to wait for the manufacturer to update their version, and possibly for the carrier to issue an update as well. For new devices this does happen, but for older devices that are no longer made, the updates may never be available. This results in the Android ecosystem being particularly fragmented.

iOS 10 was released in September 2016, and by November it was on 79% of devices (figure 6.22), iOS 9 was on 17%, and the remaining 4% were on older OS versions (data from Mixpanel, https://mixpanel.com/trends/). This means most app developers can target the most recent two versions (iOS 9 and 10 at the time of writing), and not worry about their app working on earlier versions.

On the other side of the mobile fence, the picture is not so rosy—Android Nougat has been out for the same length of time but is on less than 1% of devices (figure 6.23), with the majority being on Marshmallow, Lollipop, and even 19% of users on KitKat. Not only are users on older versions, but most of these users won’t be able to upgrade—for example, I have a two-year-old tablet purchased from a major retailer in

![Figure 6.22](image1.png)  
**Figure 6.22** iOS users upgrade often, with 79% of users being on the latest iOS version two months after launch.

![Figure 6.23](image2.png)  
**Figure 6.23** Android users don’t (or can’t) upgrade as often as iOS users, with 1% being on the latest Android version two months after launch.
the UK that’s running Android Lollipop and will never be updated. This means that, as cross-platform mobile developers, we need to support a lot more versions of Android than we do of iOS.

6.6.1 Android SDK versions and the SDK manager

When you install Xamarin, it will also install the Android SDK for you. These are the libraries and tools used by Xamarin to compile Android apps, and they’re the same tools that the native development IDE (Android Studio) uses.

You can see what’s installed by going to Tools > SDK Manager from Visual Studio for Mac, or going to Tools > Android > Android SDK Manager from Visual Studio in Windows. This will load the Android SDK Manager, showing what versions of the Android tools and SDKs are installed, as well as the images for creating Android emulators. From here you can download new SDKs, download new emulator images, and update the versions of the build tools.

One of the downsides of Xamarin development is that there are a lot of moving pieces, some controlled by Xamarin, and others not (such as the Android SDKs). This means it’s easy to get weird errors just by using combinations of the different tools that don’t quite work together. As a general rule, I find it better to keep the SDK up to date with the latest stable version.

Android SDKs are referred to in three different ways—by version number, by API level (which can cover multiple version numbers), or by alphabetical confectionary-based nickname (with some names covering multiple API levels). This is as confusing as it sounds, and developers will mix and match their terminology. Google only supports (as in, provides security patches for) KitKat and above, and at the time of writing the latest version generally available is Nougat (with Oreo being rolled out to a limited set of Google devices). Table 6.1 shows how the names match up to API levels and to versions for the most recent versions.

Table 6.1 The different Android code names, API levels, and versions for the most recent and popular versions

<table>
<thead>
<tr>
<th>Name</th>
<th>API level</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jelly Bean</td>
<td>16–18</td>
<td>4.1–4.3.1</td>
</tr>
<tr>
<td>KitKat</td>
<td>19</td>
<td>4.4–4.4.4</td>
</tr>
<tr>
<td>Lollipop</td>
<td>21–22</td>
<td>5.0–5.1.1</td>
</tr>
<tr>
<td>Marshmallow</td>
<td>23</td>
<td>6.0–6.0.1</td>
</tr>
<tr>
<td>Nougat</td>
<td>24–25</td>
<td>7.0–7.1.1</td>
</tr>
<tr>
<td>Oreo</td>
<td>26</td>
<td>8.0</td>
</tr>
</tbody>
</table>
**IMPROVEMENTS WITH APPCOMPAT AND GOOGLE PLAY SERVICES**  It’s not all bad in the Android world. Google is working to back-port new features to older Android versions using a thing called AppCompat (providing libraries for using newer features on old OS versions) and by moving out a lot of the core APIs into a set of Google services called Google Play Services. This means you can still access newer features on older devices. This will be covered more in chapter 9.

