Technology trends come and go, but few have generated the excitement, momentum, or long-term impact that mobile computing has. Mobile computing impacts people's lives at work and at home on a daily basis. Many companies and individual developers are looking to become a part of the movement but are unsure how to best utilize their existing skills and assets. The Xamarin suite of products provides new opportunities to those who already have a significant investment in C# development skills and .NET code bases, and would like to enter into this new, exciting world.

This example-oriented guide provides a practical approach to quickly learn the fundamentals of Android app development using C# and Xamarin.Android. It will lead you through building an Android app step-by-step with steadily increasing complexity before finally deploying it to the various Android app stores.

Who this book is written for

If you are a C# developer who wants to develop Android apps and enhance your existing skill set, then this book is ideal for you. A good working knowledge of C#, .NET, and object-oriented software development is assumed.

What you will learn from this book

- Build a multi-view, orientation-aware Android application using the LinearLayout, RelativeLayout, and TableLayout managers
- Use a ListView and ListAdapter to build collection view from server data
- Consume REST web services to perform the GET, UPDATE, and DELETE operations
- Use Fragment, ListFragment, and DialogFragment to make the app compatible for Android tablets
- Use Android SQLite for data persistence and caching
- Use device camera to capture photo and upload to server using HTTP multipart uploading
- Capture the current location of a device using GPS, determine the street address, and integrate with the map app
- Test, debug, and deploy an Android app

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Creating the Points Of Interest App'
- A synopsis of the book’s content
- More information on Xamarin Mobile Application Development for Android Second Edition
About the Author

Nilanchala Panigrahy is a software developer, community speaker, and thought leader in the fusion of design and mobile technologies. Currently, he is employed as a lead application developer by Ness Technologies (http://www.ness-ses.com), a leading global technology services provider. He has over 8 years of experience in developing rich mobile applications in Java, C#, Android, Xamarin, HTML5, and PhoneGap.

Over the years, he has published 20 plus Android applications in the consumer and enterprise spaces and authored over 200 articles/tutorials on Android and Xamarin application development, which are freely available on the Web.

You can find more information about him on his blog at http://javatechig.com or contact him via his Twitter handle at, @npanigrahy.
Preface

Xamarin is built on top of Mono, an open source version of the .NET framework, based on the ECMA standards. Xamarin brings you a set of tools that includes its own C# compiler and a Common Language Runtime (CLR). The Mono framework source project is maintained by Xamarin, a San Francisco-based company (formerly by Novell and originally by Ximian). The prime intention of the Mono project was to make the .NET platform compatible with other non-Windows platforms such as Linux.

After Attachmate acquired Novell in April 2011, the future of the Mono platform was pushed into dark. A few months later, Miguel de Icaza, a former Novell employee, founded a company called Xamarin and declared to continue using the Mono platform for commercial software development. Since then, Xamarin has sponsored the Mono open source platform development and provided the commercial .NET stacks for both the iOS and Android platforms. The .NET for iOS is called MonoTouch, or Xamarin.iOS, and .NET for Android is called Mono for Android, or Xamarin.Android.

Xamarin frameworks enable developers to write cross-platform mobile applications targeting different platforms, including iOS, Android, and Windows Phone. Using Xamarin, you can develop a pure naive Android or iOS application using the C# programming language and share the application logic between the different platforms. This results in a faster development cycle and developers can leverage the existing C# and .NET programming skills, which helps reduce the learning curve to develop the mobile application.

This book is structured in a logical sequence to help C# and .NET developers to build Xamarin.Android applications from the ground up. It explains the widely used basic and advanced Android concepts, including a user interface, data storage, consuming web services, geolocation, map, camera, and the build distribution process.
Preface

This book provides the most comprehensive explanations of the basic and advanced Xamarin.Android concepts; you can precisely build with practical live examples to develop a complete working application. Over the course of this book, you will build a single application, the POIApp. With this application, we will cover all the fundamentals of Xamarin.Android to help you get started with your own application development.

What this book covers

Chapter 1, The Anatomy of an Android App, provides an overview of the Android platform and what Android apps are composed of.

