Xamarin Mobile Application Development for Android
Second Edition

Develop, test, and deliver fully featured Android applications using Xamarin

Nilanchala Panigrahy
Xamarin Mobile Application Development for Android
Second Edition

Develop, test, and deliver fully featured Android applications using Xamarin

Nilanchala Panigrahy

PACKT open source publishing
community experience distilled
BIRMINGHAM - MUMBAI
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.

__Firstly, I'd like to thank Bibhu, whose love, patience, and understanding pushes me to a new high every time I take up new opportunities. Your support makes everything I do possible. A big thanks to my family for their love, encouragement, and belief.__

Thanks to the entire editorial team at Packt Publishing, especially Arun and Meeta, who had to contend with many delays while I juggled between full-time application development and my book writing commitments.

A special thanks to all the technical reviewers of this book, who did an excellent job to ensure that the content of this book is up to date and the source code is error-free.

I would like to thank you all for choosing this book. I hope you will find all the information you seek.
About the Reviewers

Mart Roosmaa is a senior software consultant. He has worked with a wide range of technologies over the years, ranging from backend to embedded to desktop development, and most recently, mobile app development. He prefers to create Android applications with Xamarin/C#, as it allows him to deliver better results faster.

Ed Snider is a senior software developer at InfernoRed Technology, a speaker, and a Xamarin MVP based in the DC/Northern VA area. He has a passion for mobile development. He regularly speaks at local user groups and community events and is the organizer of the DC and Northern VA Mobile .NET Developers group. He is primarily focused on building mobile solutions on the Windows, iOS, and Android platforms for small and large organizations and has been working with .NET for over 10 years. He blogs at http://www.edsnider.net and can be found on Twitter at http://www.twitter.com/edsnider.
Adrian Stevens has over 14 years of experience in developing mobile applications, specializing in C# and C++ cross-platform development. He founded and led a successful mobile consulting company.

Adrian is also an experienced author and instructor, having written several published articles on mobile and IoT development as well as authored the curriculum on cross-platform development.

Adrian's expertise includes wireless communication, architecture, data processing, and mathematics.

He is based out of Vancouver, Canada, and co-runs a C# cross-platform development Meetup community.

Michael Williams is a hybrid mobile architect. He is an insightful, result-driven IT professional with notable experience in creating his own business, developing mobile applications for iOS, Android, and Windows Phone. He also has experience in game development, directing a broad range of corporate IT initiatives, and participating in the planning, analysis, and implementation of solutions in support of business objectives. He has hands-on experience in all the stages of system development, including the requirements definition, design, architecture, testing, and support.

His previous employers include IBM, Sound Therapy International, and Bepoz. He currently works for DNS Technology, Radio DI, and Flush Arcade (his own company).
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>ix</td>
</tr>
<tr>
<td><strong>Chapter 1: The Anatomy of an Android App</strong></td>
<td>1</td>
</tr>
<tr>
<td>The Android platform</td>
<td>2</td>
</tr>
<tr>
<td>Versions of Android</td>
<td>2</td>
</tr>
<tr>
<td>The Linux kernel</td>
<td>9</td>
</tr>
<tr>
<td>Native libraries</td>
<td>9</td>
</tr>
<tr>
<td>The Android runtime</td>
<td>9</td>
</tr>
<tr>
<td>The application framework</td>
<td>10</td>
</tr>
<tr>
<td>The application layer</td>
<td>11</td>
</tr>
<tr>
<td><strong>The building blocks of the Android application</strong></td>
<td>11</td>
</tr>
<tr>
<td>The Android packages (.apk)</td>
<td>11</td>
</tr>
<tr>
<td>The application manifest</td>
<td>12</td>
</tr>
<tr>
<td>Activities</td>
<td>12</td>
</tr>
<tr>
<td>The life cycle of an activity</td>
<td>13</td>
</tr>
<tr>
<td>The events of an activity</td>
<td>14</td>
</tr>
<tr>
<td>Fragments</td>
<td>15</td>
</tr>
<tr>
<td>The fragment life cycle</td>
<td>16</td>
</tr>
<tr>
<td>Services</td>
<td>18</td>
</tr>
<tr>
<td>Content providers</td>
<td>18</td>
</tr>
<tr>
<td>Broadcast receivers</td>
<td>18</td>
</tr>
<tr>
<td>Views and ViewGroups</td>
<td>19</td>
</tr>
<tr>
<td>Declarative versus programmatic View creation</td>
<td>19</td>
</tr>
<tr>
<td>User interface widgets</td>
<td>19</td>
</tr>
<tr>
<td>Common layouts</td>
<td>20</td>
</tr>
<tr>
<td>Adapter layouts</td>
<td>22</td>
</tr>
<tr>
<td>XML layout files</td>
<td>22</td>
</tr>
<tr>
<td>Element and attribute names</td>
<td>23</td>
</tr>
<tr>
<td>The View and layout identifiers</td>
<td>23</td>
</tr>
<tr>
<td>Table of Contents</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td></td>
</tr>
<tr>
<td>Using XML layouts from activities</td>
<td>24</td>
</tr>
<tr>
<td>Intents</td>
<td>24</td>
</tr>
<tr>
<td>Resources</td>
<td>24</td>
</tr>
<tr>
<td>The R.java file</td>
<td>24</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>25</td>
</tr>
<tr>
<td><strong>Chapter 2: The Xamarin.Android Architecture</strong></td>
<td>27</td>
</tr>
<tr>
<td>Getting started with Xamarin</td>
<td>27</td>
</tr>
<tr>
<td><strong>Why Xamarin.Android?</strong></td>
<td>29</td>
</tr>
<tr>
<td>The benefits of using Xamarin.Android</td>
<td>29</td>
</tr>
<tr>
<td>The drawbacks of using Xamarin.Android</td>
<td>29</td>
</tr>
<tr>
<td><strong>What is Mono?</strong></td>
<td>30</td>
</tr>
<tr>
<td>Mono and Dalvik side by side</td>
<td>30</td>
</tr>
<tr>
<td>The Java Native Interface</td>
<td>31</td>
</tr>
<tr>
<td>Peer objects</td>
<td>31</td>
</tr>
<tr>
<td>Xamarin.Android application packaging</td>
<td>32</td>
</tr>
<tr>
<td><strong>The Android bindings design</strong></td>
<td>33</td>
</tr>
<tr>
<td>Design principles</td>
<td>33</td>
</tr>
<tr>
<td>C# properties</td>
<td>33</td>
</tr>
<tr>
<td>Delegates</td>
<td>34</td>
</tr>
<tr>
<td>Constants to enumerations</td>
<td>34</td>
</tr>
<tr>
<td><strong>Development environments</strong></td>
<td>35</td>
</tr>
<tr>
<td>Xamarin Studio</td>
<td>35</td>
</tr>
<tr>
<td>Xamarin for Visual Studio</td>
<td>36</td>
</tr>
<tr>
<td>IDE comparison</td>
<td>37</td>
</tr>
<tr>
<td>Compatibility</td>
<td>38</td>
</tr>
<tr>
<td><strong>Summary</strong></td>
<td>38</td>
</tr>
<tr>
<td><strong>Chapter 3: Creating the Points Of Interest App</strong></td>
<td>39</td>
</tr>
<tr>
<td>The sample POIApp</td>
<td>39</td>
</tr>
<tr>
<td>Installing Xamarin.Android</td>
<td>40</td>
</tr>
<tr>
<td>Installing platforms and tools</td>
<td>43</td>
</tr>
<tr>
<td>Tools</td>
<td>44</td>
</tr>
<tr>
<td>The Android platform APIs</td>
<td>45</td>
</tr>
<tr>
<td>The Android platform Extras</td>
<td>46</td>
</tr>
<tr>
<td><strong>Creating Android Virtual Devices</strong></td>
<td>47</td>
</tr>
<tr>
<td>Cloning a virtual device</td>
<td>49</td>
</tr>
<tr>
<td><strong>Speeding up the Android emulator</strong></td>
<td>50</td>
</tr>
<tr>
<td>Using the x86 emulator</td>
<td>50</td>
</tr>
<tr>
<td>Third-party emulator solutions</td>
<td>50</td>
</tr>
<tr>
<td>The Xamarin Android Player</td>
<td>50</td>
</tr>
<tr>
<td>Genymotion</td>
<td>51</td>
</tr>
<tr>
<td>Table of Contents</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td></td>
</tr>
<tr>
<td>Creating the POI app</td>
<td>52</td>
</tr>
<tr>
<td>The Xamarin Studio IDE</td>
<td>56</td>
</tr>
<tr>
<td>The Project Options view</td>
<td>57</td>
</tr>
<tr>
<td>Understanding the project structure</td>
<td>57</td>
</tr>
<tr>
<td>The Xamarin Studio layout designer</td>
<td>59</td>
</tr>
<tr>
<td>The designer content layout</td>
<td>60</td>
</tr>
<tr>
<td>The Document Outline and Properties pads</td>
<td>61</td>
</tr>
<tr>
<td>Toolbox</td>
<td>61</td>
</tr>
<tr>
<td>Setting the target framework</td>
<td>61</td>
</tr>
<tr>
<td>Setting the app icon and package name</td>
<td>62</td>
</tr>
<tr>
<td>The launcher activity</td>
<td>63</td>
</tr>
<tr>
<td>Running and debugging the app</td>
<td>64</td>
</tr>
<tr>
<td>Debugging with an Android device</td>
<td>66</td>
</tr>
<tr>
<td>Enabling USB debugging</td>
<td>67</td>
</tr>
<tr>
<td>Installing a USB driver</td>
<td>67</td>
</tr>
<tr>
<td>Running apps on a device</td>
<td>67</td>
</tr>
<tr>
<td>Behind the scenes</td>
<td>67</td>
</tr>
<tr>
<td>The peer object</td>
<td>67</td>
</tr>
<tr>
<td>The AndroidManifest.xml file</td>
<td>68</td>
</tr>
<tr>
<td>Summary</td>
<td>69</td>
</tr>
<tr>
<td>Chapter 4: Adding a List View</td>
<td>71</td>
</tr>
<tr>
<td>Creating the POI ListView layout</td>
<td>71</td>
</tr>
<tr>
<td>Creating POIListActivity</td>
<td>75</td>
</tr>
<tr>
<td>Creating the POI list row layout</td>
<td>76</td>
</tr>
<tr>
<td>Adding a RelativeLayout view group</td>
<td>78</td>
</tr>
<tr>
<td>Adding an ImageView widget</td>
<td>79</td>
</tr>
<tr>
<td>Adding a LinearLayout widget</td>
<td>80</td>
</tr>
<tr>
<td>Adding the name and address TextView classes</td>
<td>80</td>
</tr>
<tr>
<td>Adding the distance TextView</td>
<td>82</td>
</tr>
<tr>
<td>Creating the PointOfInterest apps entity class</td>
<td>84</td>
</tr>
<tr>
<td>Populating the ListView item</td>
<td>85</td>
</tr>
<tr>
<td>Creating POILListViewAdapter</td>
<td>86</td>
</tr>
<tr>
<td>Implementing a constructor</td>
<td>86</td>
</tr>
<tr>
<td>Implementing Count { get }</td>
<td>87</td>
</tr>
<tr>
<td>Implementing GetItemId()</td>
<td>87</td>
</tr>
<tr>
<td>Implementing the index getter method</td>
<td>87</td>
</tr>
<tr>
<td>Implementing GetView()</td>
<td>88</td>
</tr>
<tr>
<td>Reusing row Views</td>
<td>88</td>
</tr>
</tbody>
</table>
**Table of Contents**

**Populating row Views** 88  
Populating the list thumbnail image 89  
Using UrlImageViewHelper 89  
Adding Internet permissions 91  

**Hooking up POIListViewAdapter** 92  
Consuming the web service 94  
An introduction to web services 95  
Deploying the POI web service 95  
Consuming REST web services asynchronously 96  
Creating the POIService class 97  
Asynchronous programming with async and await 98  
Serializing and deserializing using Json.NET 100  
Updating POIListActivity 101  

**Adding actions to ActionBar** 102  
Defining the menu XML file 103  
Setting menus in OnCreateOptionsMenu() 104  
Handling selection in OnOptionsItemSelected() 104  
Handling the ListView click event 105  
Handling no network condition 106  
Toast 108  
Summary 108  

Chapter 5: Adding a Details View 109  
Creating the POIDetail layout 109  
Understanding ScrollView 110  
Using the TableLayout manager 113  
Working with EditText's InputType 114  

**Creating POIDetailActivity** 117  
Adding navigation to POIDetailActivity 117  
Navigation on the POIList item click 118  
Receiving the data in POIDetailActivity 120  
Binding variables to controls 121  
Populating user interface widgets 121  
Adding the save and delete actions 123  
Disabling the delete action 124  

**Adding the save and delete actions in POIService** 125  
Consuming the web service to create or update a POI 125  
Adding the CreateOrUpdatePOIAasync method to POIService 126  
Consuming the web service to delete the POI 128  
Adding the DeletePOIAasync method to POIService 129
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating SavePOI()</td>
<td>130</td>
</tr>
<tr>
<td>Creating DeletePOI()</td>
<td>132</td>
</tr>
<tr>
<td>Adding validation</td>
<td>133</td>
</tr>
<tr>
<td>Using the EditText.Error property</td>
<td>134</td>
</tr>
<tr>
<td>Adding a delete confirmation prompt</td>
<td>136</td>
</tr>
<tr>
<td>Refreshing POIListActivity</td>
<td>137</td>
</tr>
<tr>
<td>Summary</td>
<td>138</td>
</tr>
<tr>
<td>Chapter 6: Making Your App Orientation-aware</td>
<td>139</td>
</tr>
<tr>
<td>The Android behavior on configuration changes</td>
<td>139</td>
</tr>
<tr>
<td>Locking the Android application orientation</td>
<td>140</td>
</tr>
<tr>
<td>Dynamically requesting the orientation</td>
<td>141</td>
</tr>
<tr>
<td>Saving the activity state for configuration changes</td>
<td>142</td>
</tr>
<tr>
<td>Manually saving the activity state</td>
<td>143</td>
</tr>
<tr>
<td>Retaining the POI list scroll position</td>
<td>144</td>
</tr>
<tr>
<td>Building the orientation-aware layout</td>
<td>145</td>
</tr>
<tr>
<td>Manually handling the orientation behavior</td>
<td>148</td>
</tr>
<tr>
<td>Summary</td>
<td>149</td>
</tr>
<tr>
<td>Chapter 7: Designing for Multiple Screen Sizes</td>
<td>151</td>
</tr>
<tr>
<td>An introduction to Android tablets</td>
<td>152</td>
</tr>
<tr>
<td>Creating an Android tablet emulator</td>
<td>152</td>
</tr>
<tr>
<td>An introduction to fragments</td>
<td>155</td>
</tr>
<tr>
<td>Creating a new fragment</td>
<td>155</td>
</tr>
<tr>
<td>Making POIApp compatible with Android tablets</td>
<td>157</td>
</tr>
<tr>
<td>Using fragments for displaying POIDetails</td>
<td>159</td>
</tr>
<tr>
<td>Creating the POIDetailsFragment layout</td>
<td>159</td>
</tr>
<tr>
<td>Creating the POIDetailFragment</td>
<td>159</td>
</tr>
<tr>
<td>Working with the save and delete actions</td>
<td>161</td>
</tr>
<tr>
<td>Adding the POIDetailFragment to POIDetailActivity</td>
<td>163</td>
</tr>
<tr>
<td>Modifying the POI details activity layout</td>
<td>163</td>
</tr>
<tr>
<td>Refactoring POIDetailActivity to add POIDetailFragment</td>
<td>164</td>
</tr>
<tr>
<td>Using the list fragment for displaying the POI list</td>
<td>166</td>
</tr>
<tr>
<td>Understanding ListFragment</td>
<td>166</td>
</tr>
<tr>
<td>Creating the POIListFragment layout</td>
<td>166</td>
</tr>
<tr>
<td>Creating the fragment to display the POI list using POIListFragment</td>
<td>167</td>
</tr>
<tr>
<td>Adding POIListFragment to POIListActivity</td>
<td>171</td>
</tr>
<tr>
<td>Modifying the POI list activity layout</td>
<td>171</td>
</tr>
<tr>
<td>Adding POIListFragment to POIListActivity</td>
<td>172</td>
</tr>
<tr>
<td>Creating a multi-pane layout for tablets</td>
<td>173</td>
</tr>
<tr>
<td>Updating the POIApp to support the multi-pane split layout</td>
<td>177</td>
</tr>
<tr>
<td>Chapter 8: Creating Data Storage Mechanisms</td>
<td>189</td>
</tr>
<tr>
<td>Data storage in Android</td>
<td>190</td>
</tr>
<tr>
<td>Shared preferences</td>
<td>190</td>
</tr>
<tr>
<td>The SQLite database storage</td>
<td>191</td>
</tr>
<tr>
<td>Using the SQLite.net ORM component</td>
<td>192</td>
</tr>
<tr>
<td>Using attributes to mark POIApp to be persisted</td>
<td>193</td>
</tr>
<tr>
<td>Adding a database helper class</td>
<td>194</td>
</tr>
<tr>
<td>Creating or updating the POI record</td>
<td>196</td>
</tr>
<tr>
<td>Reading POI details from the database</td>
<td>196</td>
</tr>
<tr>
<td>Deleting the POI data from the database</td>
<td>197</td>
</tr>
<tr>
<td>Using the Xamarin.Android NUnitLite</td>
<td>197</td>
</tr>
<tr>
<td>Setting up for tests</td>
<td>199</td>
</tr>
<tr>
<td>Creating the test methods</td>
<td>200</td>
</tr>
<tr>
<td>The CreatePOI test</td>
<td>200</td>
</tr>
<tr>
<td>The DeletePOI test</td>
<td>201</td>
</tr>
<tr>
<td>Executing the tests</td>
<td>202</td>
</tr>
<tr>
<td>Implementing the caching logic to POIApp</td>
<td>204</td>
</tr>
<tr>
<td>Summary</td>
<td>206</td>
</tr>
<tr>
<td>Chapter 9: Making POIApp Location Aware</td>
<td>207</td>
</tr>
<tr>
<td>Working with location services</td>
<td>207</td>
</tr>
<tr>
<td>Setting the app permissions</td>
<td>209</td>
</tr>
<tr>
<td>Configuring the emulator</td>
<td>210</td>
</tr>
<tr>
<td>Obtaining an instance of LocationManager</td>
<td>211</td>
</tr>
<tr>
<td>Requesting location change notifications</td>
<td>211</td>
</tr>
<tr>
<td>Implementing ILocationListener</td>
<td>212</td>
</tr>
<tr>
<td>Adding location services to POIApp</td>
<td>212</td>
</tr>
<tr>
<td>Adding location services to the POI list</td>
<td>212</td>
</tr>
<tr>
<td>Adding location services to POI details</td>
<td>215</td>
</tr>
<tr>
<td>Updating the user interface</td>
<td>215</td>
</tr>
<tr>
<td>Adding the code</td>
<td>216</td>
</tr>
<tr>
<td>Mocking location data for testing</td>
<td>217</td>
</tr>
<tr>
<td>Getting an address for a location</td>
<td>218</td>
</tr>
<tr>
<td>Keeping the user informed</td>
<td>220</td>
</tr>
<tr>
<td>Adding map integration</td>
<td>222</td>
</tr>
<tr>
<td>Navigating to the Map app</td>
<td>222</td>
</tr>
<tr>
<td>Checking for registered map apps</td>
<td>224</td>
</tr>
<tr>
<td>Summary</td>
<td>225</td>
</tr>
</tbody>
</table>
### Chapter 10: Adding the Camera App Integration

- Picking an integration approach 227
- Permissions and features 228
- Configuring the emulator 229
- Extending the data service 230
  - Implementing GetFileName() 230
  - Implementing GetImage() 230
  - Implementing DeleteImage() 231
- Capturing an image from POIDetailFragment 231
  - Adding UI elements 232
  - Creating the camera Intent 234
  - Checking for registered camera apps 234
  - Providing additional information with the Intent
    - Providing a filename and location 235
    - Providing a size limit 235
  - Starting the Intent 235
  - Completing the NewPhotoClicked() method 236
  - Processing the results of the Intent 237
- Upload an image using an HTTP multipart 238
- Summary 242

### Chapter 11: Publishing an App to the App Store

- Preparing the application for release 243
  - Disabling the debug mode 244
  - Configuring linking options 244
    - Side effects of linking 246
  - Selecting supported ABIs 247
  - Verifying the package name, icon, and app version
    - The package name 248
    - The application name and icon 249
    - The application versioning scheme 250
  - Review user permissions 251
- Publishing a signed APK for uploading 252
  - Android keystores 252
  - Publishing from Xamarin.Android 253
  - Republishing the app 256
- Different app distribution options 257
  - Self-publishing via website links or e-mails 257
  - Publishing on Google Play 258
  - Third-party app stores 260
- Summary 261

Index 263
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.
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 stared 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 List View, 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.

**What you need for this book**

All the examples in this book can be completed using a 30-day trial version of Xamarin.Android. The examples were developed using Mac OS X (Yosemite), Xamarin Studio 5.9.3, and Xamarin.Android 5.1.3 (trial edition). Any later versions should work fine as long as they are valid Xamarin configurations. You can check the Xamarin website for specifics.

Xamarin.Android can also be used in other configurations, including Windows OS. In Windows OS, you can either use Xamarin Studio or the Visual Studio Xamarin plugin as an IDE of your choice. Using a different configuration from what was used while developing the examples of this book may result in slight variations in the screens or steps described in the book.

The examples provided in this book consume the REST web service developed in Java JAX-RS. You can deploy the web service code on your system to perform end-to-end testing, or alternatively, you can use the Apiary mock feed URLs provided in the code bundle. To deploy the web service code, you need MySQL and the Apache Tomcat™ application server.

**Who this book is for**

This book for C# and .NET developers with a desire to develop Android apps using their existing skill sets. This book includes a step-by-step approach to build an Android application using the Xamarin platform which will be valuable whether you're an experienced mobile developer or making your first push.
It is assumed that you have some experience in software development and are familiar with the basic object-oriented development concept and practices. An understanding of C# syntax is a requirement and a good working knowledge of C# is a distinct advantage, though it is strictly not necessary.

**Conventions**

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:

"These constants are placed in a Java source file named `R.java`.