**SETTING THE ANDROID SDK VERSION FOR THE APP**

As already mentioned, the APIs available to Android developers change over time. Nothing is ever deleted; instead, out-of-date APIs are marked as obsolete and new APIs are added. For example, Android has a text-to-speech class, `TextToSpeech` ([http://mng.bz/T5u2](http://mng.bz/T5u2)). This has a method on it called `Speak` with two overrides. One override was added in API 21, and the other was deprecated as of API 21. Not only was it deprecated, but it also no longer works on devices running Lollipop (API 21) or above.

When you build a Xamarin Android app, you can choose three different Android SDK versions—the one to build against, the minimum your app should support, and the target version that your app is intended to run against:

- The minimum API is used at install time—the Google Play store won’t let users install an app that has the minimum set to a version higher than the device is using.
- The build version is the SDK that’s used when compiling, so you can only use APIs that are available in that version.
- The target version is used at runtime to ensure that everything works smoothly.

With the `TextToSpeech` API, if you wanted to use it in an app that supports KitKat and above, you’d need to set your minimum version to API 19, and then compile against a later version. This way your app will run on any device with KitKat and above, but you’d be able to call both APIs. Xamarin only binds the libraries from API 15 and above—you can target older versions if you want, and your app should run, but the compiler won’t check that APIs that don’t exist on those versions aren’t called.

Obviously there’s a problem here—there are two different overrides of a method: one that only runs on APIs 19 and 20, and one that only runs on APIs 21 and above. What can you do? First, when a method is deprecated, it’s marked with the C# `Obsolete` attribute, so if you’re compiling against a later SDK, you’ll get a compiler warning if you call this method. This can really help you see what’s no longer available, and this is a good reason to have warnings set to errors on your release builds! Second, you can query the SDK version at runtime and call different code depending on which OS version you’re running against. The following listing shows an example of this.
Listing 6.2 Checking the current Android SDK

```java
if (Android.OS.Build.VERSION.SdkInt >= Android.OS.BuildVersionCodes.Lollipop)
{
    // Do things the Lollipop way
}
else
{
    // Do things the pre-Lollipop way
}
```

A check is made against the current OS build, and if it’s Lollipop or later, one code branch is run. If not, another is run. In the Lollipop and higher branch, the new `Speak` method is called; in the branch for versions prior to Lollipop, the old override can be called (and wrapped in an appropriate `#pragma` directive to suppress the warning).

**Warnings should be errors in release builds**

Warnings are the compiler’s way of suggesting something might be wrong, such as a field being declared but not used, a variable being compared to itself in an `if` statement, or a deprecated API being used.

To keep your code clean, it’s advisable to set all warnings as errors for your release builds so that you can’t compile an app for release without fixing all warnings. You can set this by ticking Treat Warnings as Errors in the Compile tab of the project Properties tab (project Options dialog for Visual Studio for Mac) for release builds. For debug builds, this doesn’t matter so much, because it’s easy to get warnings during development that you’ll clean up once your code is ready.

If you get a warning that you want to ignore instead of having it as an error (such as calling an obsolete API inside an SDK version check), you can wrap the offending line of code in a directive to tell the compiler to ignore it:

```csharp
#pragma warning disable
// call obsolete code here
#pragma warning restore
```

You can read more about this in Microsoft’s C# Guide, at http://mng.bz/8f0K.

Let’s look at this example in the context of the three different versions:

- **Target framework**—This is the version of the Android SDK you’re compiling against. You can only use APIs that are available in this version. If you use an API that wasn’t introduced until a later version, your app won’t compile. Normally this is set to Use Latest Installed Platform, which means that it will compile against the latest version that’s installed from the Android SDK manager.
Minimum Android version—This is the lowest Android version that your app will support. Your app won’t be available to devices with a lower version, so those users won’t be able to install it from the Google Play store. This doesn’t mean that there are any compiler checks to ensure it will work—you can have your target as a later version and call APIs that aren’t available in the minimum version. In this case your app will crash, so you need to make sure that if you call any newer APIs, you use runtime SDK checks, like the one shown in listing 6.2.