Chapter 2, The Xamarin.Android Architecture, provides an overview of the Xamarin platforms and describes how the Mono and Android runtime work together in order to allow developers to build Android apps using C#.

Chapter 3, Creating the Points of Interest App, walks you through how to set up the development environment, create a new Xamarin.Android app, and run the app in the Android emulator.

Chapter 4, Adding a ListView, describes Android's AdapterView architecture and walks you through how to use ListView and create a custom adapter. This chapter also covers how to download the data asynchronously from a web service and display the response on a custom ListView.

Chapter 5, Adding a Details View, walks you through how to create a details view to display the details of POIApp, add navigation from ListView, and add actions to perform the save and delete web service operations.

Chapter 6, Making Your App Orientation-aware, walks you through how to detect the device orientation and handle application behaviors on the configuration changes.

Chapter 7, Designing for Multiple Screen Sizes, introduces you to the Android fragments and the different techniques used for managing the resources and layout to support multiple screen sizes, including Android tablets.

Chapter 8, Creating Data Storage Mechanisms, discusses a number of available data storage options in Xamarin.Android and stores the Point of Interest list fetched from a web service using the SQLite database engine to make the list accessible when a device is offline.
Chapter 9, Making POIApp Location Aware, discusses the various options that developers have in order to make their apps location-aware, and this chapter also covers how to add logic to determine a device's location, the address of a location, and display a location in the map app.

Chapter 10, Adding the Camera App Integration, discusses the various options for integrating with a device camera to capture a picture of POIApp and upload the captured photo to a web service using the HTTP multipart form upload.

Chapter 11, Publishing an App to the App Store, discusses the various options for distributing Android apps and also covers how to prepare a Xamarin.Android app for distribution.
Creating the Points Of Interest App

In this chapter, we will move to the practical side of creating an app and cover the facilities that Xamarin.Android provides developers for creating, executing, and debugging applications. This chapter covers the following topics:

- An overview of the sample app
- Installing and configuring Xamarin.Android
- Creating the sample app
- Running and debugging apps

The sample POIApp

In this chapter, we will begin by building a sample Points Of Interest (POIApp) app that will be completed through the remaining chapters of this book. This app will allow the user to capture, save, and manage the POIApp and it supports the following features:

- It captures information about the POIApp, including name, description, address, latitude, longitude, and photos
- It captures the address, latitude, and longitude of the POIApp using the devices location capabilities
- It captures and saves a photo of the POIApp using the devices camera
- It saves details of the POIApp on the Cloud
- It fetches and displays a list of the POIApp from the Cloud
- It stores/caches the POIApp for offline viewing
Installing Xamarin.Android

Before we move on, we need to get Xamarin.Android installed. This section walks you through installing the latest Xamarin development platform (Xamarin.Android Version 5.1.3 and Xamarin Studio Version 5.9.3) on the Mac operating system.

At the time of writing this book, the installation instructions presented in this chapter are accurate. However, the tools are being updated rapidly, so these instructions may be out of date by the time you read this. You can refer to the official Xamarin website for the updated installation instructions.

You may choose to install Xamarin.Android and work through the samples on Windows OS; in which case, you will encounter only minor deviations in some of the directions. If you choose to install on the Windows operating system, you may refer to the installation guide from the official Xamarin developer portal at http://developer.xamarin.com/guides/android/getting_started/installation/windows/.

To install Xamarin.Android, perform the following steps:

1. The Xamarin Studio IDE and Xamarin.Android platforms are the prerequisites to develop an Android application using Xamarin. Go to the download section of http://xamarin.com/, fill in your personal details, download the unified installer for your operating system, and launch it:
2. Click on the installer page and the agreement page until you come to the **Product Selection** page. The installer allows the installation of both Xamarin. Android and Xamarin.iOS, as shown in the following screenshot:

3. Xamarin.iOS will not be needed for the exercises in this book; you may uncheck Xamarin.iOS for the installer window and click on **Continue**.