A block of code is set as follows:

```csharp
public override bool OnCreateOptionsMenu(IMenu menu)
{
    MenuInflater.Inflate(Resource.Menu.POIListViewMenu, menu);
    return base.OnCreateOptionsMenu(menu);
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```csharp
public override Dialog OnCreateDialog (Bundle savedInstanceState)
{
    POIDetailFragment targetFragment = (POIDetailFragment)TargetFragment;
    string poiName = Arguments.GetString("name");
}
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this:

"Click on **Step Over** twice to watch the progress of the execution."
Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback
Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support
Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code
You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

Downloading the color images of this book
We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/B04210_Graphics.pdf.
Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.
The Anatomy of an Android App

We live in an age where technologies are being developed and have become more and more accessible than ever before. The rise of the mobile computing platform has taken the technology evolution to a new high. The phones and tablets are getting smarter day by day and becoming the alternatives to a traditional PC. In this fiercely competitive world of mobile computing, every traditional PC manufacturers to small start-ups are in the race bringing the devices of various form factors.

In this book, we will show you how to take advantage of your existing C# skills to write applications that run on Android powered devices. While most of this book will be focused on learning how to develop Android apps using C# and Xamarin. Android, we will start with a more general discussion of Android. What is Android? How does Android facilitate the task of creating great mobile apps? This chapter will help you answer these questions by providing a base-level understanding of the following topics:

• An overview of the Android platform
• Android platform versions and feature releases
• Android applications (building blocks)
The Android platform

The Android platform has been one of the most powerful, evolving, and advanced mobile operating systems developed in recent years, which provides various services and features, that helps developers to build rich mobile applications. Android is an open source operating system currently developed and maintained by Google. Due to its open source nature, it has a larger community base of developers and device manufacturers.

The Android operating system was primarily designed for low powered computing phones, but later, its base was widen to various form factors, including smartphones, tablets, Android TV, and Wearables.

Versions of Android

The Android operating system has evolved with a series of frequent updates since its initial beta release in November 2007. Identifying the version of the Android platform can be somewhat confusing; there is a version number, API level, and nickname, and these are sometimes used interchangeably.

The version number represents a release of the platform. Sometimes, a new release is created to deliver new capabilities, while sometimes it is created to fix bugs.

The API level is an integer value that represents a set of capabilities. As the API level increases, new capabilities are delivered to the developer.

The following table lists all the major Android platform releases in the reverse chronological order:

<table>
<thead>
<tr>
<th>Platform version</th>
<th>API level</th>
<th>Release date</th>
<th>Feature updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 (Lollipop)</td>
<td>22</td>
<td>03/09/2015</td>
<td>• Stability and performance improvements over Lollipop. • Adds multiple SIM card support.</td>
</tr>
<tr>
<td>Platform version</td>
<td>API level</td>
<td>Release date</td>
<td>Feature updates</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------</td>
<td>--------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>5.0 (Lollipop)</td>
<td>21</td>
<td>11/12/2014</td>
<td>• Introduced new runtime ART that replaced Dalvik.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• A complete UI overhaul with an introduction to material design.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Improved lock screen notification.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Updated the media API for better camera capture and media playback.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Added the project Volta to increase the battery life.</td>
</tr>
<tr>
<td>4.4W, 4.4W.1, 4.4W.2 (KitKat Wearables)</td>
<td>20</td>
<td>06/25/2014</td>
<td>• Initial release for the Android Wear platform's smart watches.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Fork of same Android 4.4 KitKat code branch with added wearable extensions.</td>
</tr>
<tr>
<td>4.4.x (KitKat)</td>
<td>19</td>
<td>10/31/2013</td>
<td>• The default interface is made white instead of blue.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Added wireless printing capability.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Support for translucent navigation and status bars.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Ability for apps to use immersive mode to allow navigation and status bars hidden while maintaining user interaction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Action overflow menu buttons are always visible, even for the devices with the hardware Menu key.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• New framework (property animation) for UI transitions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Built-in screen recording feature.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Introduced Android Runtime (ART) as a new experimental runtime environment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Other security enhancements and bug fixes.</td>
</tr>
<tr>
<td>Platform version</td>
<td>API level</td>
<td>Release date</td>
<td>Feature updates</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------</td>
<td>--------------</td>
<td>-----------------</td>
</tr>
</tbody>
</table>
| 4.3.x (Jelly Bean) | 18 | 07/24/2013 | • Supports restricted access mode for new user profiles.  
• Introduced a platform support for Khronos OpenGL ES 3.0, providing better performance for 2D and 3D graphics rendering.  
• Bluetooth low energy support.  
• Optimized location and sensor capabilities, including hardware geo fencing optimizations.  
• Many security enhancements, performance enhancements, and bug fixes. |
| 4.2, 4.2.2 (Jelly Bean) | 17 | 11/13/2012 | • Improved the hardware-accelerated 2D renderer to make the animation smoother.  
• Introduced the interactive screensaver mode called Daydream.  
• Adds a presentation window and external display support.  
• Full native support for right-to-left (RTL) layouts.  
• Numerous bug fixes. |
| 4.1, 4.1.x (Jelly Bean) | 16 | 07/09/2012 | • Faster and smoother interface.  
• Bi-directional text and other language support.  
• Introduced expandable notification.  
• Added the new activity launcher animation via ActivityOptions.  
• Improved WebView to provide better HTML5 video viewing and canvas animation. |
| 4.0.3, 4.0.4 (Ice Cream Sandwich) | 15 | 12/16/2011 | • Bug fixes and stability improvements.  
• New APIs, including a social stream API in the Contacts provider.  
• Better camera performance.  
• Smoother screen rotation. |
<table>
<thead>
<tr>
<th>Platform version</th>
<th>API level</th>
<th>Release date</th>
<th>Feature updates</th>
</tr>
</thead>
</table>
| 4.0, 4.0.1, 4.0.2 (Ice Cream Sandwich) | 14 | 10/19/2011 | • Major interface overhaul with the new Roboto font family.  
• Unified the UI framework to work for phones, tablets, and more.  
• Lock screen improvements with the ability to access apps from lock screen.  
• Introduced the new voice input engine.  
• Improved media streaming capabilities.  
• Ability to shut down applications from the recent apps list with a swipe.  
• Hardware acceleration of the UI. |
| 3.2.x (Honeycomb) | 13 | 07/15/2011 | • Optimizations for a wider range of tablets.  
• Added a system-wide sync facility that made the SD card files accessible to apps from the system media store.  
• Bug fixes and other minor improvements. |
| 3.1 (Honeycomb) | 12 | 05/10/2011 | • Connectivity API for USB accessories.  
• Various UI framework updates.  
• Resizable home screen app widgets.  
• Support for an HTTP proxy for each connected Wi-Fi access point.  
• High-performance Wi-Fi lock, maintaining high-performance Wi-Fi connections when the device's screen is off.  
• Updated the animation framework class with ViewPropertyAnimator and animating background color. |
<table>
<thead>
<tr>
<th>Platform version</th>
<th>API level</th>
<th>Release date</th>
<th>Feature updates</th>
</tr>
</thead>
</table>
| 3.0 (Honeycomb)  | 11        | 02/22/2011   | • Added a new user interface, which optimized tablets.  
                                 • Introduced an action bar, giving access to contextual quick actions at the top of the screen.  
                                 • Added the fragment, a self-contained container that can be embedded in an activity. It has its own life cycle callback and is used to design tablets.  
                                 • Added the system-wide clipboard.  
                                 • Improved the status bar notification to support more content-rich notifications.  
                                 • Added new animation frameworks. |
| 2.3.3, 2.3.7     | 10        | 02/02/2011   | • Improvements and bug fixes. |
| (Gingerbread)    |           |              | |
| 2.3, 2.3.1, 2.3.2 (Gingerbread) | 9 | 12/06/2010 | • Updated the user interface design for simplicity and speed.  
                                 • Added support for **Near Field Communication (NFC)**.  
                                 • Support for extra-large screen sizes and resolutions.  
                                 • Native support for more sensors, including gyroscopes and barometers.  
                                 • Introduced the concurrent garbage collection for improved app responsiveness and smoother animation. |
| 2.2.x (Froyo)    | 8         | 05/20/2010   | • Improved speed, memory, and performance optimizations.  
                                 • Improved application speed using the JIT compilation.  
                                 • Support for the **Android Cloud to Device Messaging (C2DM)** service.  
                                 • Support for installing applications to the SD card memory.  
                                 • USB tethering and Wi-Fi hotspot functionality.  
                                 • Bug fixes and security patch updates. |
<table>
<thead>
<tr>
<th>Platform version</th>
<th>API level</th>
<th>Release date</th>
<th>Feature updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 (Eclair)</td>
<td>7</td>
<td>01/12/2010</td>
<td>• Minor API changes and bug fixes.</td>
</tr>
<tr>
<td>2.0.1 (Eclair)</td>
<td>6</td>
<td>12/03/2009</td>
<td>• Minor API changes and bug fixes.</td>
</tr>
</tbody>
</table>
| 2.0 (Donut)      | 5         | 10/26/2009   | • Updated camera features, including flash, digital zoom, white balance, color effect, and scene mode.  
|                  |           |              | • Optimized hardware speed and UI overhaul.  
|                  |           |              | • The MotionEvent class enhanced to track multi-touch events.  
|                  |           |              | • Expanded account sync, allowing users to add multiple accounts to a device. |
| 1.6 (Donut)      | 4         | 09/15/2009   | • Multilingual speech synthesis engine added to convert a text to speech.  
|                  |           |              | • Updated support for CDMA/EVDO, 802.1x, VPNs technologies. |
| 1.5 (Cupcake)    | 3         | 04/27/2009   | • Added third-party keyboard support with text prediction and user dictionary.  
|                  |           |              | • Video recording and playback in MPEG-4 and 3GP formats.  
|                  |           |              | • Added copy and paste features in the web browser.  
|                  |           |              | • Animated screen transitions.  
|                  |           |              | • Home screen widget support. |
| 1.1              | 2         | 02/09/2009   | • First Android platform update.  
|                  |           |              | • Updated the Map application.  
|                  |           |              | • In-call screen timeout default is now longer when using the speakerphone.  
|                  |           |              | • Adds support for saving attachments from MMS.  
|                  |           |              | • Adds support for marquee in layouts.  
|                  |           |              | • Various bug fixes. |
The Anatomy of an Android App

<table>
<thead>
<tr>
<th>Platform version</th>
<th>API level</th>
<th>Release date</th>
<th>Feature updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1</td>
<td>09/23/2008</td>
<td>• First commercial version of the Android platform.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Included apps such as Android Market, Gmail, Camera, Calendar, Contacts, Google Talk, Map, Media Player, Picture, Settings, and Browser.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Wi-Fi and Bluetooth support.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Instant messaging, text messaging, and MMS support.</td>
</tr>
</tbody>
</table>

The Android platform is comprised of applications, operating systems, runtime, middleware, services, and libraries. The following diagram provides a high-level view of how each layer in the Android platform is organized, and the subsequent sections provide a brief description of each major component:
The Linux kernel

Android is a Linux-based operating system designed and customized primarily for mobile devices, such as smartphones and tablets. Positioned at the bottom of the Android stack, the Linux kernel provides the interface between device hardware and Android software layers. The latest versions of Android are based on the Linux kernel version 3.4 or above (version 2.6 for versions prior to Android 4.0).

The Linux kernel provides some of the core system services such as memory management, process and task management, power management, networking stack, and various device drivers to interact with the device hardware.

Native libraries

Android is delivered with a set of native libraries written in C/C++, which provide various types of services. These libraries predominantly come from the open source community.

The Android runtime

The Android apps run within the Dalvik Virtual Machine (Dalvik VM), which is similar to a Java VM but has been optimized for devices with limited memory and processing capacity.

The Android apps are initially compiled to the Java byte code using the Java compiler, but they have an additional compilation step that transforms the Java byte code to the Dalvik byte code using a process called the Just in Time (JIT) compilation. The output produced by the JIT compiler is suitable to run within the Dalvik VM:

```
Java Source ➔ Java Compiler ➔ *.class files ➔ Java byte code ➔ Dalvik compiler (dex) ➔ *.dex files ➔ Dalvik byte code
```
Dalvik is delivered with the Android core libraries. These libraries do not align with a specific Java platform (JSE, JEE, or JME) but rather act as a hybrid platform most closely aligned with JSE, minus the user interface-focused components, AWT and Swing. The **Android Application Framework (AAF)** provides an alternate means of creating user interfaces.

Although Dalvik worked pretty well, the downside is that there is a huge lag every time the application is launched. That's where the new virtual machine, ART, comes in.

ART is precursor to Dalvik. It is the new application runtime introduced in Android 4.4 (KitKat) as a new experimental runtime environment and is implemented fully in Android 5.0 (Lollipop). This is primarily designed for performance and an improved app start up time. The primary difference between ART and Dalvik is the compilation approach. While Dalvik uses JIT, ART employs a new concept called **Ahead-of-Time (AOT)**. What this means is that new apps are getting compiled during installation, before they are even launched. To learn more about ART, you can refer to https://source.android.com/devices/tech/dalvik/.

---

### The application framework

The application framework is the part of the Android platform, which is most familiar to developers. It is delivered as a set of Java libraries and allows you to build user interfaces, interact with device capabilities such as the camera or location services, load and work with various types of application resources, and perform many more useful tasks. Here are some of the major services:

- **ActivityManager**: This service is responsible for the activity life cycle, state management, and controls the activity stack. Later, in this chapter, we will learn more about the activity life cycle.

- **WindowManager**: This service is responsible for managing the z-order list of screens. Each activity is attached to a window that is used to display the content on the screen, which is controlled by WindowManager.

- **Content providers**: This provides an interface to publish and share data between applications.

- **View system**: This provides a set of UI controls to build an application user interface.
• **NotificationManage**: This service manages application alerts and notifications.

• **ResourceManager**: This service provides access to resources, such as user interface layout, strings, color, dimensions, and so on.

• **PackageManager**: This holds the metadata of all the installed applications on the device.

• **TelephonyManager**: This provides information on the telephone services available on the device, such as status and subscriber information, to the application.

• **LocationManager**: This provides access to system location services.

The application layer
At the top of the stack sits the humble application, the component that actually delivers value to the user. Android comes with a set of applications that provide base functionality such as managing contacts, using the phone, checking e-mails, and browsing the web. The key to Android’s success is the vast array of third-party applications that can be installed, which allow users to do things, such as stream live sports events, edit a movie captured on the phone, interact with friends through their favorite social media site, and much more.

The building blocks of the Android application
Now, let's spend some time discussing applications—those things we write that provide value to the user. The Android applications are made up of various types of classes and resources. The following sections describe the different building blocks that an application can be composed of.

The Android packages (.apk)
Applications are delivered for installation in an Android package format. An Android package is created as the result of compiling an Android app and is an archive file with an `.apk` extension.
The Anatomy of an Android App

An Android package contains all of the code and the supporting files required to run a single application, including the following:

- Dalvik executables (.dex files)
- Resources
- Native libraries
- The application manifest

The Android packages can be installed directly via e-mails, URLs, or memory cards. They can also be installed indirectly through app stores such as Google Play.

**The application manifest**

All the Android applications have a manifest file (AndroidManifest.xml) that tells the Android platform everything it needs to know to successfully run the application, including the following:

- Minimum API level required by the application
- Hardware/software features used or required by the application
- Permissions required by the application such as location or camera
- The initial screen (Android activity) to start with when the application is launched
- The ability to install the application in the external memory
- Libraries, other than AAF, required by the application and so on

**Activities**

One of the most fundamental parts of an Android application is an activity. An activity represents a single screen with a user interface through which a user can interact with the application. A single application is composed of many activities. For example, a phone book application can have multiple activities representing different functions, such as list contacts, add contacts, capture contact photos, and so on.

A user interacts with an activity through one or more Views, which are described later in this chapter. If you are familiar with the **Model-View-Controller (MVC)** pattern, you would have noticed that the activities fulfill the role of the controller.
The life cycle of an activity

Activities have a well-defined life cycle that can be described in terms of states, transitions, and events. The following diagram provides a graphical view of the life cycle of an activity:

The states depicted in the preceding diagram are derived, which means that there is no State variable on an activity that explicitly identifies one of these states, but the state is implied and useful for discussion. The following table describes the behavior of an activity based on its state:

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running</td>
<td>The activity has been created and initialized and is visible and available to the user for interaction.</td>
</tr>
<tr>
<td>Paused</td>
<td>The activity view is being partially blocked by another activity.</td>
</tr>
<tr>
<td>Stopped</td>
<td>The activity is no longer visible to the user. The activity has not been destroyed, and the state is retained but it is placed in the background and no processing is allowed.</td>
</tr>
</tbody>
</table>
The events of an activity

During the transition between states, a series of events are called on the activity. These events provide developers a platform for various types of processing. The following table describes the different event callbacks and typically, the processing done in the application during each callback:

<table>
<thead>
<tr>
<th>Event</th>
<th>Called</th>
<th>Typical processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>onCreate</td>
<td>When an activity is created, generally from a user choosing to start the app</td>
<td>• This creates Views&lt;br&gt;• This initializes variables&lt;br&gt;• This allocates long-lived resources</td>
</tr>
<tr>
<td>onStart</td>
<td>After onCreate and right before an activity becomes visible to the user</td>
<td>• This allocates resources</td>
</tr>
<tr>
<td>onResume</td>
<td>Before an activity is ready to start interacting with a user and immediately after the onStart callback</td>
<td>• This initializes UI widgets for viewing&lt;br&gt;• This starts animations or videos&lt;br&gt;• This starts listening for GPS updates</td>
</tr>
<tr>
<td>onPause</td>
<td>When an activity's view has become partially blocked and is not the focus of input</td>
<td>• This commits unsaved updates&lt;br&gt;• This pauses animations or videos&lt;br&gt;• This stops listening for GPS updates</td>
</tr>
<tr>
<td>onStop</td>
<td>When an activity's view is no longer visible to the user</td>
<td>• This releases resources</td>
</tr>
<tr>
<td>onRestart</td>
<td>An activity is being placed back in the foreground, generally, because the user has selected the back button</td>
<td>• This allocates resources</td>
</tr>
<tr>
<td>onDestroy</td>
<td>Before the activity is destroyed</td>
<td>• This cleans up resources that may have been allocated by an activity</td>
</tr>
</tbody>
</table>

Something that is not obvious to developers and new to Android is the way the framework deals with device orientation changes. By default, when the orientation of a device is changed from portrait to landscape, Android destroys and recreates existing activities to help ensure that the most appropriate layout is used for the current device orientation.
If needed, this behavior can be overridden and activities can be retained. We will discuss special considerations in dealing with state and other processing concerns related to this topic in Chapter 6, *Making Your App Orientation-aware*.

**Fragments**

A fragment is a reusable user interface component, introduced since Android 3.0 (API level 11), and is primarily intended to build dynamic and modular user interfaces for different screen sizes. A fragment is always embedded in an activity, and like any other view, it lives in a **ViewGroup** (**ViewGroups** are explained in more detail later in this chapter) inside the view hierarchy. Like an activity, a fragment defines its own layout and has its own life cycle callbacks. When designing your application to support multiple form factors, fragments can be reused to optimize the user experience based on the available screen space.

Let's examine how fragments can be used to develop a modular user interface with the following example.

The following figure depicts the wireframe of a newsreader application, designed to work on both smartphone and tablet devices. As a tablet has more screen space, the news list and the details are presented as split views in a single activity, whereas the phone uses two different activities for this:
The Android smartphone uses two activities: **ActivityA** containing **FragmentA** is used to show the news list and **ActivityB** containing **FragmentB** is used to show the details of the selected news. In a tablet, we have a single activity **ActivityA** that contains both **FragmentA** and **FragmentB**.

As you can see here, **FragmentA** and **FragmentB** are the same implementation and are reused in different layout configurations to provide a different user experience on both the phone and tablet.

**The fragment life cycle**

Unlike the activity life cycle, understanding the fragment life cycle can be a bit tricky. In the following section, we will dig more into the fragment behavior and its life cycle methods.

The Android fragment has its own life cycle method, which is very similar to an activity. It contains all of the activity life cycle methods and is supplied with some additional callback methods. Fragments are always embedded in an activity, so its callbacks are directly affected due to the host activities' life cycle. For example, if the host activity receives **onStop()**, all the attached fragments also receive the **onStop()** callback.

The following diagram provides a graphical view of the fragment life cycle:
Let's take a look at each of the fragment life cycle events that gets called:

- **onInflate**: This event is called only if we define a fragment directly in an activity layout using the `fragment` tag, and while the content view of the activity is being inflated (typically, when `setContentView()` is called on an activity). This method passes `AttributeSet` that holds all the fragment attributes passed from the `fragment` tag. These attributes can be stored for later use. At this stage, the fragment is not even associated with an activity, and hence, we cannot perform any user interface-related tasks.

- **onAttach**: This is called once the fragment instance is associated with an activity.

- **onCreate**: This event is called after `onAttach` and before `onCreateView`; when the fragment instance is created or recreated. At this point, the base activity that holds this fragment is in the process of being created. At this point, you may use a background thread to get data for the fragment to use.

- **onCreateView**: At this point, the fragment instantiates its user interface and loads the view object hierarchy it contains. This method passes three arguments: `LayoutInflater`, `ViewGroup`, and `bundle`. The `LayoutInflater` argument can be used to inflate any layout for a fragment. A `bundle` specifies whether the fragment is created freshly or recreated. If it is recreated from the previous saved state, then the bundle will be non-null.

- **onActivityCreated**: This method is called when the activity that contains the fragment has been created, and the fragment's view hierarchy is instantiated. At this point, you can access the view by its ID using the `findViewById()` method and make any changes before it is visible to the user.

- **onStart**: This method is tied to the activity `onStart()` callback and is called when the fragment becomes visible to the user. At this point, the fragment is visible but not available for user interaction just yet.

- **onResume**: This method is called before the fragment is ready to start interacting with a user. At this point, the fragment is said to be running and the user is free to perform any operations on the app.

- **onPause**: This method is tied to the activity `onPause()` callback and is called when the fragment is taken out of the foreground.

- **onStop**: This method is tied to the activity `onStop()` callback and called when the fragment is not visible.
• onDestroyView: This method tells the fragment that the view created from 
  onCreateView() is now detached from the fragment. This callback is called 
  after onStop() and before the onDestroy() method.
• onDestroy: This method is called when the fragment is no longer in use. 
  This is called after onStop() and before onDetach().
• on_detach: This method is called after onDestroy(), and when the fragment 
  is no longer attached to an activity.

Services
Services are application components that run in the background to perform 
long-running operations with no direct access to the user interface. A typical 
long-running task can be periodic downloading of data from the Internet, persisting 
multiple records in a database, performing file I/O, fetching a phone contacts list, 
and so on. Such long-running tasks can be implemented using services to provide 
a smooth user experience by letting the user interact with other activities, while 
long-running jobs are being processed in the background.

Content providers
Content providers manage access to a central repository of data such as contacts. 
It provides you with a standard interface through which the other applications can 
access and manage the data repository.

Broadcast receivers
Broadcast receivers are components that perform some type of processing in 
response to system-wide broadcasts. Broadcasts are generally initiated by the system 
for events such as low battery, taking a picture, or turning on Bluetooth. Applications 
may also choose to send broadcasts; a content provider might send a broadcast when 
data, such as a contact, has been updated. While broadcast receivers do not have a 
user interface, they may indirectly cause updates to a status.
Views and ViewGroups
Everything that you see in an Android app is a View; buttons, labels, text boxes, and radio buttons are all examples of Views. Views are organized in a hierarchy using various types of ViewGroups. A ViewGroup is a special kind of View that is used to arrange (layout) other Views on the screen.

Declarative versus programmatic View creation
Views and ViewGroups can be created using two different methods: programmatically or declaratively. When using a programmatic approach, a developer makes API calls to create and position each individual View on the screen. When using a declarative approach, a developer creates XML layout files that specify how Views should be arranged. The declarative method enjoys several advantages stated as follows:

- It provides better separation of the visual design of an application from the processing logic
- It allows multiple layouts to be created to support multiple devices or device configurations with a single code base
- Development tools, such as Android Studio and the Android plugins for Eclipse and Xamarin Studio Android designer, allow you to view the user interface as you build it, without the need to compile and execute your application after each change

While most developers prefer the declarative method of View creation; in practice, some combination of programmatic and declarative methods is often required.

User interface widgets
Android provides a comprehensive set of user interface widgets that can be used to build a rich user experience. All of these widgets are subtypes of a View and can be organized into sophisticated layouts using various types of ViewGroups. All of the user interface widgets can be found in the android.widget package within the application framework.
The following screenshot depicts some of the basic Android widgets:

**Common layouts**

The application framework has a number of subclasses of `ViewGroup`, each of which provides a unique and useful way of organizing content:
The preceding diagram depicts some of the common layout managers available in Android. Layout managers are the ViewGroup classes that act as containers to host child views or layouts. Each of these standard layout managers provides a specific strategy to manage the size and position of its children. For example, the LinearLayout class places its children either horizontally or vertically, one view adjacent to the other.

The following table lists the different types of layout managers available in Android:

<table>
<thead>
<tr>
<th>Layout</th>
<th>Description</th>
<th>Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear layout</td>
<td>This organizes its children into a single horizontal or vertical row and creates a scrollbar when required.</td>
<td>Use this when widget positions flow horizontally or vertically.</td>
</tr>
<tr>
<td>Relative layout</td>
<td>This organizes child objects relative to each other or to the parent.</td>
<td>Use this when widget positions can best be described in relationship to another widget (to the left of) or the boundary area of the parent (right side, centered).</td>
</tr>
<tr>
<td>Table layout</td>
<td>This organizes its children into rows and columns.</td>
<td>Use this when widget positions would naturally fit into rows and columns. This is great when multiple columns of entry and labels are needed.</td>
</tr>
</tbody>
</table>

For complex layout scenarios, Android allows layouts to be nested. Deeply nested layouts can have an impact on the performance and should be avoided if possible.

**Downloading the example code**

You can download the example code files from your account at [http://www.packtpub.com](http://www.packtpub.com) for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit [http://www.packtpub.com/support](http://www.packtpub.com/support) and register to have the files e-mailed directly to you.
The Anatomy of an Android App

Adapter layouts
For layouts that are driven by a dynamic data source, the application framework has a set of classes derived from `AdapterView`:

The preceding diagram depicts two of the most common adapter layouts:

- **ListView**: This organizes content from the data source into a scrolling single column list
- **GridView**: This organizes content from the data source into a grid of columns and rows

XML layout files
To create a UI using a declarative method, Android provides an XML vocabulary with tags that define the various types of elements that can compose a View. The concept behind Android XML layout files is very similar to the way HTML tags are used to define web pages or Microsoft’s XAML tags are used to define **Windows Presentation Foundation (WPF)** user interfaces. The following example shows a simple View using a linear layout and containing a search entry field and search button:

```xml
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Other content here -->
    <EditText android:id="@+id/search_entry" android:hint="Search" />
    <Button android:id="@+id/search_button" android:text="Search" />
</LinearLayout>
```
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent">

<TextView
    android:text="Enter Search Criteria"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/searchCriteriaTextView" />

<Button
    android:text="Search"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/searchButton" />
</LinearLayout>

**Element and attribute names**

Care has been taken to align the names for elements and attributes in the XML vocabulary with class and method names from the application framework. In the previous example, the element names `LinearLayout`, `TextView`, and `Button` correspond to class names in the application framework. Likewise, in the `Button` element, the `android:text` attribute corresponds to the `setText()` setter on the class.

**The View and layout identifiers**

Each View can have a unique integer ID associated with it and can be used to reference the View from within an application's code. In the XML file, the ID is specified as a user-friendly text name. For example, consider the following line of code:

```xml
android:id="@+id/searchButton"
```

In this example, the `@` operator tells the parser that it should treat the remainder of the string as an ID resource; the `+` symbol tells the parser that this is a new resource name that should be added to the resource file, `R.java`. The resource file defines integer constants that can be used to reference resources.
Using XML layouts from activities
XML layouts can easily be loaded by an application at runtime. This task is generally performed from within the `onCreate()` method of an activity using the `setContentView()` method. For example, consider the following line of code:

```java
setContentView(R.layout.main);
```

Intents
Intents are messages that can be sent to the various types of components in an Android app in order to request some type of action to be performed. Intents may be used to accomplish any of the following:

- Start an activity with the option of receiving a result
- Start or stop a service
- Notify the component of conditions, such as low battery or time zone change
- Request an action from another app, such as request the map app to display a location or request that the camera app take a picture and save it

Resources
Creating an Android application involves more than simply writing code. A rich mobile app requires things such as images, audio files, animations, menus, and style, just to name a few. The application framework provides APIs that can be used to load and utilize the various types of resources with your Android apps.

The R.java file
Resources are generally referenced from within an application using an integer constant that is automatically assigned when the resource is added to the project and compiled. These constants are placed in a Java source file named `R.java`. The following example shows the R.java class from a simple application:

```java
public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
    }
    public static final class id {
```
public static final int myButton=0x7f050000;
public static final int searchButton=0x7f050002;
public static final int searchCriteriaTextView=0x7f050001;
}
public static final class layout {
    public static final int main=0x7f030000;
    public static final int search=0x7f030001;
}
public static final class string {
    public static final int app_name=0x7f040001;
    public static final int hello=0x7f040000;
}

Summary
In this chapter, we have provided a concise and adequate introduction to the Android platform and the Android application's building blocks. We have also seen how the Android platform has evolved with rich features being added to each platform releases.

In the next chapter, we will turn our attention to Xamarin.Android and the facilities it provides to allow the Android development with .NET and C#.
Now that we have an understanding of the Android platform, let’s talk about Xamarin. In this chapter, we will take a look at the architecture of Xamarin.Android and how it facilitates the development of Android apps using C# and .NET. This chapter covers the following topics:

- The benefits and drawbacks of adopting Xamarin.Android
- What is Mono?
- Mono and Android Runtime side by side (peer objects)
- The Xamarin.Android binding libraries
- Development of IDE choices

**Getting started with Xamarin**

Xamarin is a San Francisco, California-based software company, which provides commercial software development tools, that leverage the Mono open source project in order to allow you to develop applications for Android, iOS, and Mac using C# and the .NET framework.

Xamarin brings a wide range of products to simplify the mobile application development in a cross-platform way. The following are some of the products offered by Xamarin:

- **The Xamarin platform**: Xamarin uses the open source implementation of the .NET framework called Mono. The Xamarin framework implementation includes its own compiler written in C# and the .NET libraries. The Xamarin platform includes the following products:
Xamarin.Android Architecture

- Xamarin.iOS: This is also known as MonoTouch. This is used for building native iOS applications using C# and .NET.
- Xamarin.Android: This is also known as Mono for Android or formally known as MonoDroid. This is used for building native Android applications using C# and .NET.
- Xamarin.Forms: In both Xamarin.Android and Xamarin.iOS, we cannot build a pure cross-platform application. The part of the application that is platform-independent can then be isolated and reused across the platform; however, you still need to write platform-specific code for designing the application's interface. This is where Xamarin.Forms comes into picture. Xamarin.Forms allows you to write the user-interface code that can be compiled for the iPhone, Android, and Windows Phone.
- Xamarin.Mac: This is also known as Mono for Mac. Xamarin.Mac allows you to develop fully native Mac applications in C# and .NET.
- Xamarin.Windows: This is also known as Mono for Windows. Xamarin.Windows allows you to develop a fully Windows App in C# and .NET.

The product we will use in this book for developing the Android application is Xamarin.Android.

- Development IDEs: Along with frameworks, it also brings the required development IDEs, such as Xamarin Studio and the Visual Studio plugin. Xamarin Studio is a fully integrated IDE, which comes handy with the Xamarin package. Xamarin Studio can be used for both Windows and Mac operating systems. Xamarin Studio includes some of the rich features, including code completion, a debugging interface, an Android layout builder, and integration with Xcode Interface Builder for iOS app design. However, if you're familiar with Visual Studio, you can continue to take almost all the benefits mentioned earlier from Visual Studio using the Visual Studio support plugin.

- Xamarin Test Cloud: Mobile application testing is quite challenging, as we have to consider the various form factors, device densities, connectivity types, and different OS versions. It is nearly impossible to test your application on all the targeted devices. Xamarin Test Cloud is an answer to this problem. Xamarin Test Cloud makes it possible to test mobile apps written in any language on collection of real devices from around the world. You can write your test scripts using the Xamarin testing framework and automate your app testing from CI Systems.
Why Xamarin.Android?

Before we take a dive into the architecture of Xamarin.Android, let’s first discuss the question of why Xamarin.Android is our choice. Like any significant platform decision, one size does not fit all, and there are a number of things that should be considered. The following two lists identify some of the key benefits and drawbacks of using Xamarin.Android.

The benefits of using Xamarin.Android

• **It leverages existing C# and .NET skills**: Developers invest a great deal of time and energy in mastering the many features of the C# language and the effective use of the .NET framework. Yes, Java and all object-oriented languages have many similarities, but there is a real cost associated with going from being proficient in C# and .NET to making the same claim in Java. Individuals and groups that have made a significant investment in C# and .NET and need to develop Android apps would be well served to at least consider Xamarin.Android.

• **It can be reused in cross-platform development**: While Xamarin will not allow you to build a single app that can be deployed to Android, iOS, and Windows, it does give you the capability to reuse large portions of your code base across all of these platforms. In general, the user interface code and the code that deals with the device capabilities tend to be written for each platform, while things such as service client logic, client-side validation, data caching, and client-side data storage can potentially be shared across multiple platforms. This can save a significant amount of time and cost.

The drawbacks of using Xamarin.Android

• **The licensing requirement**: Xamarin.Android as well as Xamarin.iOS and Xamarin.Mac are all commercial tools and must be licensed, so there is a tangible cost of entry. Check the Xamarin website for current pricing.

• **Waiting for updates**: There is some lag time between a new release of the Android platform and the corresponding release of Xamarin.Android. However, Xamarin is aiming for zero day support for the new version of Android and iOS.

• **Distribution size**: There are a number of runtime libraries that must be distributed with a Xamarin.Android application. We will discuss the actual size and strategies for minimizing the distribution size in the last chapter.
While the list of drawbacks may seem extensive; in most cases, the impact of each can be minimized. If you are a group or individual that places a high value on the benefits, you should seriously consider Xamarin.Android.

What is Mono?

Mono is an open source, cross-platform implementation of a C# compiler, and a Common Language Runtime (CLR) that is binary compatible with Microsoft .NET. The Mono CLR has been ported to many platforms, including Android, most Linux distributions, BSD, OS X, Windows, Solaris, and even some game consoles, such as Wii and Xbox 360. In addition, Mono provides a static compiler that allows apps to be compiled for environments, such as iOS and PS3.

Mono for Android runs natively and provides almost all the capabilities that a typical native Android app can have. It allows developers to reuse a larger portion of the code without a major performance trade-off.

Mono and Dalvik side by side

As you can recall from, Chapter 1, The Anatomy of an Android App, Android apps run within the Dalvik VM, and we now know that Mono apps run within the Mono CLR. So how does a Xamarin.Android app run? A simple answer is that it uses both the Mono CLR and the Dalvik VM. The following diagram depicts how the runtimes coexist:
Xamarin.Android applications use both Mono CLR and the Dalvik VM side by side and run on top of the Linux kernel. The .Net API resides as a part of the Mono CLR and provides a set of classes (for example, System.Data, System.Net, System.IO, and so on.) to access various device OS features. However, with .Net APIs, you cannot directly access most of the device-specific features, such as Audio, Telephony, OpenGL, and so on. They are made available as a part of the Android SDK or as Java API and can be accessed using the Android binding libraries. The following section covers more on the Android binding libraries.

Since Android 5.0 (Lollipop) release, the Dalvik VM was replaced by its successor, Android Runtime (ART). This means that now Xamarin.Android applications run with the Mono VM alongside ART. Both the runtimes work on top of the Linux kernel and expose a set of classes to access the device features.

So, how do the Mono CLR and Android Runtime (ART) work together in a Xamarin.Android app? The magic is accomplished through a concept called and a framework called the JNI.

The Java Native Interface

The Java Native Interface (JNI) is a framework that allows a non-Java code (such as C++ or C#) to call or be called by a Java code running inside a JVM. As you can see from the preceding diagram, JNI is a critical component in the overall Xamarin.Android architecture.

Peer objects

Peer objects are a pair of objects consisting of a managed object residing in the Mono CLR and a Java object residing in the Dalvik VM, which work together to perform the functions of a Xamarin.Android app.

Xamarin.Android is delivered with a set of assemblies called the Android binding libraries. Classes in the Android binding libraries correspond to the Java classes in the Android application framework, and the methods in the binding classes act as wrappers to call corresponding methods on Java classes. Binding classes are referred to as Managed Callable Wrappers (MCW). Anytime you create a C# class that inherits from one of these binding classes, a corresponding Java proxy class is generated at build time. The Java proxy contains a generated override for each overridden method in your C# class and acts as a wrapper to call the corresponding method on the C# class.
The creation of peer objects can be initiated from within the Dalvik VM by the Android application framework or from within the Mono CLR by the code you write in the overridden methods. A reference between the two peer objects is kept by each instance of a MCW and can be accessed through the `Android.Runtime.IJavaObject.Handle` property.

The following diagram depicts how peer objects collaborate:

**Xamarin.Android application packaging**

In *Chapter 1, The Anatomy of an Android App*, we discussed Android packages (.apk files). Xamarin.Android creates the .apk files but also includes the following additional types of files:

- The C# code is stored as assemblies (containing IL) in the assembly folder of the archive.
- The Mono runtime is packaged as native libraries within the apk. The Xamarin.Android application must contain the native libraries for the desired Android architectures. If it doesn't contain the required libraries, the application will fail to run for those architectures.
The Android bindings design
The core parts of Xamarin.Android are the bindings for the Android APIs. The Xamarin team focused a great deal in developing a consistent approach to create the bindings so that a C# .NET developer would feel at home when using them. This has resulted in a number of key benefits as follows:

• The Android API feels natural to a C# .NET developer and allows the developer to explore the API using the code completion and pop-up documentation from within the IDE
• C# developers can leverage the vast array of Java/Android examples and documentation that can be easily transformed for use with C# and Xamarin.Android

Design principles
The following are some of the key design principles for the Xamarin.Android binding. A complete set of design principles can be found on the Xamarin website:

• Allowing developers to subclass Java classes from the Android application framework
• Exposing a strongly typed API
• Exposing JavaBean properties as C# properties
• Exposing Java event listeners as C# delegates

C# properties
The JavaBean properties, the getter and setter methods, are transformed to C# properties, when appropriate. The following rules are used to determine when properties should be created:

• Read/write properties are created for the getter and setter method pairs
• Read-only properties are created for getters without corresponding setter methods
• No write-only properties are created in the rare case that only a setter exists
• Properties are not created when the type would be an array
As you may be aware, Java does not have a property construct but instead follows a design pattern defined in the JavaBean specification. In order to define a property, a developer simply creates the public getter and setter methods with read-only properties that only provide a getter method.

Delegates
The Android APIs follow the Java pattern for defining and hooking up event listeners. The C# developers are more familiar with using delegates and events, so the Android bindings attempt to facilitate this using the following rules:

- When a listener callback has a `void` return, an event is generated based on the `EventHandler` delegate
- When a listener callback does not have a `void` return, a specific delegate is generated that supports the appropriate signature

These events or properties are only created under the following conditions:

- The Android event handling method has a prefix, for example, `setOnClickListener`
- The Android event handler has a `void` return type
- The Android event handler has a single parameter

Constants to enumerations
It is common in the Android APIs to see methods that accept or return an `int` type that must be mapped to a constant to determine its meaning. When possible, the Xamarin team creates a .NET enumeration to replace the constants and adjusts the appropriate methods to work with the enumerations. This provides a significant productivity gain by being able to use IntelliSense from within the IDE as well as enhance the type safety of the methods.
Development environments

The choice of an appropriate IDE for development is absolutely mandatory, as it can greatly ease and speed up your development, if you choose the right one. There are two choices when it comes to IDEs: Xamarin Studio or Visual Studio.

For developing an iOS application, you can use either use Xamarin Studio or the Xamarin iOS Visual Studio plugin on Windows machines. However, you cannot build and run an iOS application on Windows OS. You must have a Mac computer.

Windows users have two IDE choices for developing the Android application. You can either use Xamarin Studio or Visual Studio. If you're on Mac OS, then you have to use the Android Studio IDE. All the examples in this book are developed using Xamarin Studio on Mac.

The following section lists some of the unique features that come handy with both Xamarin Studio and the Visual Studio IDE.

Xamarin Studio

Xamarin Studio is a customized version of the MonoDevelop IDE, which can be used to develop Android, iOS, and OS X applications. Xamarin Studio is available on both OS X and Windows and has many advanced features as follows:

- Code completion
- Smart syntax highlighting
- Code navigation
- Code tooltips
- Integrated debugging for mobile apps running on emulators or on devices
- Source control integration with Git and subversion built-in
- The Xamarin component store
- The NuGet package browser
The following screenshot shows Xamarin Studio on Mac OS with the Android user interface designer opened:

Xamarin for Visual Studio
Xamarin for Visual Studio is an add-in that supports the development of the Xamarin.Android and Xamarin.iOS apps. The Visual Studio add-in for Xamarin requires at least a business or enterprise license. It is not available for basic Indie lenience users. If you already have a license for Visual Studio and are comfortable with the environment, the add-in will likely be more appealing than Xamarin Studio because of the simplicity of adoption. Apart from the basic features, such as code completion, syntax highlighter, smart navigation and tooltip, the Xamarin Visual Studio add-in extends the IDE capabilities to make the mobile development painless. The following are some of the specialized features from Xamarin for Visual Studio that are add-ins:

- **IntelliSense**: This helps developers take a quick look up at the language reference for both iOS and Android APIs.
- **Visual Designer**: With Visual Designers, you don't necessarily need to remember all the properties of a view, while building UI layouts for multiple resolutions. This also integrates with a property editor that enables easy property configurations such as color, font, size, margin, view ID, and so on.
The following screenshot shows Visual Studio 2012 with the Android user interface designer opened:

![Visual Studio 2012 with Android user interface](image)

### IDE comparison

It is not a surprise that every IDE provides the basic core functionalities for developers and some of the unique features of its own. The following table depicts the pros and cons of two different IDE choices available for the Xamarin Android development:

<table>
<thead>
<tr>
<th>IDE</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
</table>
| Xamarin Studio | • It comes with Xamarin.Android and no additional license is required. It runs on Windows and OS X. | • It runs on both Windows and Mac OS for the Xamarin. Android license.  
|              |                                                                      | • By default, it does not support the use of TFS for source control.  |
### Compatibility

The solution and project files created and updated by Xamarin Studio are compatible with Visual Studio, making it easy to switch between the two environments throughout the duration of a project. This also allows the team members to adopt the tool that they are most comfortable with or that runs on their platform of choice.

### Summary

In this chapter, we discussed the architecture of Xamarin.Android and the magic of how it facilitates the creation of Android apps using C# and .NET. We also reviewed a set of benefits and drawbacks of adopting Xamarin.Android. In the next chapter, we will install Xamarin.Android and create a project that we will build for the remainder of the book.
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:

![Product Selection 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:

![Configuration Screenshot]
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:

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:
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**:

![Installation Finished](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.
Creating the Points Of Interest App

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 Screenshot](image)

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:

![Create new Android Virtual Device (AVD)](image)

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:

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

![Configure your new project]

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.
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.
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:
Creating the Points Of Interest App

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.
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 POIAp project, right-click on it, and click on Options.


7. Change the application name to POIAp. This will cause the resulting APK file to be named POIAp.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.POIAp" >
<application
android:allowBackup="true"
android:icon="@drawable/ic_launcher"
android:label="POIAp"
android:theme="@style/AppTheme" >
<activity
android:name=".MainActivity"
android:label="POIAp" >
<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]

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:

   ```java
   // Set our view from the "main" layout resource
   setContentView (Resource.Layout.Main);
   ```

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:

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:

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\ poiapp 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
<?xml version="1.0" encoding="utf-8"?>
<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="md56a0a1b7026a6184892491193f52dfa6.MainActivity" />
    </application>
</manifest>
```
<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.
Adding a List View

In this chapter, we finally get to what many of you have been waiting for, developing the user interface. We will walk you through the activities related to creating and populating a ListView, which includes the following topics:

- Creating the POIApp activity layout
- Creating a custom list row item layout
- The ListView andListAdapter classes
- Extending BaseAdapter to provide data to the ListView widget
- Working with web services in Xamarin.Android
- Working with the ActionBar menu options
- Handling list item click events
- Handling the network state

Creating the POI ListView layout

It is technically possible to create and attach the user interface elements to your activity using C# code. However, it is a bit of a mess. We will go with the most common approach by declaring the XML-based layout. Keeping this in mind, let’s begin this chapter by creating a layout to display the POI list items.

When we created the new POIApp solution in the previous chapter (Chapter 3, Creating the Points Of Interest App), a default layout and activity was created as part of the Xamarin Studio project template.
Adding a List View

Rather than deleting these files, let's give them more appropriate names and remove unnecessary content as follows:

1. Select the Main.axml file in Resources | Layout and rename it to POIList.axml.
2. Double-click on the POIList.axml file to open it in a layout designer window.

   Currently, the POIList.axml file contains the layout that was created as part of the default Xamarin Studio template. As per our requirement, we need to add a ListView widget that takes the complete screen width and a ProgressBar in the middle of the screen. The indeterminate progress bar will be displayed to the user while the data is being downloaded from the server. Once the download is complete and the data is ready, the indeterminate progress bar will be hidden before the POI data is rendered on the list view.

3. Now, open the Document Outline tab in the designer window and delete both the button and LinearLayout.
4. Now, in the designer Toolbox, search for RelativeLayout and drag it onto the designer layout preview window.
5. Search for ListView in the Toolbox search field and drag it over the layout designer preview window. Alternatively, you can drag and drop it over RelativeLayout in the Document Outline tab.

We have just added a ListView widget to POIList.axml. Let's now open the Properties pad view in the designer window and edit some of its attributes:
As we may recall from Chapter 3, Creating the Points of Interest App, the **Properties** pad allows you to modify the properties of a selected widget. There are five buttons at the top of the pad that switch the set of properties being edited. The `@+id` notation notifies the compiler that a new resource ID needs to be created to identify the widget in API calls, and `listView1` identifies the name of the constant. Now, perform the following steps:

1. Change the ID name to `poiListView` and save the changes. Switch back to the **Document Outline** pad and notice that the `ListView` ID is updated.
2. Again, switch back to the **Properties** pad and click on the **Layout** button.
3. Under the **View Group** section of the layout properties, set both the **Width** and **Height** properties to `match_parent`.
4. The `match_parent` value for the **Height** and **Width** properties tells us that the `ListView` can use the entire content area provided by the parent, excluding any margins specified. In our case, the parent would be the top-level **RelativeLayout**.

Prior to API level 8, `fill_parent` was used instead of `match_parent` to accomplish the same effect. In API level 8, `fill_parent` was deprecated and replaced with `match_parent` for clarity. Currently, both the constants are defined as the same value, so they have exactly the same effect. However, `fill_parent` may be removed from the future releases of the API; so, going forward, `match_parent` should be used.

So far, we have added a `ListView` to **RelativeLayout**, let’s now add a **Progress Bar** to the center of the screen.

5. Search for **Progress Bar** in the **Toolbox** search field. You will now notice that several types of progress bars will be listed, including horizontal, large, normal, and small. Drag the normal progress bar onto **RelativeLayout**.

By default, the **Progress Bar** widget is aligned to the top left of its parent layout. To align it to the center of the screen, select the progress bar in the **Document Outline** tab, switch to the **Properties** view, and click on the **Layout** tab. Now select the **Center In Parent** checkbox, and you will notice that the progress bar is aligned to the center of the screen and will appear at the top of the list view.

6. Currently, the progress bar is visible in the center of the screen. By default, this could be hidden in the layout and will be made visible only while the data is being downloaded.
7. Change the **Progress Bar** ID to `progressBar` and save the changes.

8. To hide the **Progress Bar** from the layout, click on the **Behavior** tab in the **Properties** view. From **Visibility**, select **Box**, and then select **gone**.

   This behavior can also be controlled by calling `setVisibility()` on any view by passing any of the following behaviors. Later in this chapter, we will see how to hide the view programmatically using the activity code.

The `View.Visibility` property allows you to control whether a view is visible or not. It is based on the `ViewStates` enum, which defines the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gone</td>
<td>This value tells the parent <code>ViewGroup</code> to treat the View as though it does not exist, so no space will be allocated in the layout</td>
</tr>
<tr>
<td>Invisible</td>
<td>This value tells the parent <code>ViewGroup</code> to hide the content for the View; however, it occupies the layout space</td>
</tr>
<tr>
<td>Visible</td>
<td>This value tells the parent <code>ViewGroup</code> to display the content of the View</td>
</tr>
</tbody>
</table>

Click on the **Source** tab to switch the IDE context from visual designer to code, and see what we have built so far. Notice that the following code is generated for the `POIList.axml` layout:

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:p1="http://schemas.android.com/apk/res/android"
    p1:layout_width="match_parent"
    p1:layout_height="match_parent"
    p1:id="@+id/relativeLayout1">
    <ListView
        p1:minWidth="25px"
        p1:minHeight="25px"
        p1:layout_width="match_parent"
        p1:layout_height="match_parent"
        p1:id="@+id/poiListView" />
    <ProgressBar
        p1:layout_width="wrap_content"
        p1:layout_height="wrap_content"
        p1:id="@+id/progressBar"
        p1:layout_centerInParent="true"
        p1:visibility="gone" />
</RelativeLayout>
Creating POIListActivity

When we created the POIApp solution, along with the default layout, a default activity (MainActivity.cs) was created. Let's rename the MainActivity.cs file to POIListActivity.cs:

1. Select the MainActivity.cs file from Solution Explorer and rename to POIListActivity.cs.
2. Open the POIListActivity.cs file in the code editor and rename the class to POIListActivity.
3. The POIListActivity class currently contains the code that was created automatically while creating the solution using Xamarin Studio. We will write our own activity code, so let's remove all the code from the POIListActivity class.
4. Override the OnCreate() activity life cycle callback method. This method will be used to attach the activity layout, instantiate the views, and write other activity initialization logic. Add the following code blocks to the POIListActivity class:

   ```csharp
   namespace POIApp
   {
       [Activity (Label = "POIApp", MainLauncher = true, Icon = "@drawable/icon")]
       public class POIListActivity : Activity
       {
           protected override void OnCreate (Bundle savedInstanceState)
           {
               base.OnCreate (savedInstanceState);
           }
       }
   }
   ```

5. Now let's set the activity content layout by calling the SetContentView(layoutId) method. This method places the layout content directly into the activity's view hierarchy. Let's provide the reference to the POIList layout created in previous steps.