Target Android version—This is the version of the SDK that you’ve tested your application against. This tells the Android OS not to enable any compatibility behaviors to help your app work. This is outside the scope of this book, and it’s easiest to leave this as Automatic, to use the same version as the target framework.

Setting the SDK versions using Visual Studio for Mac
The target framework is a compiler setting, so this is set in the project options. You can access this by double-clicking on your project in the Solution Pad (SquareRt.Droid or Countr.Droid for this chapter’s examples), or by right-clicking the project, selecting Options, and then selecting the General tab from the Build section (figure 6.24).

![Setting the SDK versions using Visual Studio for Mac](image)

Figure 6.24 The SDK to compile against can be selected here. It’s usually good to leave it as Use Latest Installed Platform.

The minimum Android version and target SDK version aren’t used at compile time. Instead, they’re checked at install time and runtime, which means they need to be set in the Android manifest file (figure 6.25).

The two settings we care about in this file are the Minimum Android Version and Target Android Version. The minimum should be set to the lowest version you want to support. Unless there’s a particular API your app needs to use, I recommend setting this to KitKat to target the most devices. This is the default setting for new Xamarin Android apps. The target should be left as Automatic to use the latest installed platform.

Setting the SDK versions using Visual Studio for Windows
Unlike Visual Studio for Mac, Windows Visual Studio doesn’t have an explicit editor for the AndroidManifest.xml file. Instead you can set the SDK versions from the project...
SDK versions

Properties tab (figure 6.26). Open these by right-clicking the project in the Solution Explorer (SquareRt.Droid or Countr.Droid in this chapter) and selecting Properties, or by selecting the project and pressing Alt-Enter. From the properties, select the Application tab to configure the Target Framework to compile against, and select the Android Manifest tab to configure the minimum and target SDK versions.

Running against a particular SDK version

When Xamarin is installed, it will install the Android SDK for you. In the process, it will install some system images for different Android devices and create emulators for them using a version of the SDK that may not be the latest. The installed version may vary over time, so it’s always worth installing the latest available stable SDK (at the time of writing this is 26—Oreo). Each year Google will roll out a new version starting with a beta version of the SDK, so unless you need to test against this beta, you should avoid installing and using it.

At the time of writing, Xamarin creates emulators running Marshmallow (API 23). This means that when you run your Android app, you can run it on an emulator targeting Marshmallow (figure 6.27)—you already did this back in chapter 2 when testing out the Hello Cross-Platform World app.
You can find more information on configuring emulators in the “Android SDK Emulator” section of the Xamarin documentation, including how to create emulators using different versions of the SDK (http://mng.bz/4CZX).
6.6.2 iOS SDK versions

Setting iOS SDK versions is much, much simpler than Android. For starters, the OS names match the SDK versions, and there’s only one version number, not a version number, API level, and highly sugary codename. When you build iOS apps, you always build against the latest SDK; the only option you have is to choose the minimum version that your app will support. On iOS you only need to support two versions—the latest and the previous. This will cover a large proportion of the iOS user base. This is helped by the fact that Apple supplies the latest SDK version and will only accept submissions of apps to the store using a recent version of the SDK.

Where Android has an SDK manager to allow you to install multiple versions of the Android SDK, iOS has a much simpler model. Apple always wants you to use the latest SDK version, and that’s pretty much all you can install. Every time you update Xcode (Apple’s development environment that contains the tools Xamarin needs to build iOS apps) you always get the latest SDK to compile against, and the macOS App Store will always try to keep you on the latest version of Xcode. The way to compile against older versions of the SDK is to install older versions of Xcode, something that Apple doesn’t support.