4. Now the installer will proceed to configure the installation directory. You may either change the installation location on your computer, or continue with the installation:
5. Next, the prerequisites for the installation will be listed, as shown in the following screenshot. It requires installing the Mono framework, the Android SDK, Xamarin Studio, and the Xamarin.Android component. Click on **Continue** to proceed with the installation:

![Xamarin Installer Prerequisites](image)

6. You will now see the Android SDK agreement page. Just accept and click on **Continue** for installation. An installation progress page will be displayed depicting the component installation in progress:

![Xamarin Installer Installation in progress](image)
During installation of Xamarin.Android, if you're encountering Android SDK download issues, please ensure that downloads are not blocked by your corporate network or firewall. For such events, you can download and install the standalone Android SDK from https://developer.android.com/sdk/index.html. Once Android SDK is installed on your computer, you can resume with Xamarin package installer.

7. As each component is installed, a check mark will be placed next to the component, and once all the items are installed, a final installation completion page will be displayed.

8. Now click on **Launch Xamarin Studio**: 

![Xamarin Installer](image)

### Installing platforms and tools

The Android platform installation comes with the Android SDK manager toolkit that allows you to select and download the tools and platforms that are required to build the Android apps. For example, when a new version of Android is released, you can test your application against the new platform by downloading the new bundles using the SDK manager.
The **Android SDK Manager** can be launched from Xamarin Studio by navigating to the **Tools | Open Android SDK Manager...** option:

Notice that the Android SDK Manager has now opened and it lists the following options for you to install or delete.

### Tools

The **Tools** section is one of the core parts of the Android installation. This is required to build, install, and debug the Android application on the device:
A brief description of the tools is as follows:

- **Android SDK Tools**: These are the development tools that must be installed to compile Xamarin.Android applications. When you install the Xamarin platform, it automatically gets installed; however, you may need to keep this updated.

- **Android SDK Platform-tools**: The SDK platform tools are required to connect the devices to deploy builds and debug applications. You should always have the latest version of the platform tools to ensure compatibility with the latest version of the Android APIs; so keep this up to date as well.

- **Android SDK Build-tools**: Build tools are required to compile the source code to the application that can be run on the Android devices or emulators. By default, Xamarin Studio installs the most recent version of the build tools. It is always recommended that you install the latest version of the SDK build tools as well.

For more information, you may visit the following official Android documentation:


### The Android platform APIs

Each successive release of Android operating system also includes the Android framework API for developers to take advantage of the set of new packages, classes to build apps. The platform API section lists all the Android API versions along with the API levels:
A brief description of the various platforms is as follows:

- **SDK Platform**: The SDK Platform for a given API level allows you to compile against that version of Android. When a new platform is released, it requires the newer version of the SDK Platform tools and SDK tools; and hence, you need to keep those up to date.

- **Samples for SDK**: These are Android sample applications for each API level developed using Java. They are not required by Xamarin developers. You don't need to install these samples unless you have a specific need or if you want to analyze and port the Java samples to C#.

- **System Images**: System images are used with **Android Virtual Device** (AVD). All the recent Android versions include both the ARM and x86 system images. The x86 images run significantly faster and are preferred over the ARM. Notice that some of the system images are prefixed with **Google APIs**. They include the Google Play services runtime and are useful for testing the apps that use Google Play services features such as map, in-app purchases, and so on.

The **Android platform Extras**

The **Extras** section includes some of the additional optional tools you may require during development, which are shown in the following screenshot:

![Extras Section Screenshot](image-url)

The following section will brief you on some of the important sections of the Android platform extra tools:

- **Android Support Library**: The **Android Support Library** is a set of code libraries that are used to provide backward compatibility for new API features on older devices. It is considered as a best practice to use the support libraries while developing some of the features, as it makes the app compatible with the older versions of devices.
• **Google Play services**: The Google Play services runtime provides the set of APIs to develop certain Android features such as Google Maps, integrating with Google+, Google Play subscription, and so on. All the Android devices include the Google Play service runtime. However, the Android emulators don't include the Google Play services runtime by default but it can be installed separately.

• **Google USB Driver**: If you are running Windows OS, you will need to install these drivers to enable USB debugging for Android devices. For some devices, you may also need to install the specific device driver's software provided by the device manufacturers. You don't need to install this driver if you are using Mac OS X for development.