   ```csharp
   namespace POIApp
   {
       [Activity (Label = "POIApp", MainLauncher = true, Icon = "@drawable/icon")]
       public class POIListActivity : Activity
       {
           protected override void OnCreate (Bundle savedInstanceState)
           {
               base.OnCreate (savedInstanceState);
               SetContentView (ResourceId.Layout.POIList);
           }
       }
   }
   ```

At this point, the POIListActivity class looks as follows:
Notice that in the preceding code snippet, the POIListActivity class uses some of the [Activity] attributes such as Label, MainLauncher, and Icon. During the build process, Xamarin.Android uses these attributes to create an entry in the AndroidManifest.xml file. As we have already learnt form Chapter 1, The Anatomy of an Android App, the AndroidManifest.xml file is one of the simple application configuration files that describes the functionality and requirements of your Android application. Xamarin makes it easier by allowing all of the Manifest properties to set using attributes so that you never have to modify them manually in AndroidManifest.xml.

So far, we have declared an activity and attached the layout to it. At this point, if you run the app on your Android device or emulator, you will notice that a blank screen will be displayed. The following sections in this chapter will walk you through the magic of making the POIListActivity activity fully functional.

Creating the POI list row layout

We now turn our attention to the layout for each row in the ListView widget. The Android platform provides a number of default layouts out of the box that can be used with a ListView widget:

<table>
<thead>
<tr>
<th>Layout</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleListItem1</td>
<td>A single line with a single caption field</td>
</tr>
<tr>
<td>SimpleListItem2</td>
<td>A two-line layout with a larger font and a brighter text color for the first field</td>
</tr>
<tr>
<td>TwoLineListItem</td>
<td>A two-line layout with an equal sized font for both lines and a brighter text color for the first line</td>
</tr>
<tr>
<td>ActivityListItem</td>
<td>A single line of text with an image view</td>
</tr>
</tbody>
</table>

All of the preceding three layouts provide a pretty standard design, but for more control over content layout, a custom layout can also be created, which is what is needed for poiListView.

To create a new layout, perform the following steps:

1. In the Solution pad, navigate to Resources | Layout, right-click on it, and navigate to Add | New File.
2. Select **Android** from the list on the left-hand side, **Android Layout** from the template list, enter **POIListItem** in the name column, and click on **New**.

Before we proceed to lay out the design for each of the row items in the list, we must draw on a piece of paper and analyze how the UI will look like. In our example, the POI data will be organized as follows:

There are a number of ways to achieve this layout, but we will use **RelativeLayout** to achieve the same result. There is a lot going on in this diagram. Let's break it down as follows:

- A **RelativeLayout** view group is used as the top-level container; it provides a number of flexible options for positioning relative content, its edges, or other content.
- An **ImageView** widget is used to display a photo of the POI, and it is anchored to the left-hand side of the **RelativeLayout** utility.
- Two **TextView** widgets are used to display the POI name and address information. They need to be anchored to the right-hand side of the **ImageView** widget and centered within the parent **RelativeLayout** utility. The easiest way to accomplish this is to place both the **TextView** classes inside another layout; in this case, a **LinearLayout** widget with the orientation set to vertical.
- An additional **TextView** widget is used to display the distance, and it is anchored on the right-hand side of the **RelativeLayout** view group and centered vertically.

Now, our task is to get this definition into **POIListItem.axml**. The next few sections describe how to accomplish this using the Content view of the designer when feasible and the Source view when required.
Adding a RelativeLayout view group

The RelativeLayout layout manager allows its child views to be positioned relative to each other or relative to the container or another container. In our case, for building the row layout, as shown in the preceding diagram, we can use RelativeLayout as a top-level view group. When the POIListItem.axml layout file was created, by default a top-level LinearLayout was added. First, we need to change the top-level ViewGroup to RelativeLayout. The following section will take you through the steps to complete the layout design for the POI list row:

1. With POIListItem.axml opened in the content mode, select the entire layout by clicking on the content area. You should see a blue outline going around the edge. Press Delete. The LinearLayout view group will be deleted, and you will see a message indicating that the layout is empty.

2. Alternatively, you can also select the LinearLayout view group from the Document Outline tab and press Delete.

3. Locate the RelativeLayout view group in the toolbox and drag it onto the layout.

4. Select the RelativeLayout view group from Document Outline. Open the Properties pad and change the following properties:
   - The Padding option to 5dp
   - The Layout Height option to wrap_content
   - The Layout Width option to match_parent

   The padding property controls how much space will be placed around each item as a margin, and the height determines the height of each list row. Setting the Layout Width option to match_parent will cause the POIListItem content to consume the entire width of the screen, while setting the Layout Height option to wrap_content will cause each row to be equal to the longest control.

5. Switch to the Code view to see what has been added to the layout. Notice that the following lines of code have been added to RelativeLayout:

   `<RelativeLayout xmlns:p1="http://schemas.android.com/apk/res/android"
   p1:minWidth="25px"
   p1:minHeight="25px"
   p1:layout_width="match_parent"
   p1:layout_height="wrap_content"
   p1:id="@+id/relativeLayout1"
   p1:padding="5dp"/>

   `
Android runs on a variety of devices that offer different screen sizes and densities. When specifying dimensions, you can use a number of different units, including pixels (px), inches (in), and density-independent pixels (dp). Density-independent pixels are abstract units based on 1 dp being 1 pixel on a 160 dpi screen. At runtime, Android will scale the actual size up or down based on the actual screen density. It is a best practice to specify dimensions using density-independent pixels.

### Adding an ImageView widget

The `ImageView` widget in Android is used to display the arbitrary image for different sources. In our case, we will download the images from the server and display them in the list. Let's add an `ImageView` widget to the left-hand side of the layout and set the following configurations:

1. Locate the `ImageView` widget in the toolbox and drag it onto `RelativeLayout`.
2. With the `ImageView` widget selected, use the `Properties` pad to set the ID to `poiImageView`.
3. Now, click on the `Layout` tab in the `Properties` pad and set the `Height` and `Width` values to 65 dp.
4. In the property grouping named `RelativeLayout`, set `Center Vertical` to `true`. Simply clicking on the checkbox does not seem to work, but you can click on the small icon that looks like an edit box, which is to the right-hand side, and just enter `true`. If everything else fails, just switch to the `Source` view and enter the following code:

   ```xml
   p1:layout_centerVertical="true"
   ```

5. In the property grouping named `ViewGroup`, set the `Margin Right` to 5dp. This brings some space between the POI image and the POI name.
6. Switch to the `Code` view to see what has been added to the layout. Notice the following lines of code added to `ImageView`:

   ```xml
   <ImageView
       p1:src="@android:drawable/ic_menu_gallery"
       p1:layout_width="65dp"
       p1:layout_height="65dp"
       p1:layout_marginRight="5dp"
       p1:id="@+id/poiImageView" />
   ```
Adding a LinearLayout widget

LinearLayout is one of the most basic layout managers that organizes its child views either horizontally or vertically based on the value of its orientation property. Let's add a LinearLayout view group that will be used to lay out the POI name and address data as follows:

1. Locate the LinearLayout (vertical) view group in the toolbox.
2. Adding this widget is a little trickier because we want it anchored to the right-hand side of the ImageView widget.
3. Drag the LinearLayout view group to the right-hand side of the ImageView widget until the edge turns to a blue dashed line, and then drop the LinearLayout view group. It will be aligned with the right-hand side of the ImageView widget.
4. In the property grouping named RelativeLayout of the Layout section, set Center Vertical to true. As before, you will need to enter true in the edit box or manually add it to the Source view.
5. Switch to the Code view to see what has been added to the layout. Notice the following lines of code added to LinearLayout:

```xml
<LinearLayout
    p1:orientation="vertical"
    p1:minWidth="25px"
    p1:minHeight="25px"
    p1:layout_width="wrap_content"
    p1:layout_height="wrap_content"
    p1:layout_toRightOf="@id/poiImageView"
    p1:id="@+id/linearLayout1"
    p1:layout_centerVertical="true" />
```

Adding the name and address TextView classes

Add the TextView classes to display the POI name and address:

1. Locate TextView in the Toolbox and add a TextView class to the layout. This TextView needs to be added within the LinearLayout view group we just added, so drag TextView over the LinearLayout view group until it turns blue and then drop it.
2. Name the `TextView` ID as `nameTextView` and set the text size to 20sp. The text size can be set in the **Style** section of the **Properties** pad; you will need to expand the **Text Appearance** group by clicking on the ellipsis (…) button on the right-hand side.

   **Scale-independent pixels (sp)** are like dp units, but they are also scaled by the user's font size preference. Android allows users to select a font size in the Accessibility section of Settings. When font sizes are specified using sp, Android will not only take into account the screen density when scaling text, but will also consider the user's accessibility settings. It is recommended that you specify font sizes using sp.

3. Add another `TextView` to the `LinearLayout` view group using the same technique except dragging the new widget to the bottom edge of the `nameTextView` until it changes to a blue dashed line and then drop it. This will cause the second `TextView` to be added below `nameTextView`. Set the font size to 14sp.

4. Change the ID of the newly added `TextView` to `addrTextView`.

5. Now change the sample text for both `nameTextView` and `addrTextView` to **POI Name** and **City, State, Postal Code**.

6. To edit the text shown in `TextView`, just double tap the widget on the content panel. This enables a small editor that allows you to enter the text directly. Alternately, you can change the text by entering a value for the **Text** property in the **Widget** section of the **Properties** pad.

7. It is a design practice to declare all your static strings in the `Resources/values/string.xml` file. By declaring the strings in the `strings.xml` file, you can easily translate your whole app to support other languages. Let's add the following strings to `string.xml`:

   ```xml
   <string name="poi_name_hint">POI Name</string>
   <string name="address_hint">City, State, Postal Code.</string>
   ```

8. You can now change the **Text** property of both `nameTextView` and `addrTextView` by selecting the ellipsis (…) button, which is next to the **Text** property in the **Widget** section of the **Properties** pad. Notice that this will open a dialog window that lists all the strings declared in the `string.xml` file. Select the appropriate strings for both `TextView` objects.
9. Now let's switch to the **Code** view to see what has been added to the layout. Notice the following lines of code added inside `LinearLayout`:

```xml
<TextView
  p1:layout_width="match_parent"
  p1:layout_height="wrap_content"
  p1:id="@+id/nameTextView"
  p1:textSize="20sp"
  p1:text="@string/app_name" />
<TextView
  p1:text="@string/address_hint"
  p1:layout_width="match_parent"
  p1:layout_height="wrap_content"
  p1:id="@+id/addrTextView"
  p1:textSize="14sp" />
```

### Adding the distance TextView

Add a `TextView` to show the distance from POI:

1. Locate the `TextView` in the toolbox and add a `TextView` to the layout. This `TextView` needs to be anchored to the right-hand side of the `RelativeLayout` view group, but there is no way to visually accomplish this; so, we will use a multistep process. Initially, align the `TextView` with the right-hand edge of the `LinearLayout` view group by dragging it to the left-hand side until the edge changes to a dashed blue line and drop it.

2. In the **Widget** section of the **Properties** pad, name the widget as `distanceTextView` and set the font size to 14sp.

3. In the **Layout** section of the **Properties** pad, set **Align Parent Right** to true, **Center Vertical** to true, and clear out the `linearLayout1` view group name in the **To Right Of** layout property.

4. Change the sample text to **204 miles**. To do this, let's add a new string entry to `string.xml` and set the **Text** property from the **Properties** pad.
The following screenshot depicts what should be seen from the Content view at this point:

![Screenshot](image-url)

Switch back to the **Source** tab in the layout designer, and notice the following code generated for the `POILListItem.axml` layout:

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:p1="http://schemas.android.com/apk/res/android"
    p1:minWidth="25px"
    p1:minHeight="25px"
    p1:layout_width="match_parent"
    p1:layout_height="wrap_content"
    p1:id="@+id/relativeLayout1"
    p1:padding="5dp">
    <ImageView
        p1:src="@android:drawable/ic_menu_gallery"
        p1:layout_width="65dp"
        p1:layout_height="65dp"
        p1:layout_marginRight="5dp"
        p1:id="@+id/poiImageView" />
    <LinearLayout
        p1:orientation="vertical"
        p1:layout_width="wrap_content"
        p1:layout_height="wrap_content"
        p1:layout_toRightOf="@id/poiImageView"
        p1:layout_marginRight="5dp"
        p1:id="@+id/linearlayout" />
</RelativeLayout>
```
Creating the PointOfInterest apps entity class

The first class that is needed is the one that represents the primary focus of the application, a PointofInterest class. POIAApp will allow the following attributes to be captured for the Point Of Interest app:

- Id
- Name
- Description
- Address
- Latitude
- Longitude
- Image
The POI entity class can be nothing more than a simple .NET class, which houses these attributes.

To create a POI entity class, perform the following steps:

1. Select the POIApp project from the Solution Explorer in Xamarin Studio. Select the POIApp project and not the solution, which is the top-level node in the Solution pad.
2. Right-click on it and select New File.
3. On the left-hand side of the New File dialog box, select General.
4. At the top of the template list, in the middle of the dialog box, select Empty Class (C#).
5. Enter the name PointOfInterest and click on OK. The class will be created in the POIApp project folder.
6. Change the visibility of the class to public and fill in the attributes based on the list previously identified.

The following code snippet is from \POIApp\POIApp\PointOfInterest.cs from the code bundle available for this book:

```csharp
public class PointOfInterest
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public string Address { get; set; }
    public string Image { get; set; }
    public double? Latitude { get; set; }
    public double? Longitude { get; set; }
}
```

Note that the Latitude and Longitude attributes are all marked as nullable. In the case of latitude and longitude, (0, 0) is actually a valid location so a null value indicates that the attributes have never been set.

**Populating the ListView item**

All the adapter views such as ListView and GridView use an Adapter that acts as a bridge between the data and views. The Adapter iterates through the content and generates Views for each data item in the list.
Adding a List View

The Android SDK provides three different adapter implementations such as ArrayAdapter, CursorAdapter, and SimpleAdapter. An ArrayAdapter expects an array or a list as input, while CursorAdapter accepts the instance of the Cursor, and SimpleAdapter maps the static data defined in the resources. The type of adapter that suits your app need is purely based on the input data type.

The BaseAdapter is the generic implementation for all of the three adapter types, and it implements the IListAdapter, ISpinnerAdapter, and IDisposable interfaces. This means that the BaseAdapter can be used for ListView, GridView, or Spinners.

For POIApp, we will create a subtype of BaseAdapter<T> as it meets our specific needs, works well in many scenarios, and allows the use of our custom layout.

Creating POIListViewAdapter

In order to create POIListViewAdapter, we will start by creating a custom adapter as follows:

1. Create a new class named POIListViewAdapter.
2. Open the POIListViewAdapter class file, make the class a public class, and specify that it inherits from BaseAdapter<PointOfInterest>.

Now that the adapter class has been created, we need to provide a constructor and implement four abstract methods.

Implementing a constructor

Let's implement a constructor that accepts all the information we will need to work with to populate the list.

Typically, you need to pass at least two parameters: an instance of an activity because we need the activity context while accessing the standard common resources and an input data list that can be enumerated to populate the ListView. The following code shows the constructor from the code bundle:

```csharp
private readonly Activity context;
private List<PointOfInterest> poiListData;
public POIListViewAdapter (Activity _context, List<PointOfInterest> _poiListData)
    :base()
```
Implementing Count { get }

The `BaseAdapter<T>` class provides an abstract definition for a read-only Count property. In our case, we simply need to provide the count of POIs as provided in `poiListData`. The following code example demonstrates the implementation from the code bundle:

```csharp
public override int Count {
    get {
        return poiListData.Count;
    }
}
```

Implementing GetItemId()

The `BaseAdapter<T>` class provides an abstract definition for a method that returns a long ID for a row in the data source. We can use the `position` parameter to access a POI object in the list and return the corresponding ID. The following code example demonstrates the implementation from the code bundle:

```csharp
public override long GetItemId (int position)
{
    return position;
}
```

Implementing the index getter method

The `BaseAdapter<T>` class provides an abstract definition for an index getter method that returns a typed object based on a position parameter passed in as an index. We can use the position parameter to access the POI object from `poiListData` and return an instance. The following code example demonstrates the implementation from the code bundle:

```csharp
public override PointOfInterest this[int index] {
    get{
        return poiListData [index];
    }
}
```
Adding a List View

Implementing GetView()

The `BaseAdapter<T>` class provides an abstract definition for `GetView()`, which returns a view instance that represents a single row in the `ListView` item. As in other scenarios, you can choose to construct the view entirely in code or to inflate it from a layout file. We will use the layout file we previously created. The following code example demonstrates inflating a view from a layout file:

```java
view = context.LayoutInflater.Inflate (Resource.Layout.POILListItem, null, false);
```

The first parameter of `Inflate` is a resource ID and the second is a root `ViewGroup`, which in this case can be left `null` since the view will be added to the `ListView` item when it is returned.

Reusing row Views

The `GetView()` method is called for each row in the source dataset. For datasets with large numbers of rows, hundreds, or even thousands, it would require a great deal of resources to create a separate view for each row, and it would seem wasteful since only a few rows are visible at any given time. The `AdapterView` architecture addresses this need by placing row Views into a queue that can be reused as they scroll out of view of the user. The `GetView()` method accepts a parameter named `convertView`, which is of type `view`. When a view is available for reuse, `convertView` will contain a reference to the view; otherwise, it will be `null` and a new view should be created. The following code example depicts the use of `convertView` to facilitate the reuse of row Views:

```java
var view = convertView;
if (view == null){
    view = context.LayoutInflater.Inflate (Resource.Layout.POILListItem, null);
}
```

Populating row Views

Now that we have an instance of the view, we need to populate the fields. The `View` class defines a named `FindViewById<T>` method, which returns a typed instance of a widget contained in the view. You pass in the resource ID defined in the layout file to specify the control you wish to access.

The following code returns access to `nameTextView` and sets the `Text` property:

```java
PointOfInterest poi = this [position];
view.FindViewById<TextView>((Resource.Id.nameTextView).Text = poi.Name;
```
Populating addrTextView is slightly more complicated because we only want to use the portions of the address we have, and we want to hide the TextView if none of the address components are present.

The View.Visibility property allows you to control the visibility property of a view. In our case, we want to use the ViewState.Gone value if none of the components of the address are present. The following code shows the logic in GetView:

```csharp
if (String.IsNullOrEmpty(poi.Address)) {
    view.FindViewById<TextView>(Resource.Id.addrTextView).Visibility = ViewStates.Gone;
} else{
    view.FindViewById<TextView>(Resource.Id.addrTextView).Text = poi.Address;
}
```

Populating the value for the distance text view requires an understanding of the location services. We need to do some calculation, by considering the user's current location with the POI latitude and longitude. This part will be covered in Chapter 9, Making POIApp Location Aware.

**Populating the list thumbnail image**

Image downloading and processing is a complex task. You need to consider the various aspects, such as network logic, to download images from the server, caching downloaded images for performance, and image resizing for avoiding the memory out conditions. Instead of writing our own logic for doing all the earlier mentioned tasks, we can use UrlImageViewHelper, which is a free component available in the Xamarin Component Store.

The Xamarin Component Store provides a set of reusable components, including both free and premium components, that can be easily plugged into any Xamarin-based application.

**Using UrlImageViewHelper**

The following steps will walk you through the process of adding a component from the Xamarin Component Store:

1. To include the UrlImageViewHelper component in POIApp, you can either double-click on the Components folder in the Solution pad, or right-click and select Edit Components.
2. Notice that the component manager will be loaded with the already downloaded components and a Get More Components button that allows you to open the Components store from the window. Note that to access the component manager, you need to log in to your Xamarin account:

3. Search for UrlImageViewHelper in the components search box available in the left-hand side pane. Now click on the download button to add your Xamarin Studio solution.

4. Now that we have added the UrlImageViewHelper component, let's go back to the GetView() method in the POIListViewAdapter class. Let's take a look at the following section of the code:

```csharp
var imageView = view.FindViewById<ImageView>(Resource.Id.poiImageView);
if (!String.IsNullOrEmpty(poi.Address)) {
    Koush.UrlImageViewHelper.SetUrlDrawable(imageView, poi.Image, Resource.Drawable.ic_placeholder);
}
```

Let us examine how the preceding code snippet works:

1. The SetUrlDrawable() method defined in the UrlImageViewHelper component provides a logic to download an image using a single line of code. It accepts three parameters: an instance of imageView, where the image is to be displayed after the download, the image source URL, and the placeholder image.

2. Add a new image ic_placeholder.png to the drawable Resources directory. While the image is being downloaded, the placeholder image will be displayed on imageView.

---

[90]
3. Downloading the image over the network requires Internet permissions. The following section will walk you through the steps involved in defining permissions in your AndroidManifest.xml file.

**Adding Internet permissions**

Android apps must be granted permissions while accessing certain features, such as downloading data from the Internet, saving an image in storage, and so on. You must specify the permissions that an app requires in the AndroidManifest.xml file. This allows the installer to show potential users the set of permissions an app requires at the time of installation.

To set the appropriate permissions, perform the following steps:

1. Double-click on AndroidManifest.xml in the Properties directory in the Solution pad. The file will open in the manifest editor. There are two tabs: Application and Source, at the bottom of the screen, that can be used to toggle between viewing a form for editing the file and the raw XML, as shown in the following screenshot:
Adding a List View

2. In the Required permissions list, check Internet and navigate to File | Save.
3. Switch to the Source view to view the XML as follows:

![AndroidManifest.xml](image)

Hooking up POIListViewAdapter

We have the list layout and adapter ready; let's now proceed to hook up the data in poiListView. We need to switch back to the POIListActivity class and add the following changes:

1. Declare the following variables inside the POIListActivity class:
   ```java
   private ListView poiListView;
   private ProgressBar progressBar;
   private List<PointOfInterest> poiListData;
   private POIListViewAdapter poiListAdapter;
   ```

2. Now, in the OnCreate method, instantiate the ListView and ProgressBar:
   ```java
   poiListView = FindViewById<ListView> (Resource.Id.poiListView);
   progressBar = FindViewById<ProgressBar> (Resource.Id.progressBar);
   ```

3. For now, we will create an Async method that is responsible for downloading the data from the server and displaying it in POIListActivity. Add the following method to the POIListActivity class:
   ```java
   public async void DownloadPoisListAsync(){
   }
   ```

4. Call the DownloadPoisListAsync() method from the OnCreate() activity life cycle callback.
5. Note that this chapter uses the Android device networking capabilities for downloading the data form a REST web service. The following sections in this chapter will cover how to make a network request fetch data from the server in detail.

Now for testing purposes, let’s add the following method that provides a dummy POI list object. Later in this chapter, we will remove this method while we integrate with a REST web service:

```csharp
private List<PointOfInterest> GetPoisListTestData(){
    List<PointOfInterest> listData = new List<PointOfInterest> ();
    for(int i=0; i<20; i++){
        PointOfInterest poi = new PointOfInterest ();
        poi.Id = i;
        poi.Name = "Name " + i;
        poi.Address = "Address " + i;
        listData.Add (poi);
    }
    return listData;
}
```

6. Let’s add the following logic to the `DownloadPoisAsync()` method to make our POIApp fully functional:

```csharp
public async void DownloadPoisListAsync(){
    progressBar.Visibility = ViewStates.Visible;
    poiListData = GetPoisListTestData();
    progressBar.Visibility = ViewStates.Gone;

    poiListAdapter = new POIListViewAdapter (this, poiListData);
    poiListView.Adapter = poiListAdapter;
}
```

Notice the following in the preceding code:

- The progress bar is shown to the user when the download starts. When the download is complete, the progress bar is hidden.
- Currently, the `GetPoisListTestData()` method simulates the network request and provides the list of POI objects.
- Once the data download is complete, the `POIListViewAdapter` class is instantiated by passing the downloaded POI list result and then it is set to list view.
Adding a List View

We have done a great deal of work so far! Let's just build and run the app on the Android emulator or device. You will see the output, as shown in the following screenshot:

![POIApp](image)

The ListView works great with the test data. Now it is the time to worry about consuming the real POI data from the REST web service. The following section will guide you through establishing an HTTP network request form the Android device.

**Consuming the web service**

Until now, we have created the layout for the list view and list adapter; now it is the time to worry about how to consume the web service to download the data and to hook it up to the screen. The following section will walk you through the steps on how to download data asynchronously from the web service.
An introduction to web services

Web services are one of the integral parts of the World Wide Web (WWW) infrastructure. It allows the server application to share the data or logic with connected clients via web protocols such as REST and SOAP using data formats, such as XML and JSON.

Web services expose a set of Application Programming Interfaces (APIs) that provides a uniform data access mechanism for client applications. It doesn't matter which programming language the web service is written in; the client applications with different operating systems or different programming languages can access the services seamlessly, as long as they adhere to the web service API specification. For example, a web service that is written in Java and hosted on Apache Tomcat can be used by a .NET web form, iOS, or an Android application.

SOAP and REST are the two standard web service architectures used vastly in the industry by larger players. Microsoft developed SOAP and REST was developed by W3C Technical Architecture Group. While SOAP brings its own protocol with additional security layers, the REST implementation is considerably easier. Companies such as Google and Microsoft are migrating most of their existing services to REST. Which web service architecture is better is an endless debate worth Googling; however, choosing the one that works for you is purely an architectural decision based on your requirements.

The POIApp example code provided in this book will consume the web service developed in Java (JAX-RS) using the REST architecture. The following section will walk you through the steps on how to deploy and set up on your system for testing POIApp.

Deploying the POI web service

The code bundle provided in this book includes the POI web service project code, which will be used for completing the remaining chapters in this book. The code bundle includes a readme file describing the steps required for deploying the POI Web service. The following section will walk you through the steps provided in the readme file and deploy the web service before you proceed with this chapter.

The POI web service sample application provides two APIs: one to fetch the list of POIs available on the server and the other to create a new POI record in the server database.
In this chapter, we will be using the following API specification for getting the list of POIs from the web server:

- **Request method**: GET
- **Resource URI**: /com.packet.poiapp/api/poi/pois
- **Content-type**: application/json
- **Accept-type**: application/json
- **Response Body**:

```json
{
    "poi": [
        {
            "description": "The London Eye is a giant Ferris wheel on the South Bank of the River Thames in London",
            "id": "1",
            "latitude": "50.59938",
            "longitude": "80.8897",
            "address": "London SE17PB, UK",
            "name": "London Eye"
        },
        {
            ...
            ...
        },
        {
            ...
            ...
        }
    ]
}
```

### Consuming REST web services asynchronously

Consuming REST web services in the Xamarin Android application is relatively easier than it looks. There are various framework classes, such as `WebClient`, `WebRequest`, `HttpWebRequest`, `HttpClient`, and other third-party libraries, such as `RestSharp` and `Service Stack`, that are available for Xamarin to consume REST web services.
For developing POIApp, we will taper our discussion to HttpClient. The HttpClient is newly introduced in .NET 4.5 and provides some of the advanced features, such as a strong type header, shared cache, cache control, and so on. The following section will walk you through the steps on how to use HttpClient to make an asynchronous web service request from the Xamarin Android application.

**Creating the POIService class**

Now, we will create a standard C# class that abstracts all the logic to consume REST web services and will make our activity look much tidier. To create the new POIService class, perform the following steps:

1. Select the POIApp project in the Solution pad in Xamarin Studio.
2. Right-click on it and select New File.
3. On the left-hand side of the New File dialog box, select General.
4. At the top of the template list, in the middle of the dialog box, select Empty Class(C#).
5. Enter the name POIService and click on OK.
6. Declare a string constant that represents the resource's URI endpoint to access the POI web service for fetching the list of POIs available in the server. At this stage, I assume that the web service code is hosted on the local computer with port 8080. In real time, your application will use the domain where the endpoint is hosted:

   ```csharp
   
   For any technical reasons/limitations you couldn't complete the web service installation, you can still continue to test your application with Apiary mock feeds. For testing with mock feeds, you can use the following feed URL:
   
   ```csharp
   ```
Adding a List View

Asynchronous programming with async and await

Downloading data from the server is a long-running blocking operation, and it is recommended for mobile apps to perform all such long-running tasks off the main thread. For making a responsive and smooth user experience, mobile applications need to create a new thread for any long-running operations. Since the release of .NET 4.5, the async and await keywords are used to implement multi-threading easily, without getting your hands dirty in threads. The methods defined by the async and await keywords are often called the async methods.

The following are some of the key things you must know before implementing the async methods:

- The async keyword is used to notify the .NET language Common Language Runtime (CLR) to create a new thread of execution and execute tasks asynchronously.
- The await keyword automatically pauses the caller thread and executes the tasks on a new thread; once the task is finished, the control is returned. A single async method can have one or more await keywords.
- The async method always returns Task<T>, where T represents the data type of the result expected after the execution. A void return type can be used when no result is expected.
- As a convention, Microsoft recommends the name of the async methods to suffix with async. However, this is not mandatory, but it helps to remind the caller to use the await keyword.

Now that we understand the basics to create the async method, let's define a method in the POIService class and name it GetPOIListAsync. The GetPOIListAsync method will be used to consume the REST web service asynchronously, deserialize the response in the POI list collection, and return the result back to the activity for displaying results in the list:

```csharp
public async Task<List<PointOfInterest>> GetPOIListAsync() {
}
```

Notice that the return type of the GetPOIListAsync method is Task<List<PointOfInterest>>, and the List<PointOfInterest> is the result expected on POIListActivity for rendering the data on the list view.
The GetPOIListAsync method now creates an instance of HttpClient, sets the HTTP header accept type as application/json, and calls the GetAsync() method by passing the web service URL. The accept type header metadata tells the server about the response the media format client expects. The GetAsync() method initiates a GET request to the specified endpoint and returns the HttpResponseMessage instance.

If the response status code is returned as a success (200OK), we are good to proceed to fetch the contents. As we know from the web service specification, the response is a structured JSON string; we can retrieve the values by calling the GetStringAsync method. For the status code other than success, we can place the logic to handle the error case. In order to simplify this example, we will just print the error log on the console:

```csharp
HttpClient httpClient = new HttpClient();
httpClient.DefaultRequestHeaders.Accept.Add(new
  MediaTypeWithQualityHeaderValue("application/json"));
HttpResponseMessage response = await httpClient.GetAsync(GET_POIS);
if (response != null || response.IsSuccessStatusCode) {
  string content = await response.Content.ReadAsStringAsync();
  Console.Out.WriteLine("Response Body: \r\n {0}", content);
} else {
  Console.Out.WriteLine("Failed to fetch data. Try again later!");
  return null;
}
```

Note that the preceding GetPOIListAsync method used for consuming the REST web service requires the Internet permission. As we have already added the Internet permission, while downloading the image using the UrlImageViewHelper Xamarin component, we don't need to add the permission again.

The await keyword expects the download results to proceed further and hence, it waits for the ongoing download task to complete. Once the download task is completed, the POI web service response JSON string is assigned to a content variable. At this point, the string result has to be deserialized to the .NET object and the result has to be returned back to POIListActivity. The following section will guide you with the steps to deserialize the content string into a JSON object.

While HttpClient in combination with async and await keyword provides a native API support for dealing with asynchronous network request, you can also take advantages of the powerful and yet popular framework such as Service Stack or Rest Sharp. Here are a few links that you can refer to:

- https://github.com/restsharp/RestSharp
- https://servicestack.net/
Serializing and deserializing using Json.NET

Another important decision that we need to make while serializing and deserializing the JSON data is how we will get the response string converted to a .NET object and vice versa. There are a number of options available, including DataContractJsonSerializer from .NET. Json.NET is an open source component library created by James Newton-King, and this is definitely worth considering because of the following reasons:

- It's small, fast, and reliable
- It's available as a free component in the Xamarin Component Store and via NuGet
- It makes simple tasks extremely simple to accomplish

With these characteristics in mind, we will proceed by adding the Json.NET component to POIApp from the Xamarin Component Store. To add the Json.NET component, follow the same steps that we performed while adding the UrlImageViewHelper component.

Once the Json.NET component is added to the solution, the next step is to convert the response string and to make a list of PointOfInterest objects.

1. Include the following namespace directives in the POIListActivity.cs file:
   ```csharp
   using Newtonsoft.Json.Linq;
   using System.Collections.Generic;
   using Newtonsoft.Json;
   using System.Linq;
   ```

2. Declare a list collection in the GetPOIListAsync() method to hold the POI's list response:
   ```csharp
   private List<PointOfInterest> poiListData = null;
   ```

3. Initialize the poiListData list collection. Place the following code snippet in the downloadPoisAsync() method after the Console.Out.WriteLine statement:
   ```csharp
   poiListData = new List<PointOfInterest> ();
   ```

Now, it is time to deserialize the JSON string to the .NET object. Notice that the response JSON string is an object with a key `pois` representing the array of the PointOfInterest objects. Add the following code snippet to the GetPOIListAsync() method to deserialize the string to the .NET object.
The following line of code will convert the complete JSON response string to JObject:

```csharp
 JObject jsonResponse = JObject.Parse (content);
```

Now fetch the values for the pois key and iterate through it to convert to the .NET list:

```csharp
 IList<JToken> results = jsonResponse ["pois"].ToList ();
 foreach (JToken token in results) {
     PointOfInterest poi = token.ToObject<PointOfInterest>();
     poiListData.Add (poi);
 }
 return poiListData;
```

The JToken is a generic representation of the JSON value of any kind. It could be a string, object, array, property, and so on. The ToList() method returns the collections of the JToken objects. The token.ToObject method converts each of the poiJSON tokens to the PointOfInterest object type and adds it to the poiListData collection. Finally, we return the list result back to the caller.

### Updating POIListActivity

So far, we have downloaded the data, and we are ready to use the data in POIListActivity. Let's do the following changes in POIListActivity:

1. Delete the GetPoiListTestData() method from the POIListActivity class.
2. Create an instance of POIService in the DownloadPoisListAsync method and call the GetPOIListAsync method:

```csharp
 public async void DownloadPoisListAsync(){
     progressBar.Visibility = ViewStates.Visible;
     POIService service = new POIService ();
     poiListData = await service.GetPOIListAsync ();
     progressBar.Visibility = ViewStates.Gone;

     poiListAdapter = new POIListViewAdapter (this, poiListData);
     poiListView.Adapter = poiListAdapter;
 }
```
We have done a great deal of work so far; now it's time to compile and run the app. Compile and run the application using the Android emulator based on the procedure we used in the previous chapters.

Bingo! You will notice that the app will download the data from the POI web service and display the list of POIs on the scrollable list view:

![POIApp](image)

Adding actions to ActionBar

With Android 3.0 (Honeycomb, API level 11), Android introduced a uniform title, such as a widget that docks to the top of the screen, called **ActionBar**. It allows apps to add activity-specific actions to the top of the device screen, just below the status bar. We will define two actions for the `POIListActivity` class: New, to create a new POI, and Refresh, to refresh the cache of POIs from the device's local storage.
The `Activity` class provides the following virtual methods that can be overridden to add actions:

<table>
<thead>
<tr>
<th>Virtual Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnCreateOptionsMenu</td>
<td>It allows the creation of the actions either through API calls or through inflating an XML definition</td>
</tr>
<tr>
<td>OnOptionsItemSelected</td>
<td>It is called when an action in ActionBar is clicked</td>
</tr>
</tbody>
</table>

**Defining the menu XML file**

Actions can be defined in a menu XML file that resides in the `Resources/menu` folder, or it can be created programmatically using API calls. We will define the **New** and **Refresh** actions in an XML file named `POIListViewMenu.xml`.

To create `POIListViewMenu.xml`, perform the following steps:

1. Select the `Resources` folder in `POIApp`, right-click on it, and navigate to Add | New Folder.
2. Name the folder menu.
3. Select the menu folder, right-click on it, and navigate to Add | New File.
4. Navigate to XML | Empty XML file, enter `POIListViewMenu.xml` for the name, and click on New.

You now need to fill in the definitions for the two actions we identified. Unfortunately, Xamarin Studio does not contain a template for menu XML files, so you have to hunt the format down from the Android documentation or online examples. The following code contains definitions for `actionNew` and `actionRefresh`:

```xml
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/actionNew"
        android:icon="@drawable/ic_new"
        android:title="New"
        android:showAsAction="ifRoom" />
  <item android:id="@+id/actionRefresh"
        android:icon="@drawable/ic_refresh"
        android:title="Refresh"
        android:showAsAction="ifRoom" />
</menu>
```
Adding a List View

Note that from the menu definition, we have referenced two new drawables: ic_new and ic_refresh. We need to add these images to the project in the same way that we did for the ic_app icon in Chapter 3, Creating the Points of Interest App. The images can be found in the drawable folder present in the assets location.

Setting menus in OnCreateOptionsMenu()

The OnCreateOptionsMenu() method is called to give an opportunity to the Activity parameter to define actions for the ActionBar. The Activity class provides a MenuInflater method, which reads the XML definition file and places the action defined on the ActionBar. The following code shows the implementation from the code bundle:

```csharp
public override bool OnCreateOptionsMenu(IMenu menu) {
    MenuInflater.Inflate(Resource.Menu.POIListViewMenu, menu);
    return base.OnCreateOptionsMenu(menu);
}
```

Handling selection in OnOptionsItemSelected()

The OnOptionsItemSelected() method is called whenever an action in the ActionBar is clicked and an instance of IMenuItem is passed in. The IMenuItem ItemId instance corresponds to the ID specified in the item definition and can be used to determine which action was clicked on. The following code shows the implementation of OnOptionsItemSelected() from the code bundle:

```csharp
public override bool OnOptionsItemSelected(IMenuItem item) {
    switch (item.ItemId) {
    case Resource.Id.actionNew:
        // place holder for creating new poi
        return true;
    case Resource.Id.actionRefresh:
        DownloadPoisListAsync(url);
        return true;
    default:
        return base.OnOptionsItemSelected(item);
    }
}
```
Note that we have simply created a placeholder for `actionNew` and placed two method calls for `actionRefresh`.

The `DownloadPoisListAsync()` method is called to download and refresh the data on the list.

Let’s now run the `POIApp` and notice the two buttons, **Add** and **Refresh**, on the POI list activity title:

![POIApp screenshot with POIs](image)

### Handling the ListView click event

When a user clicks on a row, the POI app will navigate to a detailed view in order to allow you to view and update the complete set of information. We will build the detailed view in the next chapter but will go ahead and discuss handling clicks now.

Clicks can be handled using a traditional event handler. The `ListView` item provides an `ItemClick` event handler, which accepts a `ListView.ItemClickEventArgs` parameter. The `ListView.ItemClickEventArgs` parameter provides the following information that can be used for processing the event:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>It is the ID for the data associated with the row that was clicked. This would be the value returned from <code>GetItemId()</code>.</td>
</tr>
<tr>
<td>Position</td>
<td>It is the position in the <code>ListView</code> item of the row that was clicked.</td>
</tr>
<tr>
<td>View</td>
<td>It is the view associated with the row that was clicked. This would be the view returned from <code>GetView()</code>.</td>
</tr>
<tr>
<td>Parent</td>
<td>It is the <code>AdapterView</code> architecture that contains the row that was clicked. In our case, it is <code>ListView</code>.</td>
</tr>
</tbody>
</table>
Adding a List View

Create an event handler in POIListActivity to process click events on the ListView item. We are not ready to add the navigation, as we have not yet created our detailed view, so we will just show you a Toast message. The following code is from the code bundle:

protected void POIClicked(object sender, ListView.ItemClickEventArgs e)
{
    // Fetching the object at user clicked position
    PointOfInterest poi = result.poiListData[(int)e.Id];
    Console.Out.WriteLine("POI Clicked: Name is \{0\}", poi.Name);
}

We also need to hook up the event handler. Add the following line of code to the end of the OnCreate method:

poiListView.ItemClick += POIClicked;

Run the POIApp project and click on a POI; notice that the POI name of the corresponding row is printed on the console.

Handling no network condition

The network conditions in mobile devices are uncertain. Sometimes, a user manually disables the network connections or they are unavailable due to various external reasons. For applications that use network data, you must handle different network states. Applications should react gracefully by showing an appropriate message to the user.

In POIApp, before starting the download, we must confirm the availability of the network data connections. If the network is unavailable, we should notify the user with an appropriate message otherwise continue with the download request.

The ConnectivityManager class present in the System.Net package can be used to query the state of device network connectivity. This class can also be used to monitor the network connection and notify when there is a change in the network state. In our case, we will just query the network information just to know that the network is available.
Accessing the network state using the `ConnectivityManager` class requires `ACCESS_NETWORK_STATE` user permission in the `AndroidManifest.xml` file.

Follow the same steps as we did earlier while adding the Internet permission, or alternatively, you can directly add the following code to the `AndroidManifest.xml` source editor:

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Now add the following utility method to the `POIService` class that reads the network information and returns `true` if the device is connected:

```csharp
public bool isConnected(Context activity)
{
    var connectivityManager = (ConnectivityManager)activity.GetService(Context.ConnectivityService);
    var activeConnection = connectivityManager.ActiveNetworkInfo;
    return (null != activeConnection && activeConnection.IsConnected);
}
```

Now do the following changes to the `DownloadPoisListAsync` method in the `POIService` class:

```csharp
public async void DownloadPoisListAsync()
{
    POIService service = new POIService();
    if (!service.isConnected(this)) {
        Toast toast = Toast.MakeText(this, "Not connected to internet. Please check your device network settings.", ToastLength.Short);
        toast.Show();
    } else {
        progressBar.Visibility = ViewStates.Visible;
        poiListData = await service.GetPOIListAsync();
        progressBar.Visibility = ViewStates.Gone;

        poiListAdapter = new POIListViewAdapter(this, poiListData);
        poiListView.Adapter = poiListAdapter;
    }
}
```

You must be surprised about the `Toast` method. Well, we will learn about it in the next section.
Adding a List View

Toast

The Toast is a noninteractive, auto disposable view used to display a short message for a specified period of time and disposes itself. Android recommends that you use Toast only to notify the user, where the user's attention is not mandate. For any such notification that requires the user's attention or interaction, consider using dialog.

Creating a Toast is simple, all you have to do is to call the MakeText() static method by passing three parameters: the application context, message to be shown, and the time duration for the Toast to be shown. The time duration is a non-negative integer value in milliseconds, but it is recommended that you use the standard Long and Short constants defined in the ToastLength enum. The MakeText() method initializes the Toast with the given properties and returns a Toast instance on which we can call the Show() method to display Toast.

In the preceding code snippet, we are displaying the Toast message when the device is offline. The following code depicts calls to the MakeText() and Show() methods to display Toast on the Delete action:

```java
Toast toast = Toast.MakeText (this, "Not connected to internet. Please check your device network settings.", ToastLength.Short);
toast.Show ();
```

Summary

In this chapter, we covered a lot about how to create user interface elements using different layout managers and widgets such as TextView, ImageView, ProgressBar, and ListView.

We also covered how to consume REST web services using the Xamarin.Android HttpClient class, deserialize the JSON response using the Json.NET component, and populate the data on the screen.

The next chapter will introduce you to some more view groups and build more complex UIs. We will continue with POIApp by adding a detailed view and allow the user the option to create or delete a Point Of Interest.
In this chapter, we will walk you through the process of creating a new activity to display the details of the selected POI. This activity also allows the user to create, update, and delete POIs from the server. The following topics will be covered in this chapter:

- Creating the layout and activity for displaying POI details
- Using LinearLayout, TableLayout, and ScrollView to lay out the activity
- Passing data between the activities using Intent bundles
- Consuming web services to perform the HTTP POST and DELETE operations
- Performing the EditText validation using the EditText.Error property
- Displaying confirmation prompts

Creating the POIDetail layout

So far, we have built the POIApp to display the list of POIs fetched from the server. Currently, the list view displays limited information about POIs such as name, address, image, and distance. Let's now extend the POIApp by adding another activity that displays detailed information of the selected POI. The POI details activity will be revealed only when the user taps on any of the POI item from the list or when the user selects the New (+) action from the navigation bar. Additionally, the details activity will allow the user to create a new POI, update, or delete the existing POI.
Adding a Details View

Keeping all the preceding use cases in mind, let's create a new layout for POI details:

1. Select the Resources/layout folder in the Solution pad.
2. Right-click on Add and select New File.
3. In the New File dialog box, click on Android and select Layout, enter POI detail in the Name field, and select New.

Notice that a new file is created with LinearLayout as the top-level container. The POIDetail view will have a number of fields and will likely require scrolling on devices with a smaller screen size. The default Android layout managers, such as LinearLayout, RelativeLayout, FrameLayout, or TableLayout, do not provide automatic scrolling when the content grows and the data goes beyond its actual screen size.

In our previous chapter, for displaying the POI list, the layout gave us scrolling for free because we were using ListView; however, in the case of the POI detail activity, we need to use ScrollView for making the items scrollable.

Understanding ScrollView

ScrollView is a special kind of layout designed to hold a view larger than its actual size. When the child’s Views size grows beyond the ScrollView size, it automatically adds the scroll bar and can scroll vertically. The following are some of the key things you must to know before using ScrollView:

- ScrollView can hold at most one direct child. This means that if you have a complex layout with multiple children, then you must enclose them inside another standard layout, such as LinearLayout, TableLayout, or RelativeLayout.
- Like any other standard layout manager, the layout_height and layout_width attributes can be used to adjust the height and width of ScrollView.
- ScrollView is ideal for screens where scrolling is required, but it is an overhead when scroll view is used to render a larger collection of data. For such instances, you should consider using specialized adapter views, such as ListView and GridView.
- Never place a ListView or GridView inside ScrollView because they both take care of their own vertical scrolling. By doing so, the ListView child will never receive gestures because it will be handled by the parent ScrollView.
ScrollView only supports vertical scrolling. For horizontal scrolling, you can use HorizontalScrollView.

- The android:fillViewport property defines whether the ScrollView should stretch its content to fill the viewport. You can set the same property by calling the setFillViewport(true) method on ScrollView.

Now that we understand ScrollView, let's jump back to the POIDetail layout and add a ScrollView to support vertical scrolling of content for devices of smaller screen size:

1. With the POIDetail.xaml file opened in the Content view, select the top-level LinearLayout and press the Delete key.
2. In the Toolbox pad, locate the ScrollView widget and drag it onto the Content view.
3. In the Toolbox pad, locate the LinearLayout (vertical) widget and drag it onto the Content view inside ScrollView.
4. With LinearLayout selected, set Padding in the Layout section of the Properties pad to 5dp.

We are now ready to add labels and edit controls to the layout. The following screenshot depicts the layout we are trying to achieve:

![POIDetailActivity](image)
We will use simple `TextView` widgets to serve as labels and `EditText` widgets to serve as input controls. The `EditText` widget contains a number of properties that can be used to customize its behavior. One of these properties is named `InputType`, and it controls aspects such as which type of keyboard to use for inputs (alpha, numeric, and so on) and how many lines of text are allowed. The Toolbox pad presents a number of templates or preconfigured `EditText` widgets in the list under the group name `Text Fields`.

The following screenshot depicts the list:
Add a series of TextView and EditText controls to the Name, Description, and Address fields. Name the EditText widgets according to the following table and use the corresponding Toolbox widget so that the appropriate editing characteristics are applied:

<table>
<thead>
<tr>
<th>Name</th>
<th>Widget toolbox name</th>
</tr>
</thead>
<tbody>
<tr>
<td>nameEditText</td>
<td>Plain Text</td>
</tr>
<tr>
<td>descrEditText</td>
<td>Multiline Text</td>
</tr>
<tr>
<td>addrEditText</td>
<td>Multiline Text</td>
</tr>
</tbody>
</table>

We are now ready to address the Latitude and Longitude fields, and we will employ a new layout manager, TableLayout.