That’s compiling taken care of—you always compile against the latest version of the SDK. As with Android, though, your apps can run on older versions of the SDK. You can control the minimum version that’s supported, and you can check at runtime what OS your app is running on and call the relevant APIs.

Setting the minimum supported SDK

To control the minimum supported version, you can set the Deployment Target in your iOS app’s info.plist file. This file lives in the root of the iOS app, and if you double-click it, it will open in a property editor. You can edit the raw XML if you want, but it’s a complicated schema, so it’s easier to use the property editor.

The field of interest here is the Deployment Target. From this drop-down list you can choose the minimum iOS version to support (figure 6.28). Once this is set, your app will only be able to be installed and run on devices running that version of iOS or higher. Users on a lower version won’t be able to install your app.

As with Android, APIs don’t go away when a new version of the OS SDK is released—they’re always available, but they’re deprecated when they’re no longer supported. Again, as with Android, you can see if you’re using a deprecated API by checking for compiler warnings. You can also check the OS version at runtime and call the relevant version of the API depending on what OS version your app is running under, as shown in listing 6.3.
if (UIDevice.CurrentDevice.CheckSystemVersion (9,0)) {
    // Code that uses features from Xamarin.iOS 9.0 and later
} else {
    // Code to support earlier iOS versions
}

**Listing 6.3 Checking the current iOS SDK**

Check to see if the current OS is iOS 9 or later

**RUNNING AGAINST THE iOS SDK**

When you launch your iOS app against the simulator, the simulator will default to using the latest iOS version you have installed. This is because Xcode always installs the latest versions of the SDK only, by default. If you want to test on earlier versions, you can download simulators using Xcode, but you’re limited in how far back you can go—at the time of writing, iOS 10 has been out for a couple of months, and the oldest simulator you can install is one running iOS 8.1. To install older versions from Xcode, go to Xcode > Preferences and select the Components tab. This is shown in figure 6.29.
You can also test against older versions using physical devices, but you’ll need to have these devices already configured with the OS you want to use, as Apple doesn’t make older versions of the OS available to download.

6.7 Linking

Mobile apps run on pretty constrained hardware. Mobile devices have less power, less memory, less storage, and unreliable networking. This means your mobile apps need to be optimized for a mobile environment: they need to be small so they don’t eat up space (important on a device with only 16 GB of storage) and so they can be downloaded over a cellular connection without eating into users’ data plans too much.

For desktop developers, app size isn’t normally a consideration, but for mobile it’s important, especially as there isn’t a .NET Framework available in the OS like there is on Windows. Instead, your apps must ship everything they need to run, all self-contained in one package—be it your code, NuGet packages, or the relevant bits of the .NET Framework. This means your apps could potentially be huge. They could take up a lot of space on the device (which could be a problem with a bottom-of-the-range
device with only 8 or 16 GB of storage), and they could be too large to download over a cellular connection, reducing the chance of users installing your app (for example, Apple won’t allow users to download apps over 100 MB via cellular connections).

Luckily you can make your apps considerably smaller with the help of linking.

### 6.7.1 Linking the apps

Our coffee shop, from examples in previous chapters, has been particularly successful, and it’s time to move to larger premises. Moving is hard work, so we’ll get the professionals involved. When our coffee shop moves, not everything comes with it—we only want to move the things we need to reestablish our coffee shop elsewhere. We can leave behind the blinds, the carpets, and the shelves. We need to decide which things we really need and provide a list to the movers, so they take what’s needed and leave the rest behind. This way the moving truck can be smaller, and our moving costs are reduced.

This is something we also want to do to our code. When we build our apps and ship them to our users, such as via an app store, we want our apps to be as small as possible, making them quicker to download and install—something that’s very important for users who have expensive mobile data plans with no access to WiFi. Just like when we move our coffee shop, we want to package up what’s needed for our app to run and no more. We can do this using the linker.