## Creating Android Virtual Devices

The Android emulator, also known as **Android Virtual Device (AVD)**, is used to test the Android application in the absence of a device. Emulators get installed as part of the Xamarin installation. You can create your own emulator or customize an existing one with the required device configurations for the device being emulated.

In order to create or modify an existing emulator, perform the following steps:

1. From the main menu bar, navigate to **Tools** and open **Google Emulator Manager**. It will open the **Android Virtual Device Manager** window:
2. To create a new emulator, click on the **Create** button, on the right-hand side panel. Provide the configurations, such as **AVD Name**, **Device**, **Target**, **Memory Options**, and so on.

3. To edit an existing emulator, select the emulator you wish to edit and click on the **Edit** button:

4. Note the **Target** setting; this specifies the version of the Android platform and the API level that will be used for the emulator.

5. Select the device skin from the **Device** field drop-down menu. In my case, I have selected **Nexus 5**.
6. Set the Target field to Android 4.4.2. If needed, uncheck the Hardware keyboard present option, and click on OK.

7. Check the Use Host GPU option. This option makes the emulator use the host computer's OpenGL implementation, which makes the rendering significantly faster.

8. You may select the Snapshot option to speed up the emulator's boot time. When this option is enabled, it saves a snapshot of its RAM on the first boot and restores from that snapshot on future use. You cannot have both the Snapshot and Use Host GPU options enabled at the same time.

Cloning a virtual device

There are many options that can be modified in order to emulate any device and configuration needed. The Android Virtual Device Manager dialog box also has a tab named Device Definitions that can be used to set up the devices that are available when configuring an AVD. The following screenshot shows you what can be configured as part of the device definition:
Speeding up the Android emulator
The default Android emulator that comes with the Android development toolkit is quite sluggish. It is often disappointing and painful to test an application in the Android device emulators. The wiser choice for developers is to have a real Android device to test on. However, because the Android ecosystem has various form factors and device manufactures, it is practically impossible to buy each targeted device to test the app. You have to go for some cost effective solution to test your app on all targeted devices.

The following section will guide you through some of the tips to make your Android emulator faster and other alternative options available to test Android apps.

Using the x86 emulator
Android provides an x86 emulator that can speed up the development considerably due to faster start and execution times for the AVD. The x86 emulator is not a part of the base Xamarin installation, but the directions for the installation can be found on the Xamarin website as well as the Android developer website. A very specific version may be required, particularly if you are using OS X Mavericks, so we will not replicate the directions here.

Once installed, you can take advantage of the x86 emulator by the selected Intel Atom (x86) for the CPU/ABI when editing an AVD configuration.

Third-party emulator solutions
The x86 solution should work just fine on the computer with a decent memory. Apart from the native Android emulators, you can use some of the third-party tools such as the Xamarin Android Player or Genymotion.

The Xamarin Android Player
Xamarin recently announced its own Android emulator called the Xamarin Android Player. The Xamarin Player runs on Android x86 hardware accelerated virtualization and OpenGL 2.0 for fast booting and a smooth user interface. This works on both the Windows and Mac platforms. Your system requires to have an OpenGL 2.0 compatible graphics card with at least 2 GB hard disk space and at least 2 GB or RAM to install and run the Xamarin player.
The following screenshot shows the Xamarin Android Player with the player settings pane:

![Xamarin Android Player screenshot](image)

The Xamarin player settings pane allows you to simulate some of the emulator controls, such as battery life, geo locations, volume, and power control.

**Genymotion**

Genymotion is yet another alternative in the race of faster Android emulators. Genymotion is based on the open source project Android VM, developed by a French company Genymobile. It also works on all major platforms, including Mac, Windows, and Linux. The basic version of Genymotion is free to download; however, the commercial version does much more than the free version. The paid version of Genymotion can simulate multitouch, camera, GPS, network quality simulation, accelerometer, and much more. This can be the best alternative replacement for Android AVD.
Creating the Points Of Interest App

The following screenshot shows the Xamarin Android Player on Mac OS:

Creating the POI app
Now that we already have our development environment ready, let's begin by building the POIAp. 
The following section will walk you through the several steps involved to create, build, and deploy the POI application into the Android device:

1. Launch Xamarin Studio.
2. From the File menu, navigate to New | Solution. The New Solution view will be presented, as shown in the following screenshot:

3. Select the Android section on the left-hand side of the screen, navigate to App | Android App, and then click on Next.
4. Enter the app name as POIApp and the application identifier as com.packt.poiapp.
5. From the Compatibility selection, select Maximum Compatibility to make your application compatible on a full range of devices.
6. From the **Themes** drop-down menu, you may select from the range of available themes and click on **Next**:
Chapter 3

7. Review the project location and adjust it if desired. Click on **Create**:

![Configure your new project](image)

8. Xamarin Studio will create a solution and the required project folders. The project folder will contain a default `MainActivity.cs` class and a `Main.axml` layout file.
Creating the Points Of Interest App

The Xamarin Studio IDE

After creating POIApp, the project will be opened within the environment.

The following screenshot depicts Xamarin Studio after the project has been created:

Like any other modern IDE, Xamarin Studio is organized with a set of menus at the top of the screen, a context-sensitive toolbar below it, and a series of dockable pads for viewing and manipulating various types of content. By default, Xamarin Studio is configured with the following options:

- The Solution pad is docked on the left-hand side and allows you to explore the structure and content contained in the project
- Editor windows are in the middle and present the content of files for viewing and manipulation
- Task-specific pads are collapsed on the right-hand side and bottom and can be expanded by hovering over the icon and caption

Additional pads can be accessed by navigating to View | Pads.
Chapter 3

The Project Options view

There are numerous options that can be set, which would affect the way the application is built and executed. These options can be adjusted from within the Project Options view. The following section will show you how to use the different options available in the Xamarin Studio project explorer to set various project configurations.

Understanding the project structure

A project is an organizational unit that represents the complete Xamarin Android application in the Solution pad. It contains not just the source code, but also contains the dependent libraries, resources, and other project configurations. Before we begin with writing any code, we must understand the Xamarin.Android project structure and significance of each folder. The following screenshot depicts the different components a Xamarin Android application is composed of:
Xamarin Studio project wizard creates the default project structure and adds the required files and directory to Solution pad. The most important building blocks of a Xamarin.Android project includes:

- The main project (POIApp) is the root directory that contains the entire project context. Right-click on the project name to get various options, such as clean, build, run, options for project configurations, and so on.
- The References directory contains the references to the lists of the base class libraries and assemblies used in the application. Right-click on Edit References to add a base class library or third-party assembly.
- The Components folder contains reusable pieces of code that are built by community developers and shared on the Xamarin component store. A component gives you the ability to quickly add new controls and functionality to Xamarin applications. For example, if your application uses the database operation, you can quickly integrate the SQLite.Net component to perform SQL operations by writing a few lines of code. We will discuss how to add a component to your application from the Xamarin component store in Chapter 4, Adding a List View.
- The Assets folder contains the raw assets that can be bundled with the application. It can contain files such as third-party .ttf fonts, game textures, and so on.
- The Properties folder generally contains two files: the AndroidManifest.xml and AssemblyInfo.cs files. The AndroidManifest.xml file contains the metadata of the Android application and AssemblyInfo.cs contains information about the assembly, such as the name, description, version, and so on.
- The Resources folder is the main building block composed of images (named as drawables in Android), layout descriptors, strings, colors, themes, and so on. All the resources added are grouped into different folders and are referenced with a unique resource ID.

Xamarin Studio automatically creates a new Resource.designer.cs file under the Resources directory. This file holds the unique IDs for each resource. This is similar to the R.java file that is created automatically in the native Android application. This file is maintained by Xamarin.Android and is periodically regenerated whenever there is a change made to the application’s resources.
The following are some of the subdirectories you can create inside the Resources directory:

- **Resources/drawable-xxx**: The drawable folders are used to hold the images such as .png, .jpeg, and so on. Notice that the default project structure contains multiple drawable folder suffixes with qualifiers, such as hdpi, mdpi, xhdpi, and so on. The qualifier indicates that the resources inside that directory will be used in certain circumstances only. You can create another directory Resources/drawable to place the images that are not required for different device configurations. A more detailed discussion on resource qualifiers are explained in Chapter 7, Designing for Multiple Screen Sizes.