**Using the TableLayout manager**

The TableLayout manager is an extension of LinearLayout. As the name indicates, TableLayout is used to align a child's View elements in rows and columns format.

The concept of TableLayout works something similar to an HTML table. TableLayout consists of the `<table>` tag and TableRow is like a `<tr>` element. You can add any view or view group inside a table cell.

We would like to add the Latitude and Longitude fields in a table. Let's add a TableLayout with two rows and two columns with the top row being used for the labels and the bottom row being used for the edit fields. In order to do so, perform the following steps:

1. Locate TableLayout in the Toolbox pad, drag it onto the Content view below the addrEditText widget and drop it. A TableLayout will be created with three rows and three columns.
2. Select one of the rows in TableLayout, right-click on it and select Delete Row.
3. Select one of the columns in the TableLayout, right-click on it and select Delete Column.
4. Select the first column, right-click on and select Stretch Column. Similarly, do it for the second column.
Adding a Details View

You should now have a `TableLayout` with a visible outline of two rows, each having two columns like what is depicted in the following screenshot:

We now need to add the `TextView` widgets for the two labels in the first row and number (decimal) widgets for the `Latitude` and `Longitude` edit controls, naming them `latEditText` and `longEditText`.

We have now completed the `POIDetail` layout, and the Content view you see should look the same as the preceding screenshot.

Working with EditText's InputType

The `EditText` element provides a property named `InputType` that checks the behavior of the control when data is being entered. When the description and address widgets were added, we selected `Multiline Text` from the `Toolbox` pad. The following code shows that in this case the `inputType` was automatically set:

```xml
<EditText
    android:inputType="textMultiLine"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/descrEditText" />
```

The `InputType` property can also be set or changed from within the `Properties` pad under the `Widget` tab in the `Input Format` section. What may not be obvious is that `inputType` can combine values, which come in handy in our situation. The following table shows a reasonable set of values for `inputType`; feel free to experiment:

<table>
<thead>
<tr>
<th>Widget</th>
<th>Input type</th>
</tr>
</thead>
<tbody>
<tr>
<td>nameEditText</td>
<td><code>inputType=&quot;textCapWords&quot;</code></td>
</tr>
<tr>
<td>descrEditText</td>
<td>`inputType=&quot;textMultiLine</td>
</tr>
<tr>
<td>Widget</td>
<td>Input type</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>addrEditText</td>
<td>inputType=&quot;textMultiLine&quot;</td>
</tr>
<tr>
<td>latEditText</td>
<td>inputType=&quot;numberDecimal|numberSigned&quot;</td>
</tr>
<tr>
<td>longEditText</td>
<td>inputType=&quot;numberDecimal|numberSigned&quot;</td>
</tr>
</tbody>
</table>

Switch to the Code view to see what has been added to the layout:

```xml
<ScrollView xmlns:p1="http://schemas.android.com/apk/res/android"
    p1:minWidth="25px"
    p1:minHeight="25px"
    p1:layout_width="match_parent"
    p1:layout_height="match_parent"
    p1:id="@+id/scrollView1">
    <LinearLayout
        p1:orientation="vertical"
        p1:minWidth="25px"
        p1:minHeight="25px"
        p1:layout_width="fill_parent"
        p1:layout_height="fill_parent"
        p1:id="@+id/linearLayout1"
        p1:padding="5dp">
        <TextView
            p1:text="Name"
            p1:layout_width="fill_parent"
            p1:layout_height="wrap_content"
            p1:id="@+id/textView10" />
        <EditText
            p1:inputType="textMultiLine\|textCapSentences"
            p1:layout_width="fill_parent"
            p1:layout_height="wrap_content"
            p1:id="@+id/nameEditText" />
        <TextView
            p1:text="Description"
            p1:layout_width="fill_parent"
            p1:layout_height="wrap_content"
            p1:id="@+id/textView11" />
        <EditText
            p1:inputType="textMultiLine\|textCapSentences"
            p1:layout_width="fill_parent"
            p1:layout_height="wrap_content"
```
Adding a Details View

```xml
<ScrollView>
  <LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <TextView
      android:id="@+id/descrEditText"
      android:text="Address"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"/>
    <EditText
      android:id="@+id/addrEditText"
      android:inputType="textMultiLine"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"/>
    <TableLayout
      android:id="@+id/tableLayout1"
      android:stretchColumns="*"
      android:minWidth="25px"
      android:minHeight="25px"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content">
      <TableRow
        android:id="@+id/tableRow2">
        <TextView
          android:id="@+id/textView21"
          android:text="Latitude"
          android:layout_column="0"/>
        <TextView
          android:id="@+id/textView22"
          android:text="Longitude"
          android:layout_column="1"/>
      </TableRow>
      <TableRow
        android:id="@+id/tableRow3">
        <EditText
          android:id="@+id/latEditText"
          android:inputType="numberDecimal|numberSigned"
          android:layout_column="0"/>
        <EditText
          android:id="@+id/longEditText"
          android:inputType="numberDecimal|numberSigned"
          android:layout_column="1"/>
      </TableRow>
    </TableLayout>
  </LinearLayout>
</ScrollView>
```
Creating POIDetailActivity

Now that we have the POIDetail layout ready, we need a corresponding activity. Perform the following steps to create a new activity for the POI details:

1. With the POIApp project selected in the Solution pad, right-click on it and navigate to Add | New File.
2. In the New File dialog, click on Android and select Activity, enter POIDetailActivity as Name, and click on New.

As you may recall from Chapter 3, Creating the Points Of Interest App, one of the first things that needs to be done when an activity is created is to set the layout content, which is accomplished by calling setContentView(layoutId). Add the following line of code to the OnCreate() method of the POIDetailActivity:

```java
setContentView (Resource.Layout.POIDetail);
```

Adding navigation to POIDetailActivity

There are two scenarios where we need to navigate from POIListActivity to POIDetailActivity, either by selecting the New action on the navigation bar or by selecting any of the POIs from the list. The main difference between the two scenarios is that when selecting an existing POI from the list, we will have to pass the details of the selected POI to the POIDetailActivity so that the user can edit and update the POI record. When creating a new POI, we don't pass any data. Let's first choose the simplest one, and add the navigation to the New action.

The Activity class provides a method named StartActivity() that is used to start a new activity. The StartActivity() method can also be used in a number of different scenarios. In case of the New action, we will be using it in its most basic form. For starting an activity, all that we need to do is to invoke StartActivity() by passing the type for the activity we want to start.

The following code snippet demonstrates the code that needs to be added to the placeholder that we added to POIListActivity in the OnOptionsItemSelected() method:

```java
case Resource.Id.actionNew:
    startActivity (typeof(POIDetailActivity));
    return true;
```
Let's now run the app on the Android device or simulator and see what we have built so far. In POIListActivity, click on the New button on the navigation bar, and notice that the newly created POIDetailActivity is opened:

![POIDetailActivity](image)

### Navigation on the POIList item click

In the second scenario, we need to pass the details of the POI to display POIDetailActivity. To accomplish this, we will use the Intent class. The Intent class can be used in conjunction with `startActivity()` in order to launch a new activity and pass information about the activity to be launched. We will use the Intent class to launch POIDetailActivity and pass the details of the selected POI.

First, we need to construct an instance of Intent by providing the current activity context and the type of target activity that will receive the intent; in our case, POIDetailActivity. The following code demonstrates how to properly construct the intent:

```java
Intent poiDetailIntent = new Intent(this, typeOf(POIDetailActivity));
```
The **Intent** object has an **Extras** property that is used to send extra data as a bundle from one activity to another. The **Intent** class provides a series of overloaded versions of the **PutExtra()** method, which allows you to add various types of **key/value** pairs to the **Extras** property. The values can be of any primitive types such as `int`, `boolean`, `char`, `string`, `double`, `float`, `long`, and so on. For example, to add the string data type, you can use the following syntax:

```java
intent.PutExtra("YOUR_KEY", "SOME STRING VALUE HERE");
```

For **POIApp**, we need to add all of the properties of the selected **PointOfInterest** object. Instead of passing each property with a distinct key, we can further simplify this process using the **Json.NET** component. It serializes the **PointOfInterest .NET** object into a JSON string that can be passed to **POIDetailActivity** with the distinct key `poi`, and at the receiver end, we will deserialize it back to the **PointOfInterest** object.

Apart from using the **Intent** bundle, there are various other options to pass data between activities. For complex objects, serializing using the **Json.NET** component and passing data bundles using Internet is not a recommended option. The size limit for the Android intent bundle is limited to somewhere around 1 MB, so you should always be cautioned about your data limits. You may consider using any of the other alternatives for better performance.

Some of my recommendations are as follows:

- Write the data to a file and pass only the file path to the second activity. The second activity can access the content using the same file path.
- Use SQLite to store the object in a table. Pass only the unique ID or query a parameter to the second activity. The second activity can access the data by reading from the database.
- Create a singleton class to hold the data. The second activity can directly access the data using a singleton instance.

In our case, the POI object is tiny. We are happy to go with the **Intent** bundle approach. The following code syntax converts the selected **PointOfInterest** object to a JSON string using the **SerializeObject()** method of the **JsonConvert** class and adds it to **Intent** using the **PutExtra()** method:

```java
string poiJson = JsonConvert.SerializeObject(poi);
poiDetailIntent.PutExtra("poi", poiJson);
```

The last step is a call to **StartActivity()**, passing in the **Intent** class we created in the earlier step:

```java
StartActivity(poiDetailIntent);
```
Adding a Details View

With this, your POIListActivity OnListItemClick should have the following code:

```csharp
protected void POIClicked(object sender, ListView.ItemClickEventArgs e)
{
    PointOfInterest poi = poiListData[(int)e.Id];

    Intent poiDetailIntent = new Intent(this,
        typeof(POIDetailActivity));

    string poiJson = JsonConvert.SerializeObject(poi);
    poiDetailIntent.PutExtra("poi", poiJson);
    StartActivity(poiDetailIntent);
}
```

Now that we have passed the data from POIListActivity, let's now access the POI object from the POIDetailActivity class.

Receiving the data in POIDetailActivity

When we get to the OnCreate() method of the POIDetailActivity, we need to access the PointOfInterest object sent from POIListActivity. This object will be used to display the details of the selected POI. The following section will walk you through the process of retrieving the extra bundle metadata from Intent.

Each activity has an Intent property that contains the intent and the corresponding information that was passed while starting the activity. The Intent class provides a number of methods to access any of the Extras bundle data by providing the corresponding keys. Before we retrieve the data, we can confirm whether the value for the specified key is available by calling the HasExtra ("poi") method. The HasExtra method returns a boolean value; if it returns false, we can assume that we are creating a new POI; otherwise, we need to retrieve the value of the extra.

The Intent class has a series of GetXXExtra() methods, where XX represents the type for the value of a key/value pair. In our case, we can use the GetStringExtra() method on the intent to get the poiJson string passed in from POIListActivity. The GetStringExtra() method accepts a string (which is the key from the key/value pair originally set on the intent) and returns the value associated with the key.

Let's declare a PointOfInterest variable to hold the POI object received from POIListActivity:

```csharp
    private PointOfInterest _poi;
```
Add the following listing to the OnCreate() method of POIDetailActivity:

```java
if (Intent.HasExtra ("poi")) {
    string poiJson = Intent.GetStringExtra ("poi");
    _poi = JsonConvert.DeserializeObject<PointOfInterest>(poiJson);
} else {
    _poi = new PointOfInterest();
}
```

**Binding variables to controls**

As we learned in the previous chapter, we need to manually bind user interface widgets to internal program references in order to manipulate their content, assign event handlers, and so on. Declare a set of private variables for each of the input widgets we created in the layout. The following listing is from the source folder:

```java
private EditText _nameEditText;
private EditText _descrEditText;
private EditText _addrEditText;
private EditText _latEditText;
private EditText _longEditText;
```

A call to `FindViewById<T>` is required to bind each variable to its corresponding user interface widget. The following listing depicts what should be added to the `OnCreate()` method somewhere after the call to `SetContentView()`:

```java
SetContentView (Resource.Layout.POIDetail);
_nameEditText = FindViewById<EditText> (Resource.Id.nameEditText);
_descrEditText = FindViewById<EditText> (Resource.Id.descrEditText);
_addrEditText = FindViewById<EditText> (Resource.Id.addrEditText);
_latEditText = FindViewById<EditText> (Resource.Id.latEditText);
_longEditText = FindViewById<EditText> (Resource.Id.longEditText);
```

**Populating user interface widgets**

At this point, we have a reference to the `PointOfInterest` object, but we have not taken any action to populate the content on UI. Populating the POI details on UI is a pretty straightforward process.

The `EditText` widget has a property named `Text`, which we can set to initialize the content for the widget. Let's create a simple method named `UpdateUI()`, which takes care of populating the POI details on the user interface widgets.
Adding a Details View

The following listing shows what is needed for `UpdateUI`:

```csharp
protected void UpdateUI()
{
    _nameEditText.Text = _poi.Name;
    _descrEditText.Text = _poi.Description;
    _addrEditText.Text = _poi.Address;
    _latEditText.Text = _poi.Latitude.ToString();
    _longEditText.Text = _poi.Longitude.ToString();
}
```

Call the `UpdateUI` method at the end of the `OnCreate` callback.

You should be able to run `POIApp` now, and test the navigation by clicking on any one of the list rows in `POIListActivity`. Notice that the `POIDetailActivity` will display the details of the selected POI object:

![POIDetailActivity](image)

Name

**Buckingham Palace**

Description

Buckingham Palace is the London residence and principal workplace of the monarchy of the United Kingdom. Located in the City of Westminster, the palace is often at the centre of state occasions and royal hospitality. It has been a focus for the British people at times of national rejoicing.

Address

Latitude 51.501
Longitude 0.1416
Adding the save and delete actions

Using POIDetailActivity, users can choose to save or delete POIs. The same Save button works for two scenarios: when the POI details are passed from POIListActivity, it will update the POI details; otherwise, it will create a new record.

We need a way to accomplish these tasks from the user interface. Let's use ActionBar and add two actions: Save and Delete. Create a new file named POIDetailMenu.xml under the Resources/menu directory to declare the menu layout. The following listing shows what is needed for POIDetailMenu.xml:

```xml
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/actionSave"
    android:icon="@drawable/ic_save"
    android:title="Save"
    android:showAsAction="ifRoom" />
  <item android:id="@+id/actionDelete"
    android:icon="@drawable/ic_delete"
    android:title="Delete"
    android:showAsAction="ifRoom" />
</menu>
```

Note that each menu item has an icon specified. These icons can be found in the code bundle Assets folder.

We need to override both the OnCreateOptionsMenu() and OnOptionsItemSelected() methods. This is very similar to what we created in Chapter 4, Adding a ListView. Add the following code snippets to the POIDetailActivity class:

```csharp
public override bool OnCreateOptionsMenu(IMenu menu)
{
    MenuInflater.Inflate(Resource.Menu.POIDetailMenu, menu);
    return base.OnCreateOptionsMenu(menu);
}

public override bool OnOptionsItemSelected(IMenuItem item)
{
    switch (item.ItemId)
    {
        case Resource.Id.actionSave:
            SavePOI();
            return true;
    }
}
```
Adding a Details View

```csharp
    case Resource.Id.actionDelete:
        DeletePOI();
        return true;
    default:
        return base.OnOptionsItemSelected(item);
```

You may notice that, in the previous code snippets, we introduced two new methods: `SavePOI()` and `DeletePOI()`. Both of these methods are used to keep the `OnOptionsItemSelected()` method clean and concise. The `SavePOI()` and `DeletePOI()` methods encapsulate the logic required to save or delete the POI object.

### Disabling the delete action

One thing that’s different in `POIDetailView` is that we have a scenario where we need to disable the `Delete` action. If a new POI is being created, the `Delete` action should not be allowed. First, we need to get the reference of the menu item inside the `OnPrepareOptionsMenu()` method, and then we can call the `SetEnabled(bool)` method by passing your preference to enable or disable the menu action.

The `IMenu` class provides a `FindItem()` method that can be used to obtain a reference to a specific `IMenuItem`, which in turn provides the `SetEnabled()` method for enabling and disabling actions. Disabling a menu item makes the button nonfunctional; however, it will still be visible on the screen. For better user experience, let’s hide the `Delete` action completely by calling the `SetVisible(false)` method.

The following listing shows how to disable the `Delete` action when a new POI is being entered:

```csharp
    public override bool OnPrepareOptionsMenu(IMenu menu)
    {
        base.OnPrepareOptionsMenu(menu);
        // Disable delete for a new POI
        if (_poi.Id<=0) {
            IMenuItem item = menu.FindItem(Resource.Id.actionDelete);
            item.SetEnabled(false);
            item.SetVisible(false);
        }
        return true;
    }
```
Adding the save and delete actions in POIService

In *Chapter 4, Adding a ListView*, we created the specialized `POIService` class that handles the logic for downloading data using the `async` and `await` keywords in conjunction with `HttpClient`. For now, the `POIService` class has only one method `GetPoisListAsync()`, that deals with fetching the list of records from the REST web service. Let's extend its functionality to create a new or update and delete a POI. Let's first start with creating a POI.

Consuming the web service to create or update a POI

At this point, we have already deployed the web service provided in the book code bundle, and `POIApp` is already consuming the same web services to fetch the list of POIs. In this section, we will use the same web service to create a new or update an existing POI.

The following API specification is used to create a new or update an existing POI:

```
Request Method: POST
Resource Endpoint: /com.packet.poiapp/api/poi/pois
Content-type: application/json
Request Body:
{
    "description": "The London Eye is a giant Ferris wheel on the South Bank ...",
    "latitude": "50.59938",
    "longitude": "80.8897",
    "address": "London SE17PB, UK",
    "name": "London Eye"
}
Response: Success/Failed
```

Notice some of the key points in the preceding API specification: the request method is `POST`, `Content-type` is `application/json`, and the POI JSON object is sent as part of the request body. The `POST` method is an indication for the server application to lookup for the attached request content body, and the Content-Type describes the MIME type of the data the server is about to receive. In this case, the content type is `application/json`, which means that the server is expecting the POI details to be sent in the form of a JSON string.
Adding a Details View

To create a new POI, we need to send the POI details such as name, description, latitude, longitude, and address while creating a new POI record. Once the POI is created, a unique ID will be created and assigned for each POI record. To update the existing POI record, we must send the POI ID along with the updated POI details as part of the request body.

Adding the CreateOrUpdatePOIOIAsync method to POIService

Now that we understand that the details of the web service can be used to create or update a POI record, let's create a new async method named CreateOrUpdatePOIOIAsync() in the POIService class, and perform the following steps:

1. Create a new async method CreateOrUpdatePOIOIAsync() that accepts the PointOfInterest instance. This holds the details of the POI that you need to send to the server in order to create or update the operation. This method returns Task<String> because the result of the create/update request is returned as a string:

   ```csharp
   public async Task<String> CreateOrUpdatePOIOIAsync (PointOfInterest poi, Activity activity)
   {
   }  
   }
   ```

2. Declare a string constant that represents the web service endpoint in order to create a new or update an existing POI:

   ```csharp
   
   Alternatively, you can use the following Apiary mock API URL, if you have not set up the web server code already:
   ```csharp
   ```
3. As you might have noticed in the web service API specification, the server expects the POI details in a JSON string format. Hence, we need to serialize the POI object to a JSON using the Json.NET component:

   var poiJson = JsonConvert.SerializeObject(poi, Formatting.Indented);

The preceding code will convert the POI object to the following JSON format:

   {
      "Name": "Googleplex",
      "Description": "Google HQ",
      "Latitude": "37.423441",
      "Longitude": "-102.083962",
      "Address": "1600 Amphitheater Parkway Mountain View, CA 94044",
   }

The preceding JSON string contains the uppercase keys, such as Name, Address, which are incompatible for a given web service API specification. For this, we can use the ContractResolver class from Json.NET to provide custom settings while serializing the POI object.

4. Declare an inner class named POIContractResolver inside POIService and extend it from DefaultContractResolver. Override the ResolvePropertyName method. You need to include the Newtonsoft.Json.Serialization namespace directive in the POIService class:

   public class POIContractResolver : DefaultContractResolver
   {
      protected override string ResolvePropertyName(string key)
      {
         return key.ToLower();
      }
   }

The preceding code snippet is self-explanatory. It resolves all the PointOfInterest object properties to lowercase JSON keys.

5. Now we can use the following code snippet to serialize the POI object into a JSON string with lowercase keys. Add the following listing to the CreateOrUpdatePOIAsync method:

   var settings = new JsonSerializerSettings();
   settings.ContractResolver = new POIContractResolver();
   var poiJson = JsonConvert.SerializeObject(poi, Formatting.Indented, settings);
Let us now asynchronously send the POI details JSON data using HttpClient class. The HttpClient class provides PostAsync() method that is used to sends a POST request to the specified URI as an asynchronous operation. The following code snippet demonstrates posting data to the server using the HttpClient class:

```csharp
HttpClient httpClient = new HttpClient();
StringContent jsonContent = new StringContent(poiJson, Encoding.UTF8, "application/json");
HttpResponseMessage response = await httpClient.PostAsync(CREATE_POI, jsonContent);

if (response != null || response.IsSuccessStatusCode) {
    string content = await response.Content.ReadAsStringAsync();
    Console.Out.WriteLine("\{0\} saved.", poi.Name);
    return content;
}
return null;
```

The preceding code block looks familiar to the GetPoisListAsync method but with some obvious differences. Here, we are calling PostAsync instead of the GetAsync method to send an asynchronous POST request. The PostAsync method accepts two parameters: a string representing a web service URL and an instance of HttpContent for the HTTP entity body. The PostAsync method accepts the different HTTP entity body formats such as ByteArrayContent, MultipartContent, StreamContent, and StringContent, representing the request body and content header. Here, in our case, we are sending the POI JSON as StringContent.

Currently, the CreateOrUpdatePOIAsync() method prints the result of this operation on the console. Later in this chapter, we will see how to use the CreateOrUpdatePOIAsync () method from POIDetailActivity to complete the save operation.

### Consuming the web service to delete the POI

To delete the POI from the server, the client needs to send the information about the POI to be deleted. As all the POI records are uniquely assigned with a unique ID property, we just need to pass the ID of the POI to be deleted. Let's understand the following API specification used to delete the POI:

```
Request Method: DELETE
Resource Endpoint: /com.packet.poiapp/api/poi/delete/{POI_ID}
Response: Success/Failed
```
From the preceding API specification, the request method is `DELETE` and the web service requires the POI ID to be passed to the end of the URL for the server to know which POI to be deleted. As we are not sending any data to the request body, we don't need to specify the `Content-Type` request header. The result of the operation is that this API responds back with a `Success` or `Failure` plain text message.

**Adding the DeletePOIAsync method to POIService**

Consuming the web service to delete a POI is similar to the `GetPoisListAsync()` method implementation. The following steps will help you create a new async method and consume the web service to delete the POI:

1. Create a new async method `DeletePOIAsync()` that accepts an integer value `poiId` that represents a unique `PointOfInterest` object:

   ```csharp
   public async Task<String> DeletePOIAsync (int poiId)
   {
   }
   ```

2. Declare a string constant that represents the URL to web service API for the delete action:

   ```csharp
   private const string DELETE_POI = "http://localhost:8080/com.packt.poiapp/api/poi/delete/{0}";
   ```

   Notice that the `DELETE_POI` string expects the `poiId` parameter to end of the URL. This is the ID of the POI to be deleted from the server.

   Alternatively, you can use the following Apiary test URL:

   ```csharp
   ```

3. Let's now proceed to create an instance of `HttpClient` and to delete the POI from the server. Add the following code snippets to the `DeletePOIAsync()` method:

   ```csharp
   public async Task<String> DeletePOIAsync (int poiId)
   {
   HttpClient httpClient = new HttpClient ();
   String url = String.Format (DELETE_POI, poiId);
   HttpResponseMessage response = await httpClient.DeleteAsync (url);
   if (response != null || response.IsSuccessStatusCode) {
   ```
string content = await response.Content.ReadAsStringAsync();
Console.Out.WriteLine("One record deleted.");
return content;
}
return null;
}

In the preceding code snippet, the `DeleteAsync()` method sends a delete request asynchronously to the specified Uniform Resource Identifier (URI), and the result of this operation is printed on the console.

### Creating SavePOI()

Notice that in the previous step, in the *Adding Save and Delete action* section, we are calling the `SavePOI()` and `DeletePOI()` actions from the `OnOptionsItemSelected()` method, but we haven’t declared them at all. These two methods will deal with consuming web services to create, update, and delete the POI. The following sections will take you through both the `SavePOI()` and `DeletePOI()` actions.

Let's now declare a new method `SavePOI()` in the `POIDetailActivity` class. The `SavePOI()` method could avoid placing a lot of logic in the `OnOptionsItemSelected()` method. This method will validate the user input and initiate the server request to create or update the POI.

We will cover the field validation in an upcoming section and now focus on getting the user data from the screen and initiate a create/update request. The following listing shows what should be present in `SavePOI()`:

```csharp
protected void SavePOI()
{
    _poi.Name = _nameEditText.Text;
    _poi.Description = _descrEditText.Text;
    _poi.Address = _addrEditText.Text;
    _poi.Latitude = tempLatitude;
    _poi.Longitude = tempLongitude;

    CreateOrUpdatePOIAsync(_poi);
}
```

Notice that we are calling a new method, that is, `CreateOrUpdatePOIAAsync()` from `SavePOI()`. We need to add the `CreateOrUpdatePOIAasync()` to `POIDetailActivity`. It is an async method, which will take care of initializing the `POIService` class and initiating the web service request to save the POI.
The following code snippet listing shows what should be present in the `CreateOrUpdatePOIAsync()` method:

```csharp
private async void CreateOrUpdatePOIAsync(PointOfInterest poi){
    POIService service = new POIService ();
    if (!service.isConnected(this)) {
        Toast toast = Toast.MakeText (this, "Not connected to internet. Please check your device network settings.", ToastLength.Short);
        toast.Show ();
        return;
    }

    string response = await service.CreateOrUpdatePOIAsync (_poi);
    if (!string.IsNullOrEmpty (response)) {
        Toast toast = Toast.MakeText (this, String.Format ("{0} saved.", _poi.Name), ToastLength.Short);
        toast.Show();
        Finish ();
    } else {
        Toast toast = Toast.MakeText (this, "Something went Wrong!", ToastLength.Short);
        toast.Show();
    }
}
```

Notice that the preceding method does the following tasks:

- First, it creates an instance of the `POIService` class and confirms the network availability by calling the `isConnected()` method.
- If a device is not connected to the Internet, it shows an appropriate `Toast` message to the user.
- If the network is available, it calls the `CreateOrUpdatePOIAsync()` method, which is defined in the `POIService` class. The web service request can be a long-running blocking operation, and hence, we are using `async await` to make the request asynchronous.
- A toast message is displayed to notify the user about the result of the save/update operation.
- Once the save request is successful, it displays a toast message and calls the activity `Finish()` method. The `Finish()` method causes the `POIDetailActivity` activity to be closed, and the previous activity on the stack will be brought back to the foreground; in our case, `POIListActivity`. 

---

[131]
Creating DeletePOI()

Like SavePOI(), the DeletePOI() method was created to simplify the logic in OnOptionsItemSelected(). Before deleting the POI, we must ask the user to reconfirm by showing a dialog. Later in this chapter, we will show you how to display a confirmation prompt before initiating the deleted request.

We created the CreateOrUpdatePOIAasync() method in POIDetailActivity, let's now add another new method named DeletePOIAasync(). This method performs pretty much the same task as the CreateOrUpdatePOIAasync() method does in POIDetailActivity. It checks for the Internet connection availability, initiates the delete operation asynchronously, and finally, notifies the user with a toast message.

Add the following DeletePOIAasync method to your POIDetailActivity class:

```csharp
public async void DeletePOIAasync()
{
    POIService service = new POIService();
    if (!service.isConnected(this)) {
        Toast toast = Toast.MakeText(this, "Not connected to internet. Please check your device network settings.", ToastLength.Short);
        toast.Show();
        return;
    }

    string response = await service.DeletePOIAasync(_poi.id);
    if (!string.IsNullOrEmpty(response)) {
        Toast toast = Toast.MakeText(this, String.Format("{0} deleted.", _poi.Name), ToastLength.Short);
        toast.Show();
        Finish();
    } else {
        Toast toast = Toast.MakeText(this, "Something went Wrong!", ToastLength.Short);
        toast.Show();
    }
}
```

The following listing shows what should be present in the DeletePOI() method:

```csharp
protected void DeletePOI()
{
    DeletePOIAsync();
}
```
Let's now build and run the app. You should now be able to perform the add, update, and delete operations:

If you're using the Apiary mock feed URL for the save and delete operation, no POI will be deleted or saved. Apiary is just used for testing purpose only. You must deploy the web service code bundle provided in this book for the Save and Delete action to take effect on the server data.

Adding validation
Any nontrivial app will have some level of the validation required. The POIApp app is somewhat trivial, but we have a small set of rules we need to enforce that will facilitate the discussion:

<table>
<thead>
<tr>
<th>Property</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>This cannot be empty or null</td>
</tr>
<tr>
<td>Latitude</td>
<td>This contains a valid decimal number between -90 and 90</td>
</tr>
<tr>
<td>Longitude</td>
<td>This contains a valid decimal number between -180 and 180</td>
</tr>
</tbody>
</table>
Using the EditText.Error property

The EditText widget has a string property named Error, which simplifies the effort of displaying errors to the user, particularly if you want to be able to show all the fields with errors at once. The following screenshot displays the error received for leaving the Name field empty:

![Screenshot of POIDetailActivity with an error message]

To use this facility, simply set the property to an error message and clear the property when no errors exist. The following example demonstrates implementing the rule for the Name property:

```csharp
bool errors = false;
if (String.IsNullOrEmpty (_nameEditText.Text)) {
    _nameEditText.Error = "Name cannot be empty";
    errors = true;
} else
    _nameEditText.Error = null;
```

Notice the local Boolean variable named errors, which is used to keep track of whether any errors have been found. Edits for Latitude and Longitude are a little more involved, as you need to account for converting text to a double value and allow a null value to be specified.
The following code demonstrates one of the approaches to implement the edits:

double? tempLatitude = null;
if (!String.IsNullOrEmpty(_latEditText.Text)) {
    try {
        tempLatitude = Double.Parse(_latEditText.Text);
        if ((tempLatitude > 90) | (tempLatitude < -90)) {
            _latEditText.Error = "Latitude must be a decimal value
between -90 and 90";
            errors = true;
        }
        else
            _latEditText.Error = null;
    }
    catch
    {
        _latEditText.Error = "Latitude must be valid decimal number";
        errors = true;
    }
}

Implement the rules identified at the start of this section in the SavePOI() method using the EditText.Error property.

You can update and save the POI properties only if all of the edits are passed. The following listing shows one of the ways to structure the logic:

    if (errors) {
        return;
    }

    _poi.Name = _nameEditText.Text;
    _poi.Description = _descrEditText.Text;
    _poi.Address = _addrEditText.Text;
    _poi.Latitude = tempLatitude;
    _poi.Longitude = tempLongitude;
    CreateOrUpdatePOIAsync();

Run POIApp and confirm that the validations are working correctly.
Adding a Details View

Adding a delete confirmation prompt

It's a best practice for apps to provide a confirmation before performing any type of destructive update, particularly if it cannot be undone. As such we need to provide a confirmation for the Delete action. Fortunately, Android makes this relatively easy with the AlertDialog and AlertDialog.Builder classes.

The AlertDialog class allows you to display a modal confirmation dialog. The AlertDialog.Builder class is an embedded class that helps you construct an instance of an AlertDialog method; you can think of it as a factory class. The steps are as follows:

2. Set various properties on the builder instance, such as the message, the button text, the calling of event handlers when a button is clicked, and so on.
3. Call Show() on the instance of AlertDialog.Builder to create and display an instance of AlertDialog.
   In our case, we want an AlertDialog class with a simple message and an OK and Cancel button. When we click on Cancel, we simply need to close the dialog and do nothing. When the user selects OK, we need to initiate the delete POI operation.
4. Create an event handler that will be called when you click on the OK button. This method will now call DeletePOIAasync() to perform the delete operation. The following listing depicts these changes:
   protected void ConfirmDelete(object sender, EventArgs e)
   {
     DeletePOIAasync();
   }
5. Add the logic that constructs the AlertDialog class into the existing DeletePOI() method. The following listing depicts this logic:
   protected void DeletePOI()
   {
     AlertDialog.Builder alertConfirm = new AlertDialog.Builder(this);
     alertConfirmSetTitle("Confirm delete");
     alertConfirm.SetCancelable(false);
     alertConfirm.SetPositiveButton("OK", ConfirmDelete);
     alertConfirm.SetNegativeButton("Cancel", delegate {});
     alertConfirm.SetMessage(String.Format("Are you sure you want to delete {0}?", _poi.Name));
     alertConfirm.Show();
   }
The `SetPositiveButton()` and `SetNegativeButton()` methods allow button captions and event handlers to be specified. In the case of the `negative` button **Cancel**, we provide an empty event handler because there is nothing to do; Android will take care of closing the dialog. **AlertDialog** also provides a **neutral** button.

On devices prior to Honeycomb, the button order (left to right) was **positive - neutral - negative**. On newer devices, using the Holo theme, the button order (left to right) is **negative - neutral - positive**.

Run **POIApp** and verify that the delete confirmation is working correctly. The following screenshot shows the delete confirmation dialog when the user clicks on the delete action:

![Screenshot](image)

**Refreshing POIListActivity**

Actions we take on **POIDetailActivity**, such as **Save** and **Delete**, have an effect on the data displayed in **POIListActivity**. We need to ensure that **ListView** in **POIListActivity** is refreshed and displays the updated POI list when it becomes active again. To achieve this, we have to depend on the activity life cycle callback methods.
Adding a Details View

As you may recall from Chapter 1, The Anatomy of an Android App, when an activity is moved to the background due to the start of a new activity, the OnPause() method is called. This would have happened for POIListActivity when POIDetailActivity was started. Once POIDetailActivity is completed by either calling the Finish() method or by pressing the device back button, the POIListActivity will be brought back to the foreground and the OnResume() method will be called.

Let's remove the call to DownloadPoisListAsync() in the OnCreate() method and add the following snippets to refresh POIListActivity:

```csharp
protected override void OnResume (){
    base.OnResume ();
    DownloadPoisListAsync ();
}
```

We have covered a lot of ground in this chapter. We have completed the POI details activity to perform add, update, or delete POIs. If you have any unintended deviations, you can refer to the code bundle.

Summary

In this chapter, we covered a lot of ground creating complex layouts using different layout managers such as LinearLayout, TableLayout, and ScrollView and using EditText to validate the form data.

Now a new activity is added to POIApp that displays the details of the POI and allows the user to perform add, update, or delete POI operations. The StartActivity method is used in combination with Intent for passing the data bundle between activities.

We also extended the POIService class by adding additional methods to perform the web service POST and DELETE operations.

The next chapter will guide you to deal with the application behavior when a device orientation changes.
This chapter will walk you through the process of dealing with the Android application behavior on device configuration changes. In the course of this chapter, we will cover comprehensive details of the theoretical concepts, but we will not make any significant changes to the POIApp. The following topics will be covered in this chapter:

- The Android behavior on configuration changes
- Locking the Android application orientation
- Saving the activity state for configuration changes
- Adding alternative resources
- Manually handling the orientation behavior

**The Android behavior on configuration changes**

All modern smartphones and tablets switch between portrait and landscape modes in response to the way a user rotates the device. The Android application is expected to respond to the configuration changes and displays appropriate layouts for the current device configuration. The Android device configuration can be changed at runtime in many forms, such as device orientation changes, device language changes, device font updates, device connected to external displays, device connected to docks, and so on. In all of the earlier cases, Android restarts the running activity, loads alternative resources if available, to properly load your application for the given configurations. The activity goes through a series of life cycle methods, such as `onDestroy()`, followed by `onCreate()` to deal with the activity restart behavior.
For example, if you are running your application on Nexus 5 device with 1080 x 1920 resolution in the portrait mode, and while the orientation changes to landscape, the application has to respond appropriately to fit the layout and its child views for the 1920 x 1080 dimension. You may ask why not just adjust the view in the layout to fit to the appropriate size? Why should the activity be restarted? Well, Android does this to keep things simple. However, internally, it does many things and provides some of the advance features to make the configuration changes smoother.

If the device configuration changes, Android destroys and restarts only the activity in the foreground, but the application instance still remains unchanged. When the activity restarts, the appropriate layout for that configuration is loaded. For such events, it is important that your application must restore the state of your activity and the user entered data on the UI. This can be possible using the OnSaveInstanceState() and OnRestoreInstanceState() callback methods.

This chapter tackles some of the key points to build smooth, responsive, and orientation aware applications.

**Locking the Android application orientation**

According to the Google design guidelines, the Android application must respond to the device orientation and display an appropriate layout for the given orientation. However, some kind of applications, such as games, video players, and so on, are intended to restrict the orientation to landscape or portrait only. For native Android applications, this can be achieved using the android:screenOrientation property to the <activity> declaration in the AndroidManifest.xml descriptor file. Xamarin makes this even simplified and recommends that you do not to edit the AndroidManifest.xml file manually at any point of time, instead it allows you to set all the application configuration metadata using [Activity] attribute custom properties to the activity declaration.

As we recall from Chapter 3, Creating the Points Of Interest App, we are already using some of the properties of the [Activity] attribute, such as Label, MainLauncher, and Icon. Now to lock the activity orientation behavior, we can use the ScreenOrientation property on the activity class declaration. The ScreenOrientation property needs to be specified statically to each activity that requires control over the orientation lock. It cannot be controlled globally for the whole application scope.
The `ScreenOrientation` attribute expects any one of the possible constants defined in the `Android.Content.PM.ScreenOrientation` enumeration. The `ScreenOrientation` enum defines all the set of device configuration constants such as `FullSensor`, `FullUser`, `Landscape`, `Locked`, `Nosensor`, `Portrait`, and so on.


Note that in this book, we will build the `POIApp` to respond to the device orientation and other different configuration’s change events. The following code block is used for demonstration purpose only and will not be continued along with the `POIApp` example code. You can add the following attributes to lock the `POIListActivity` orientation to landscape only:

```csharp
namespace POIApp
{
    [Activity (Label = "POI List", ScreenOrientation =
    ScreenOrientation.Landscape)]
    public class POIListActivity : Activity
    {
        ....
        .......
        }
}
```

Make the preceding changes to `POIListActivity` and run the app. You will notice that the POI list screen overrides the device orientation lock settings and it will always be shown in the landscape mode.

**Dynamically requesting the orientation**

You can also change the desired activity orientation dynamically by setting the `RequestedOrientation` property to your activity. This allows further flexibility to dynamically change the orientation anytime as required. Add the following code snippet anywhere in your activity to restrict the currently running activity orientation to landscape only:

```csharp
RequestedOrientation = ScreenOrientation.Landscape;  
```

The changes made to the `RequestedOrientation` property has an impact on the activity in the foreground and a request for the specified orientation is made immediately by causing the current activity to restart. You can use the same property to obtain the current configuration that is in effect for the current activity.
Saving the activity state for configuration changes

Runtime configuration changes in the Android device cause the activity in the foreground to go through a restart process. This behavior is intended to simplify the process for the activity to reinitialize with alternative resources for the given configuration. As the activity goes through the recreation process, you might lose the user entered data or the current state of the activity. For any such course of events, all the Android applications must retain the activity state quickly and cheaply to avoid undesirable user experience. The following section will help you dive deeper to understand the process to retain the activity state.

With the configuration changes, the activity goes through a series of life cycle methods, such as `OnDestroy()` followed by `OnCreate()`. On its way, it calls `OnSaveInstanceState()` before it destroys the activity so that you can save the activity state data. The application state can be retained during the `onCreate()` or `OnRestoreInstanceState()` callback methods. To understand the whole process, we must recall the activity life cycle methods as described in Chapter 1, *The Anatomy of an Android App*. The following figure depicts the activity life cycle and the `OnSaveInstanceState` and `OnRestoreInstanceState()` method’s call flow with their respective configuration changes:
A change in the orientation calls the `OnPause()`, `OnSaveInstanceState()`, `OnStop()`, and `OnDestroy()` methods followed by `OnCreate()`, `OnStart()`, `OnRestoreInstanceState()`, and `OnResume()`. Remember that this sequence is not always true. For example, when a user presses the device back button or calls the `Finish()` method, you do not need to save the activity state; hence, it just calls `OnPause()`, `OnStop()`, and `OnDestroy()`, but the `OnSaveInstanceState()` method is never called.

The default implementation of `OnSaveInstanceState()` and `OnRestoreInstanceState()` takes care of saving and retaining the data on all the Android view input widgets (for example, `EditText`, `CheckBox`, `RadioButton`, and so on) that are attached to the layout with an id property. This means that the Android SDK implements the `OnSaveInstanceState()` and the `OnRestoreInstanceState()` methods on each of the view controls.

Based on two factors, Android decides whether the state of a view attached to a view group will be retained in the orientation change. One factor is the id property and the other is based on the value set for the `view.SaveEnabled` property. The `SaveEnabled` property checks whether the `onSaveInstanceState()` method will be called on that view. The default value for `SaveEnabled` is true.

Apart from the `SaveEnabled` property, Android provides yet another interesting method that allows you to control the save state behavior for the entire hierarchy under a view. For example, if you have five different views under a `LinearLayout`, and you wish not to save state for all those five views, including `LinearLayout`, you can just achieve this by setting the `SaveFromParentEnabled` property of `LinearLayout` to false.

For applications using custom or compound views, you must handle the state restoration manually. You need to override the `OnSaveInstanceState()` and `OnRestoreInstanceState()` methods in your custom view implementation to store the state of your view.

**Manually saving the activity state**

If you need to manually save the activity state, you must override the following methods in your activity and write your own logic to save and restore the activity state in data bundles. The `OnSaveInstanceState()` method supplies an instance of `Bundle` in which we can dump the data, and the `OnRestoreInstanceState()` method returns the previously saved `Bundle`. 

---

[143]
Bundle is a special kind of container that provides the key/value maps of heterogeneous values. Bundle is generally used for parsing data from one activity to another or while saving and retrieving the view state. The Bundle class provides a set of overloaded PutXXX() and GetXXX() methods to store and retrieve the values.

The following code snippet depicts the OnSaveInstanceState() and OnRestoreInstanceState() methods' prototype:

```csharp
protected override void OnSaveInstanceState (Bundle outState)
{
    base.OnSaveInstanceState (outState);
    // Place your logic to save activity state
}

protected override void OnRestoreInstanceState (Bundle savedInstanceState)
{
    base.OnRestoreInstanceState (savedInstanceState);
    // Place your logic to restore activity state
}
```

Retaining the POI list scroll position

So far, we had quite a lot of ground on the activity life cycle behavior on configuration changes. Let's now implement it to POIApp. When the POI list grows, it allows the user to scroll vertically to see all of the list items. Let's say, the user has scrolled till the 10th element in the list, and in the meantime, the device orientation changed. Due to the device configuration change request, the activity restarted and causes the list to appear from the top. It is a good idea to retain the list scroll position for better user experience.

Let's perform the following steps to retain the POI list scroll position when the device configuration changes:

1. Get the index of the first visible list item and save it in Bundle in the OnSaveInstanceState() method:

```csharp
protected override void OnSaveInstanceState (Bundle outState)
{
    base.OnSaveInstanceState (outState);
    int currentPosition = poiListView.FirstVisiblePosition;
    outState.PutInt ("scroll_position", currentPosition);
}
```
In the preceding code snippet, the string `scroll_position` is used as the key to save the current scroll position of `ListView` on orientation change. The same key should be used while retrieving the data from the `OnRestoreInstanceState()` callback.

2. Restore the saved list scrolling position in `OnRestoreInstanceState()`. Note that we have to use the same key that was used to save the current scroll position:

   ```csharp
   int scrollPosition;
   protected override void OnRestoreInstanceState(Bundle savedInstanceState)
   {
       base.OnRestoreInstanceState(savedInstanceState);
       scrollPosition = savedInstanceState.GetInt("scroll_position");
   }
   ``

3. Add the following code snippet to the `DownloadPoisListAsync()` method in the `POIListActivity` class and queue the message for scrolling the POI list to the last saved scroll position:

   ```csharp
   poiListView.Post(() => {
       poiListView.SetSelection(scrollPosition);
   });
   ```

Now run the application and test the POI application by changing the orientation; notice that the scroll positions are retained.

### Building the orientation-aware layout

Android allows you to add multiple competing versions of resources to make an application compatible with different device configurations. The Android system comes handy when it comes to picking up the right resources for a given configuration. You don't need to write any code to find what is the current configuration and choose the appropriate resources. All you need to do is to add multiple versions of resources, such as `string.xml`, `.png` drawable images, or `layout XML` files, so that the best version of resource can be picked up under different circumstances.
Imagine that your app was primarily targeted to the U.S. market, but now we are looking forward to targeting Russia with the Russian language support. In this case, multiple versions of `string.xml` can be added to support the Russian language. Or imagine that a user is running the app on a tablet in the portrait mode, and now the device rotates to the landscape orientation. The screen width gets doubled and provides the opportunity to display more information. In such cases, you will need to add multiple versions of activity layouts for different orientations. Let's hold our breath till the next chapter to learn more about how to build the orientation aware layout for Android tablets.

All the resources added to the application will be added to various subdirectories under the `Resources` directory. The default resource directory structure is shown as follows:

Any resource that is added as a default resource will be used when there are no alternative best match resources available for the given configuration. Alternative resources are designed for a specific configuration.

To add alternative resources, you just need to use the specific configuration qualifier. The configuration qualifiers you can use for device screen sizes are small, normal, large, and xlarge. For example, layouts for an extra-large screen, such as a tablet layout, should go in the `layout-xlarge` directory. Since Android 3.2, Android recommends that you use the `sw<N>dp` configuration qualifier to define an extra-large layout for the tablet. For example, if your multipane tablet layout requires at least 600dp of screen width, you should place it in the `layout-sw600dp` directory.

To provide the translated version of the `strings.xml` file for Russian users, you have to place the `string.xml` file in the `Resources/values-ru/` directory.
The following table depicts the list of configuration qualifiers that allow you to provide special resources for different screen configurations:

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Qualifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Screen size</td>
<td>small, normal, large, and xlarge</td>
<td>Resources for small, normal, large, and extra-large screen sizes.</td>
</tr>
<tr>
<td>1 dpi</td>
<td></td>
<td>Resources for low-density (1 dpi) screens (~120 dpi). You can now ignore this resource set, as no new Android devices with ldpi density are out in the market.</td>
</tr>
<tr>
<td>mdpi</td>
<td></td>
<td>Resources for medium-density (mdpi) screens (~160 dpi). (This is the baseline density.)</td>
</tr>
<tr>
<td>hdpi</td>
<td></td>
<td>Resources for high-density (hdpi) screens (~240 dpi).</td>
</tr>
<tr>
<td>xhdpi</td>
<td></td>
<td>Resources for extra-high-density (xhdpi) screens (~320 dpi).</td>
</tr>
<tr>
<td>xxhdpi</td>
<td></td>
<td>Resources for extra-extra-high-density (xxhdpi) screens (~480 dpi).</td>
</tr>
<tr>
<td>xxxhdpi</td>
<td></td>
<td>Resources for extra-extra-extra-high-density (xxxhdpi) use (~640 dpi). Use this for the launcher icon only; see the earlier note.</td>
</tr>
<tr>
<td>nodpi</td>
<td></td>
<td>Resources for all densities. These are density-independent resources. The system does not scale resources tagged with this qualifier.</td>
</tr>
<tr>
<td>Device orientation</td>
<td>land</td>
<td>Resources for screens in the landscape orientation.</td>
</tr>
<tr>
<td>port</td>
<td></td>
<td>Resources for screens in the portrait orientation.</td>
</tr>
<tr>
<td>Aspect ratio</td>
<td>long</td>
<td>Resources for screens that have a significantly taller or wider aspect ratio (when in the portrait or landscape orientation, respectively) than the baseline screen configuration.</td>
</tr>
<tr>
<td></td>
<td>notlong</td>
<td>Resources for use screens that have an aspect ratio that is similar to the baseline screen configuration.</td>
</tr>
</tbody>
</table>

In the following Chapter 7, Designing for Multiple Screen Sizes, we will make use of the alternative layout resource to make the application compatible for Android tablets.
Manually handling the orientation behavior

As described, the Android system automatically takes care of updating appropriate resources when the configuration changes. However, at times due to performance reasons, you might like to restrict the activity restart and write your own logic to update appropriate resources for the given configuration. But remember that this is not recommended by the Google guidelines; if you're implementing this for your application, do it at your own risk.