The Xamarin tooling contains a linker that’s run on your code automatically after compilation. The linker looks at the code you use and bundles that into the final app, stripping out everything that’s not used (figure 6.30). It does this on a method, property, field, and event basis, so even if you use `string` in multiple places, if you never use the `Substring` method, the linker can strip out that one method and leave the methods you do use.

**Figure 6.30** The linker strips out any code that’s not explicitly used.
Linking is configurable and can be turned off. It can just be used on the SDK to strip out unused code from the OS SDK and relevant .NET Framework, or it can be used everywhere to strip unused code from your assemblies and any NuGet packages.

This is a common concept with languages like C++, but it’s not used with C#—there’s no SDK to strip out because the .NET Framework is part of the OS, and desktop PCs don’t have the hardware constraints of mobile devices.

### 6.7.2 Linker options

The linker can be configured on the iOS and Android app projects—it’s relevant for your final apps, so it’s not something that can be configured on class libraries.

For Android, to configure the linker in Visual Studio for Mac, go to the project options by double-clicking the Android app project in the Solution Pad, or by right-clicking it and selecting Options. From there, select the Android Build tab on the left, and then select the Linker tab in the right pane.

On Windows go to the project properties by right-clicking the project in the Solution Explorer and selecting Properties, or by selecting it and pressing Alt-Enter. From the Properties tab, select the Android Options tab on the left, and then select the Linker tab in the right pane (figures 6.31 and 6.32).

![Android linker settings](image)
For iOS you can configure the linker from the iOS Build tab of the project settings (figure 6.33).

The setting we’re interested in here is called Linker Behavior (Mac) or Linking (Windows), and it has three settings, available on a per-configuration (for example, Debug or Release) basis:

- **Don’t link**—Don’t do any linking, leaving everything in place. This is the default setting for debug builds and it leads to large apps but faster build times. This isn’t recommended for release builds.

**IF YOUR BREAKPOINTS AREN’T BEING HIT, CHANGE THE LINKER SETTINGS** There’s a known issue at the time of writing with Xamarin Android apps where if you use Don’t Link, sometimes your breakpoints won’t get hit when debugging. If this happens to you, change the linker settings to Link Framework SDKs Only.

- **Link Framework SDKs Only (Link SDK Assemblies Only on Visual Studio for Mac)**—This setting will perform linking on all the assemblies provided by the .NET Framework and Xamarin SDKs. It won’t link any of your code, or any NuGet packages or external code you use. This is pretty safe, as it’s unlikely you’ll be
accessing framework SDKs via reflection, and it removes most of the code you won’t be using, leading to small final app sizes.

- **Link all**—This setting will run the linker over everything—your own code, NuGet packages you use, and all the framework SDKs. This provides the smallest final packages but it risks removing things you’ll need if you do any reflection. It’s the preferred option for release builds, but you’ll need to thoroughly test your app to make sure nothing is stripped out that’s needed.

### 6.7.3 Stopping the linker from doing too much

When our coffee shop moves, we have to tell the movers what to move. When doing this it’s easy to miss something—we could tell them to move a coffee machine, but neglect to tell them to move the power leads and pipes. Similarly, when we link, the linker relies on explicit calls to public methods, properties, and events to know what to keep. It’s easy to use something without an explicit call, and the linker could strip that out, leading to a crash at runtime. The usual culprit for this is reflection—where we find a property or method based on its name and invoke it. Unfortunately, for developers who use MVVM, reflection is used a lot. You can bind a property by name, and this can be the only reference to it. The linker looks for references, doesn’t understand that the string representation is a reference to the property, and removes it.

Fortunately you can control the linker using a couple of techniques:

- Explicitly use the public property, method, or event somewhere
- Use the Preserve attribute

**Explicitly use the public property, method, or event**

By explicitly using the property or method, the linker will see the usage and will leave it in. This doesn’t have to be functional code, just a reference somewhere. MvvmCross uses this technique. It ensures code isn’t stripped out by the linker by using a file called `LinkerPleaseInclude.cs` containing a class that uses code that would be referenced by reflection. If you look in the root of the iOS and Android projects, you’ll see this file (figure 6.34).