- **Resources/layout**: This folder contains the XML layout descriptor files. In our example, the Main.axml file is created.

- **Resources/values**: This folder contains files, such as string.xml, to declare all the strings used in an app. This is helpful for app localization.

- **Resources/menu**: The menu folder holds the XML-based menu specification for each activity.

- The application source code can be managed inside multiple different folders. By default, the Xamarin Studio project template is created in the MainActivity.cs file.

- The newer version of Xamarin Studio creates the Xamarin.UITest project, which is used for automated UI acceptance testing. The Xamarin.UITest framework is based on Calabash, using which you can write a test case in C# and NUnit and execute for both the Android and iOS platforms. This book covers Android unit testing using NUnitLite in Chapter 8, Creating Data Storage Mechanisms. For now, if you wish, you can safely delete the test project.

The Xamarin Studio layout designer

Open the Main.axml file located at Resources/layout/Main.axml from the solution explorer. Notice that the graphical layout editor will be the default layout editor. Currently, the Main.axml layout contains a button inside LinearLayout. Let's now proceed to understand the different Xamarin Studio layout designer options.

---

[ 59 ]
The designer content layout

The Android Studio layout designer contains two buttons: Content and Source, which are at the bottom of the designer view. These buttons allow you to switch between a visual representation of the layout (Content) and an XML source code view of the layout. This is very useful as some tasks are just quicker to edit directly in the XML source view; however, the content view is useful to view and arrange widgets.

The content view's usefulness is somewhat limited only because most of the time portions of a view must be constructed with the code at runtime; however, when the view can be specified completely in XML, the content view is very useful. In the content view, you will notice a set of a few useful tools arranged at the top of the window, as shown in the following screenshot:

In the upper-right corner, you will find a set of zoom controls. These allow you to zoom the layout in or out, depending on your monitor's size and the level of details you would like to view. At the top of the page, you will find drop-down menus that also let you select things, such as the screen size of the device to simulate, orientation of the device, and the version of the Android platform to simulate.
The Document Outline and Properties pads

On the bottom right-hand side of the IDE, you will notice the Document Outline and Properties pads.

The Document Outline pad provides a convenient means of navigating and selecting widgets, particularly as layouts get more complex. Select listView1 and then click on the Properties tab. Activate the Document Outline pad to see your layout outline, listing all the widgets, Views, or view groups contained in it.

Select any view on the designer window and click on the Properties tab to activate the Properties window. The Properties window allows you to edit/select the attributes for the selected view. Xamarin Studio takes care of generating the source code to reflect the changes made in the designer view.

Toolbox

The Toolbox pad is organized with the basic widgets at the top of the list and container widgets such as ListViews further down the list.

By default, the Toolbox pad is displayed on the top right-hand side of the Xamarin Studio IDE. There is a search box at the top that allows you to filter the widgets in the list, and there are two buttons on the right-hand side of the search box that allow you to adjust the way widgets are listed.

Setting the target framework

The target framework setting determines which API level will be available to you during development and testing. Let’s select the automatic setting; use the target framework version. In this case, it automatically selected API level 19 Android 4.4.

The minimum SDK version tells you which minimum API level is required for the application to run. Based on the value specified in the minimum SDK, the Google Play Store prevents the user from installing the application if the system’s API level is lower than the value specified.

In order to set the target framework, perform the following steps:

1. Select the POIApp project under the POIApp solution in the Solution pad.
2. Right-click on it and click on Options.
3. Navigate to Build | General on the left-hand side of the page:

4. In the Target framework field, select Use latest installed platform and click on OK.

**Setting the app icon and package name**

Xamarin.Android provides a default icon and package name for apps. The icon will be displayed on the Android device’s home drawer alongside other application lists, as well as on the action bar at the top of each view.