**Use the Preserve attribute**

The Preserve attribute can be added to a class, or to the members on a class, to tell the linker to not strip out code. If you set this at the class level, you need to set the AllMembers property to `true` to ensure that all members are preserved. The following listing shows this in action.
Unfortunately, this isn’t an attribute that’s available to .NET Standard libraries. Instead, there are two versions of this attribute—one on iOS (Xamarin.iOS.Foundation.Preserve) and one on Android (Android.Runtime.Preserve). To use this in a .NET Standard library, you’ll need to define the attribute yourself. When you link an iOS app, the linker won’t strip out anything with an attribute called Preserve on it,
regardless of the namespace of that attribute. On Android, it specifically looks for an attribute in the Android.Runtime namespace called Preserve.

The simplest way to preserve code in a .NET Standard library is to define Android.Runtime.Preserve yourself in your core project and use that—the namespace matches, so the Android linker will use it, and the name matches, so the iOS linker will use it. This is, unfortunately, over-complicated, so hopefully Xamarin will improve on this in the future. The following listing shows an example implementation.

```csharp
namespace Android.Runtime
{
    public sealed class PreserveAttribute : System.Attribute
    {
        public bool AllMembers;
    }
}
```

You can find more information about linking in the Xamarin developer docs:
- Linking on iOS—http://mng.bz/d55a
- Linking on Android—http://mng.bz/v7x1

You now have your solutions at the ready, you’ve worked out what code you need in which layer and what code should run in the background and UI threads. You’ve also seen some of the new features of Xamarin iOS and Android apps. Now you’re ready to start coding the app proper. In the next chapter we’ll dive right into the core project and start writing the cross-platform models and view models.

**Summary**

In this chapter you learned

- iOS apps are different from Android apps, so you should think about your UI in terms of the platform your app is running on.
- By thinking about the user flows up front, you can start to build up a picture of the classes you’ll need and what threads your code can run on.
- Unlike other C# apps, iOS and Android have OSs and SDKs that change regularly, so you need to code for different OS versions if you want to use the latest features.
- iOS users mainly use the latest two OS versions, whereas Android users have a wide range of OS versions.
- Linking reduces your app size, but it can cause problems with code that’s not explicitly used but instead is accessed via reflection.
- Mobile apps are shipped with a properties file that provides information on your app to the relevant app store and OS.

You also learned how to

- Map your user flows to the different layers of MVVM.
- Map your user flows to different threads.
- Configure your app’s properties.
- Select appropriate SDK versions for compiling and running your app.
- Configure linking to ensure your app is as small as possible, while not removing any code you need.
Rewriting the same app for iOS and Android is tedious, error-prone, and expensive. Microsoft’s Xamarin drastically reduces dev time by reusing most application code—typically 70% or more. The core of your iOS and Android app is shared; you write platform-specific code only for the UI layer. And because Xamarin uses C#, your apps benefit from everything this modern language and the .NET ecosystem have to offer.

**Xamarin in Action** teaches you to build cross-platform mobile apps using Xamarin and C#. You’ll explore all the layers of a Xamarin app, from design to deployment. Xamarin expert Jim Bennett teaches you design practices that maximize code reuse and isolate device-specific code, making it a snap to incorporate the unique features of each OS.

**What’s Inside**

- Understanding MVVM to maximize code reuse and testability
- Creating cross-platform model and UI logic layers
- Building device-specific UIs
- Unit and automated UI testing
- Preparing apps for publication with user tracking and crash analytics

Readers should have some experience with C#. Mobile development experience is helpful, but not assumed.

**Jim Bennett** is a Senior Cloud Developer Advocate at Microsoft, specializing in Xamarin mobile apps.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/xamarin-in-action