In order to adjust the defaults for these settings, perform the following steps:

1. The application icon images are provided in the Assets folder of the code bundle.

2. Using Finder/Windows Explorer, copy ic_launcher.png from your computer’s hard disk to the respective Resources\drawable folders.
3. From Xamarin Studio, navigate to Resources\drawable, right-click on it, and click on Add Files or alternatively, you can drag and drop the launcher images to the Xamarin Studio Resource/drawable folders.

4. You should now see ic_launcher.png listed under drawable in the Solution pad.

5. Select the POIApp project, right-click on it, and click on Options.

6. Navigate to Build \ Android Application.

7. Change the application name to POIApp. This will cause the resulting APK file to be named POIApp.apk.

8. Change the application icon selection to @drawable/ic_launcher.

9. Click on OK.

We will cover additional options in Chapter 11, Publishing an App to the App Store, as we prepare the app for deployment.

The launcher activity

When an app is launched from the home screen on an Android device, the Android OS creates an instance of the activity in the application you have declared to be the launcher activity. When developing with the Android SDK, this is specified in the AndroidManifest.xml file. The following code excerpt from AndroidManifest.xml shows how to specify an activity as a launcher activity:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.paket.POIApp">
<application
 android:allowBackup="true"
 android:icon="@drawable/ic_launcher"
 android:label="POIApp"
 android:theme="@style/AppTheme">
<activity
 android:name=".MainActivity"
 android:label="POIApp">
<intent-filter>
<action android:name="android.intent.action.MAIN"/>
<category android:name="android.intent.category.LAUNCHER"/>
</intent-filter>
</activity>
</application>
</manifest>
```
Xamarin.Android provides a more convenient method of specifying this by the use of .NET attributes. These .NET attributes are used at build time to construct the ApplicationManifest.xml file so that you rarely ever need to worry about working with the file directly. In the case of specifying the initial activity to launch, setting MainLauncher to true does the job, as seen in the following code example:

```csharp
[Activity (Label = "POIApp", MainLauncher = true)]
public class MainActivity : Activity
{
    ...
}
```

### Running and debugging the app

The way in which you run and debug apps is very important to developers as it has a big impact on productivity and timelines. Xamarin Studio and the Android emulator team up to make the testing and debugging cycle as painless as possible. Let's go through the following steps:

1. Start a debugging session by clicking on the **Play** button on the left-hand side of the taskbar, by pressing **F5**, or by navigating to **Run** | **Start Debugging**:

2. Select **Nexus 6** (emulator) from the list and click on the **Start emulator** button. It might take a little longer for the emulator to start up. Let's wait until it is fully loaded.

3. Select **Nexus 6** at the top of the devices list and click on **OK**. Xamarin Studio will deploy the compiled app to the emulator. The progress of the deployment can be monitored from the status view in the middle of the toolbar and from the **Application Output** pad at the bottom of the IDE.
4. Toggle to the Android emulator and unlock the screen. The POI app will be present:

![Android Emulator Screen](image)

The Android emulator is used for testing Android apps during the development process. The left-hand side of the screen depicts what would be seen on a device and the right-hand side provides keys that replicate the device hardware.

5. Click on the **Hello World** button and the app will increment a counter and update the button's caption.

6. Toggle back to Xamarin Studio and stop the app by clicking on the **Stop** button at the extreme left of the toolbar.

7. Open `MainActivity.cs` and set a breakpoint on line 21 by clicking on the left margin of the editor, just to the left of the line number.
8. Restart the app by clicking on the **Start** button. Since the Android emulator is still running, you will not need to make a device selection. The app will stop at the breakpoint previously set:

```
19   // Set our view from the "main" layout resource
20   setContentView (Resource.Layout.Main);
21
22
```

9. You will notice a set of debug controls present in the toolbar. There are controls to continue the execution, namely, step over the current line, step into the current function, and step out of the current function:

![Debug controls](image)

10. You will also notice a new set of pads related to the debugging apps present at the bottom of the IDE. These pads allow you to view objects, breakpoints, threads, and the call stack:

![Call stack](image)

11. Click on **Step Over** twice to watch the progress of the execution, and then click on **Continue** to let the app start.

As you can see from this section, Xamarin Studio and the Android emulator facilitate a robust and intuitive way of executing and debugging applications.

### Debugging with an Android device

Apps can be executed and debugged on actual devices with the same simplicity of working with an emulator. To prepare for using a physical device, you need to perform a few steps as follows:

1. Enable USB debugging on the device.
2. Install an appropriate USB driver for the device (Windows only).
Enabling USB debugging

In order to enable USB debugging on a device with Android 4.0 and newer, perform the following steps:

1. For devices running Android 4.2 or newer, there is an extra step; Developer options are initially hidden. Navigate to Settings | About phone and tap Build number seven times. On some configurations, the exact menu structure may differ. On my HTC One with Android 4.3, the menu is Settings | About | Software information | More.
2. Navigate to Settings | Developer options.
3. Click on USB debugging.

Installing a USB driver

Windows users are required to install a USB driver provided by the device's manufacturer. You can refer to the Android developer website under a section titled Using Hardware Devices for more details or consult your device's manufacturer at http://developer.android.com/tools/device.html.

OS X users should be good to go.

Running apps on a device

After completing the previous steps, simply connect the device to your development computer with a USB cable, start the app from Xamarin Studio, and choose the actual hardware device from the device selection view rather than starting an emulator.

Behind the scenes

It is interesting at this point to take a quick look at a few of the things that go on behind the scenes, which we previously discussed in Chapter 2, The Xamarin.Android Architecture.

The peer object

Let's start with the peer object (the proxy object) discussed in Chapter 2, The Xamarin. Android Architecture. Navigate to the POIApp\POIApp\obj\Debug\android\src\poiappp directory code bundle in your Finder/Windows Explorer window, open MainActivity.java using Notepad.
Creating the Points Of Interest App

The following code snippet depicts some of the key pieces of the source file:

```java
package poiapp;
public class MainActivity extends android.app.Activity implements mono.android.IGCUserPeer {
    ...;

    public void onCreate (android.os.Bundle p0) {
        n_onCreate (p0);
    }

    private native void n_onCreate (android.os.Bundle p0);

    ...;
}
```

Note the following points:

- The `MainActivity` class extends `android.app.Activity`, which is what you would expect
- An `onCreate()` proxy method is created that calls the native `n_onCreate()` method, which points to the overridden `OnCreate()` method in our managed C# class
- The `MainActivity` class has a static initialization block and a constructor that establishes the link between the Java class and its managed C# peer, including initializing `n_onCreate()`

The AndroidManifest.xml file

Navigate to POIApp\POIApp\obj\Debug\android in the code bundle and open the `AndroidManifest.xml` file. The following code snippet depicts a portion of the manifest file:

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1" android:versionName="1.0" package="com.packt.poiapp">
    <uses-sdk android:minSdkVersion="10" />
    <application android:label="POIApp" android:name="mono.android.app.Application" android:debuggable="true">
        <activity android:icon="@drawable/icon" android:label="POIApp" android:name="md56a0a1b7026a61848924491193f52dfa6.MainActivity">
            <intent-filter>
```

[68]
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
<provider android:name="mono.MonoRuntimeProvider"
android:exported="false" android:initOrder="2147483647"
android:authorities="com.packt.poiapp.mono.MonoRuntimeProvider.__mono_init__" />
<receiver android:name="mono.android.Seppuku">
<intent-filter>
<action android:name="mono.android.intent.action.SEPPUKU" />
<category android:name="mono.android.intent.category.SEPPUKU.com.packt.poiapp" />
</intent-filter>
</receiver>
</application>
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
</manifest>

Note the following points:

- The minimum SDK is set to 15 in the <uses-sdk> element
- The initial activity is set using the <category> element within the activity definition

We covered a lot about the structure of the Xamarin.Android project created using Xamarin Studio. We used Xamarin Studio on the Mac platform to complete all the examples in this book. We can confirm that all the previously mentioned project configuration options are also available in the Visual Studio IDE.

**Summary**

In this chapter, we started with a sample app that we will complete through the remaining chapters of the book, and we demonstrated the facilities we have for executing and debugging apps.

In the next chapter, we will start with creating the basic layout and building ListView to display the POIApp fetched from the server.
Where to buy this book
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.