The following steps will guide you to manually handle the required configuration for your app:

1. Add the `ConfigurationChanges` attribute to the activity declaration. This allows you to declare all possible configurations values defined in `Android.Content.PM.ConfigChanges` that you want to handle yourself for your app. At runtime, this adds the `android:configChanges` attribute to your activity declaration in the `AndroidManifest.xml` file:

   
   ```
   public class POIListActivity : Activity
   {
   ---
   }
   ```

   In the preceding code syntax, we stated that we will be manually handling the configuration for the device orientation and sliding keyboard state changes. Note that the preceding declaration bypasses the whole activity destruction process and simply returns a callback to let you know about the changes.

2. Override `OnConfigurationChanged()` in your activity. This method will be called when one of the configuration changes, as listed in the `ConfigurationChanges` attribute declaration:

   ```
   {
   base.OnConfigurationChanged (newConfig);
   //update UI to reflect the orientation change
   ```
Here, in the `OnConfigurationChanged()` method, you need to update the UI so that it reflects the orientation change. For our `POIApp`, we allow the system to handle the configuration itself. Note that the all the code snippets used in this chapter are for demonstration purpose only, and we will not continue with these changes to build `POIApp` for rest of the chapters in this book.

**Summary**

In this chapter, we covered a lot of ground on the behavior of an activity when the device configuration is changed, forcing the activity to display a specified orientation and to save and restore the activity state. The next chapter will guide you to deal with designing applications for various device form factors such as Android tablets.
7

Designing for Multiple Screen Sizes

In early 2011, Android Honeycomb 3.0 (API level 11) was released specifically to support larger screen-sized tablets. Since then, the Android ecosystem has exploded and all subsequent Android releases have been targeted to offer more screen sizes, including smart phones, tablets, and Google TV.

This chapter will walk you through dealing with multiple screen sizes and making your application compatible to work with smart phones and tablets. In the course of this chapter, we will extend the POIApp we have been developing and optimize it for Android tablets. The following topics will be covered in this chapter:

- Introduction to the Android tablets ecosystem
- Creating and managing fragments
- Making POIApp compatible with Android tablets
- Dealing with ListFragment to display the POI list
- Creating the multi-pane layout for tablets
- Adding alternative layouts for Android tablets
- Using DialogFragment to display dialogs
- Working with fragments for older Android devices
- Retaining the fragment state on device configuration changes
An introduction to Android tablets

The first ever Android tablet was released in 2011 with Android Version 3.0. However, all subsequent Android releases were targeted to support a range of screen sizes, including small, medium, and large ones. Due to its open nature, the Android ecosystem has a variety of device manufacturers, and hence, there are devices with different screen sizes and densities.

Applications built for Android smart phones can run on tablets without making any changes. However, if the UI is not optimized, it will give you a disagreeable user experience. Developers must take the extra space as an opportunity and use it efficiently to design a UI that can display more information on larger screens. For example, the Gmail application in the Android phone will have two activities for displaying the recent e-mail list and details of a selected e-mail. However, the same app in the tablet displays the e-mail list and the details on the same page, using the multi-pane split view layout.

The key things to be considered while designing user interfaces for Android applications are as follows:

- **Screen size**: This is the actual physical screen size of the device; generally, grouped into small, normal, large, and extra large. Although there is no official confirmation on the minimum and maximum screen sizes, they usually range from 2.55 to 10.1 inches.

- **Screen density**: This is the number of pixels present in the physical area of the screen, which is represented as dpi (dots per inch). High-density screens have more pixels than low-density screens.

- **Resolution**: This is the number of pixels present on the device screen. It is usually defined as \( \text{width} \times \text{height} \). For example, the resolution for a Nexus 5 device is 1080 \( \times \) 1920.

- **Screen orientation**: A device's orientation could be either in landscape or portrait mode.

Creating an Android tablet emulator

So far, we have built the POIApp to display the list of POIs fetched from the server and tested it on the Android smart phone. Before we begin with extending the POIApp to support Android tablets, let's create a new emulator instance for the Android tablet and run the existing POIApp built so far.
In order to create an emulator for the tablet, perform the following steps:

1. From the main menu bar, navigate to Tools and open Google Emulator Manager. This will open the AVD manager window.
2. To create a new emulator, click on the Create button, in the right-hand side pane.
3. Set the AVD Name field to Nexus10, select Nexus 10 from the Device drop-down menu, set the Target field to Android 4.4.x, if needed, uncheck the Hardware keyboard present option, and click on OK.
4. Note that the Target setting specifies the version of the Android platform and the API level that will be used for the emulator.
5. Provide the configurations such as the AVD name, Device, Skin, Target, Memory Options, and so on, as shown in the following screenshot:
If you're using third-party emulator solutions, such as Xamarin Android Player or Genymotion, you can download either the Nexus 7 or Nexus 10 simulator for testing purposes in the tablet configuration.

Now start the Nexus 10 emulator created in the previous steps and run the existing POIAApp built so far. The following screenshot depicts the UI of the POIAApp on the tablet that has not been optimized for larger screens:

With the additional screen real estate, the UI elements can be organized in various locations to make the app look nicer. Currently, the POIAApp is using two activities for displaying the POI list and POI details. For tablets, we can use a multi-pane split layout to display both the POI list and details in the same activity by sharing the same screen space. Android activities are meant to be for single purposes and they don't allow embedding one activity inside another. The Android fragment can be used to solve this problem. A fragment is a reusable user interface component used to build dynamic and modular user interfaces for different screen sizes. An activity can contain one fragment or multiple fragments.
An introduction to fragments

Fragments are reusable pieces of a user interface component that provide flexibility to the application design. As you have already learnt from Chapter 1, The Anatomy of an Android App, fragments are reusable mini-activities, such as UI components, that can manage their own life cycles. Fragments are always intended to work without depending on another activity or fragment. Like an Activity class, a fragment class needs to extend from the Fragment class. To make things even easier, Android provides some of the additional specialized fragment subclasses such as ListFragment, DialogFragment, and PreferenceFragment.

The following table shows the list of fragment subclasses and their purposes. You may also extend any of the following fragment subclasses to create your own fragment.

<table>
<thead>
<tr>
<th>Fragment</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListFragment</td>
<td>This displays a list of data items from different sources, such as an array, a cursor, and so on.</td>
</tr>
<tr>
<td>DialogFragment</td>
<td>This displays a fragment as a floating dialog window.</td>
</tr>
<tr>
<td>PreferenceFragment</td>
<td>This displays the hierarchical app preferences list. The preferences are saved when any changes are made by a user. The Nexus device settings screen design follows the PreferenceFragment pattern.</td>
</tr>
</tbody>
</table>

Creating a new fragment

A fragment can be seen as a modular component of an activity, which maintains its own life cycle, that handles user events and can be added to or removed from a running activity. The process of creating a fragment is very similar to creating an activity.

Perform the following steps to create and add a fragment in Android:

1. The first step toward creating a fragment is to define its layout. Like the activity layout, you can either use the drag-and-drop interface builder available in Xamarin Studio, or you may use the XML code editor to create the layouts.
Designing for Multiple Screen Sizes

2. Create a new class that directly extends the Fragment class or its subclasses. The following code snippet extends the Fragment base class:

   public class MyFragment : Fragment
   {
       -----
       ----- 
   }

3. Now implement the necessary fragment methods. To begin with, let's override the OnCreateView() method. This method adds the fragment to the view hierarchy and returns a View instance that represents the fragment.

   The fragments UI can be created in two ways: by declaring the XML layout and inflating the fragment layout inside OnCreateView(), or by creating all the interfaces dynamically. For simplicity, the XML declarative approach is always preferred:

   public override View OnCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)
   {
       // Inflate the layout for your fragment
       View view = inflater.Inflate(Resource.Layout.MyFragmentLayout, container, false);

       // Initialize other view controls
       return view;
   }

   Optionally, you may override other fragment life cycles such as OnPause(), OnResume(), and so on, to control other life cycle event callbacks.

4. Finally, let's add the fragments to the activity. This can be done in two ways: either by declaratively adding the activity layout, or by adding it programmatically.

   Use the following code snippet to add the fragment by declaring the <fragment> tag in the activity layout:

   <fragment class="MyApp.MyFragment"
           android:id="@+id/myFragment"
           android:layout_width="match_parent"
           android:layout_height="match_parent"/>

   [156]
Chapter 7

To add the fragment dynamically, you need to first create a fragment placeholder
container layout and assign an ID to it:
<FrameLayout
android:id="@+id/myFragmentContainer"
android:layout_width="match_parent"
android:layout_height="match_parent" />

Now that you have specified the ViewGroup in which the fragment will be added
dynamically, you can use the FragmentTransaction instance to add, remove,
and replace the transactions. Use the following snippets to get the instance of
FragmentTransaction:
MyFragment myFragment = new MyFragment();
FragmentTransaction ft = FragmentManager.BeginTransaction();

You can then use the Add(), Remove(), and Replace() methods by passing the
instance of the fragment and the ID of the View to which the fragment will be added.
Once the fragment transactions are completed, you must call the Commit() method
for the changes to take effect:
ft.Add(Resource.Id.myFragmentContainer, myFragment);
ft.Commit();

So far, you have a fair idea of how to work with fragments; let's now proceed to
make the POIApp work with fragments and reuse the fragments to optimize
the tablet's layouts.

Making POIApp compatible with Android
tablets

We have already covered a lot of ground on the fragment basics and life cycle in
Chapter 1, The Anatomy of an Android App. I recommend that you read the concepts if
you haven't gone through them already. Let's now go through the steps to create and
manage fragments and build the POIApp to support the multi-pane tablet layout.

[ 157 ]


Currently, the POIApp is using two activities: POIListActivity for displaying the list of POIs, and POIDetailsActivity for displaying the details. Now we will create two new fragments: POIListFragment and POIDetailFragment, which will be reused for both the smart phone and multi-pane tablet layout:

Notice the following points in the preceding screenshot, which will be implemented in the rest of the sections in this chapter:

1. The phone layout contains two activities; POIListActivity and POIDetailsActivity, that host POIListFragment and POIDetailFragment respectively. Here, the activities will work like placeholders to hold fragments. Most of the application logic is to be moved from an activity to fragments.

2. The POIListFragment will contain the logic to initialize the list view, download the data using methods declared in the POIService class, and prepare the UI. Currently, the same logic is placed on POIListActivity.

3. The POIDetailFragment will contain the logic to display the details of the selected POI and the ability to create, update, and delete the POI.

4. The tablet real estate is comparatively larger, and hence its layout can hold both POIListFragment and POIDetailFragment on a single activity. This allows us to reuse the fragments and enables different user experiences on different screen sizes.

The following sections in this chapter will guide you through how to use fragments to build the POIApp using the multi-pane split view layout for Android tablets.
Using fragments for displaying POIDetails

In order to make things simple, we will move step by step. First, let’s create a new fragment to display the details of the POI, and this will enable you to edit, update, and delete the POI. Currently, the POIDetailActivity activity contains the same logic. At this point, let’s keep it simple and not complicate things by thinking about the multi-pane tablet layout.

Creating the POIDetailsFragment layout

Let’s begin with creating a new layout file for the POI details fragment and name it POIDetailFragment.axml:

1. Select the Resources/Layout folder in the Solution pad.
2. Right-click on Add and select New File.
3. In the New File dialog box, click on Android and select Layout, enter POIDetailFragment in the Name field, and select New.
4. Note that we will not make any changes to the POI details screen layout. So, let’s just copy the layout XML source code from POIDetail.axml and paste it into the newly created POIDetailFragment.axml file.

Creating the POIDetailFragment

Now that we have the POI details fragment layout ready, we need a corresponding fragment. To create POIDetailFragment, perform the following steps:

1. Select the POIApp project in the Solution pad, right-click on it, and navigate to Add | New File.
2. From the New File dialog box, click on Android and select Fragment, enter POIDetailFragment in the Name field, and click on New.

The POIDetailFragment created in the previous steps will contain the logic to display the details of the POI and enables various actions, such as create, update, and delete the POI. Currently, the business logic is in POIDetailsActivity.
3. Declare the following set of private variables for each of the input widgets we created in the layout. Add the following listing to your POIDetailFragment class:

```java
PointOfInterest _poi;

EditText _nameEditText;
EditText _descrEditText;
EditText _addrEditText;
EditText _latEditText;
EditText _longEditText;
```

4. Override the `OnAttach()` method and save the reference of the `activity` instance:

```java
private Activity activity;

public override void OnAttach (Activity activity)
{
    base.OnAttach (activity);
    this.activity = activity;
}
```

5. Override the `OnCreateView()` method, inflate the fragment layout, and bind each variable to its corresponding user interface widget by calling the `FindViewById<T>` method and the `UpdateUI()` method:

```java
public override View OnCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)
{
    View view = inflater.Inflate(Resource.Layout.POIDetailFragment, container, false);

    _nameEditText = view.FindViewById<EditText>(Resource.Id.nameEditText);
    _descrEditText = view.FindViewById<EditText> (Resource.Id.descrEditText);
    _addrEditText = view.FindViewById<EditText> (Resource.Id.addrEditText);
    _latEditText = view.FindViewById<EditText> (Resource.Id.latEditText);
    _longEditText = view.FindViewById<EditText> (Resource.Id.longEditText);

    UpdateUI();
    return view;
}
6. Copy and paste the `UpdateUI()`, `SavePOI()`, `DeletePOI()`, and `ConfirmDelete()` methods' code from the `POIDetailsActivity` class to the `POIDetailFragment` class. You will get a compilation error wherever this keyword is used. Replace this with the activity instance.

7. Now copy both the `CreateOrUpdatePOIAsync()` and `DeletePOIAsync()` methods from `POIDetailsActivity` to the `POIDetailFragment` class. Here, you will again notice a compilation error near the `Finish()` method and the `Toast` initialization. For the `Toast` instantiation, replace this with the activity instance and replace `Finish()` with `activity.Finish()`.

8. Now let's override the `OnCreate()` method. Here, we will retrieve the details of the POI sent to `POIDetailFragment`. Each fragment has an `Arguments` property that contains the data bundles. As with activities, we can use a series of `GetXXX()` methods, where `XXX` represents the type of the value of a name/value pair:

```csharp
public override void OnCreate(Bundle savedInstanceState)
{
    base.OnCreate(savedInstanceState);
    if (Arguments!=null && Arguments.ContainsKey("poi")) {
        string poiJson = Arguments.GetString("poi");
        _poi = JsonConvert.DeserializeObject<PointOfInterest>(poiJson);
    } else {
        _poi = new PointOfInterest();
    }
}
```

### Working with the save and delete actions

The `POIDetailFragment` displays the details of the selected POI to the user and also retains the ability to save a new or delete an existing POI. The save and delete actions are to be added to the action bar in `POIDetailFragment`.

The following steps will guide you through what is needed to add the actions in the `POIDetailFragment`:

1. To make the action bar button work on the fragments, you need to first enable this option by calling the `SetHasOptionsMenu()` method and passing the `bool` value `true`. The `bool` value represents whether or not the fragment wants to contribute in order to populate the menu items. Add the following block to `OnCreateView()` before the `return` statement:

```csharp
SetHasOptionsMenu (true);
```
2. In Chapter 5, Adding a Details View, we have already added the menu declaration to the POIDetailMenu.xml file. The POIDetailFragment will continue to use the same menu XML declaration.

3. Override the fragment version of the OnCreateOptionsMenu() and OnOptionsItemSelected() methods in POIDetailFragment, and add the following code blocks:

```csharp
public override void OnCreateOptionsMenu(IMenu menu, MenuInflater inflater)
{
    inflater.Inflate(Resource.Menu.POIDetailMenu, menu);
    base.OnCreateOptionsMenu(menu, inflater);
}

public override bool OnOptionsItemSelected(IMenuItem item)
{
    switch (item.ItemId)
    {
    case Resource.Id.actionSave:
        SavePOI();
        return true;
    case Resource.Id.actionDelete:
        DeletePOI();
        return true;
    default:
        return base.OnOptionsItemSelected(item);
    }
}
```

Notice that in the previous code blocks, we are not making any changes to the logic while populating the save and delete actions. It is just the copy of the same code, which was used in the POIDetailActivity class.

4. Override the OnPrepareOptionsMenu() method in POIDetailFragment to disable the delete action when a new POI is being created.

The following listing shows how to disable the delete action when a new POI is being created:

```csharp
public override void OnPrepareOptionsMenu(IMenu menu)
{
    base.OnPrepareOptionsMenu(menu);
    if (_poi.Id <= 0) {
```
IMenuItem item = menu.FindItem (Resource.Id.actionDelete);
item.SetEnabled (false);
item.SetVisible(false);
}
}

Adding the POIDetailFragment to POIDetailActivity

Now that we are ready with the POIDetailFragment, we can add it to POIDetailActivity to make it fully functional. As discussed earlier in this chapter, a fragment can be added to an activity in two ways: using a declarative approach, or by dynamically inflating the layout. For POIDetailActivity, we will add the fragment dynamically.

The following sections will take you through the code refactoring process of adding POIDetailFragment to the POIDetailActivity activity.

Modifying the POI details activity layout

Now that we have placed the entire details view layout in the POIDetailFragment layout, the POIDetail activity layout will have a simple container that can host the fragment. For this purpose, we can use FrameLayout.

FrameLayout is a special kind of a layout manager, generally used to display a single item. When multiple views are added, they are placed in a stack with the most recently added child on the top.

Let's make the following changes to the POI details activity layout and add a dummy view that acts as a placeholder to hold the fragment:

1. Open the Resources/POIDetail.axml file, and in the Document Outline view, click on ScrollView and then click on Delete. Notice that the layout will be empty.
2. Search for FrameLayout in the Toolbox and drag it to the simulator window to add it as a root element.
3. Click on FrameLayout in the Document Outline view and then click on the Properties window. Set the view ID to poiDetailLayout.
4. Click on the **Source** tab in the layout editor and notice that the following code is generated:

```xml
<FrameLayout xmlns:p1="http://schemas.android.com/apk/res/android"
    p1:minWidth="25px"
    p1:minHeight="25px"
    p1:layout_width="match_parent"
    p1:layout_height="match_parent"
    p1:id="@+id/poiDetailLayout " />
```

---

**Refactoring POIDetailActivity to add POIDetailFragment**

Currently, the POIDetailFragment contains the logic we wrote earlier to perform the add, update, or delete operations in POIDetailActivity. Now let's refactor POIDetailActivity to thin out the logic and just add POIDetailFragment to it:

1. Open the POIDetailActivity class from the Solution Explorer.
2. Select all of the code and delete everything except the basic activity template.
   For the sake of simplicity, you may replace the whole POIDetailActivity class with the following code:

```
using Android.App;
namespace POIApp
{
    [Activity (Label = "POIDetailActivity")]
    public class POIDetailActivity : Activity
    {
        PointOfInterest _poi;
        protected override void OnCreate (Bundle bundle)
        {
            base.OnCreate (bundle);
            SetContentView (Resource.Layout.POIDetail);
        }
    }
}
```
3. Now it is time to add `POIDetailFragment` to `Activity`. Let's first initialize the `POIDetailFragment` and pass the POI details data received from `POIListActivity`.

To display the details of the selected POI, the `POIDetailFragment` expects the POI data to be passed to it. Like any other activity, data bundles can be passed to a fragment using its `Arguments` property. The `Arguments` property has its own set of the `PutXX()` and `GetXX()` methods (where `XX` represents the data types, such as string, double, and so on) that are used to send and retrieve data from fragments.

Add the following code snippet to the `OnCreate()` callback right after the `SetContentView()` method:

```java
var detailFragment = new POIDetailFragment();
detailFragment.Arguments = new Bundle();
if (Intent.HasExtra("poi")) {
    string poiJson = Intent.GetStringExtra("poi");
    detailFragment.Arguments.PutString("poi", poiJson);
}
```

4. The `POIDetailActivity` layout contains a placeholder layout where the `POIDetailFragment` will be added. The `FragmentTransaction` class can be used to perform any of the fragment transactions, such as adding, replacing, or deleting fragments. Instantiate the fragment transaction to add `POIDetailFragment` to the `POIDetailActivity` view hierarchy:

```java
FragmentTransaction ft = FragmentManager.BeginTransaction();
ft.Add(Resource.Id.poiDetailLayout, detailFragment);
ft.Commit();
```

5. Let's now build and run the app in a simulator. Notice that you will see the same output that we built in Chapter 5, Adding a Details View. Now we are using `POIDetailFragment` that holds the logic to add, update, and delete POIs.

The same `POIDetailFragment` will be reused while creating the multi-pane split view layout for the tablet.
Using the list fragment for displaying the POI list

So far, we have created a fragment using the Fragment subclass to display the details of the point of interest. Let's now use a specialized ListFragment class to create a new fragment to display the POI list fetched from the server.

Understanding ListFragment

Before we start creating the POIListFragment, let's understand the following key concepts of ListFragment:

- **ListFragment** is a specialized Fragment subclass used to display the list of items from different data sources, such as an array or a cursor, that hold the query results.
- **ListFragment** provides a default layout that contains a single list view. However, it can be customized using your own custom layout.
- While using a custom layout for ListFragment, your layout must contain a **ListView** object with the ID @android:id/list.
- Like the normal ListView, the list fragment needs the instance of an adapter to operate on. It exposes a public ListAdapter property to set the list adapter.
- You can override **OnListItemClick()**, provided ListFragment allows you to detect user-click events on the list items.

Creating the POIListFragment layout

Here, in our example, the POIDetailsFragment will contain a ListView that takes the full width and height of the parent and a **ProgressBar** to the center of the screen. We will now create a custom layout for ListFragment.

Let's begin by creating a new layout file for the POI list fragment and name it POIListFragment.axml.

1. Select the Resources/Layout folder in the Solution pad.
2. Right-click on Add and select New File.
3. In the New File dialog box, click on Android and select Android Layout, enter POIListFragment in the Name field, and select New.
4. Copy the layout XML source code from POIList.axml and paste it in to the newly created POIListFragment.axml file.

5. Notice that we are declaring the custom layout for ListFragment. We must change the list view ID to @android:id/list.

The POIListFragment.axml file will have the following code snippets:

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:minWidth="25px"
    android:minHeight="25px">
    <ListView
        android:minWidth="25px"
        android:minHeight="25px"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@android:id/list" />
    <ProgressBar
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/progressBar"
        android:layout_centerInParent="true"
        android:visibility="gone" />
</RelativeLayout>
```

Creating the fragment to display the POI list using POIListFragment

Now that we have the layout ready for the POIListFragment, let's move ahead to create a new fragment to display the list of POIs. Note that we are not making any significant changes to the logic we have written for POIListActivity. Instead, we will refactor the same logic to work with fragments.
Perform the following steps to create POIListFragment and make it fully functional:

1. Select the POIApp in the Solution pad, right-click on Add, and select New File.
2. In the New File dialog box, click on Android and select New file, enter POIListFragment in the Name field, and select New.
3. Extend it from ListFragment and implement OnCreateView() to inflate the POI list fragment layout:

   namespace POIApp {
   public class POIListFragment: ListFragment {
   public override View OnCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
      View view = inflater.Inflate(Resource.Layout.POIListFragment, container, false);
      return view;
   }
   }
   }

   Note that the list fragments with the custom layout need to have a ListView object declaration with the ID @android:id/list.

4. Declare the following set of private variables to hold the instance of the list adapter, progress bar, and list of POIs, after the download is complete:

   private ProgressBar progressBar;
   private List<PointOfInterest> poiListData;
   private POIListViewAdapter poiListAdapter;

5. Override the OnAttach() method and save the reference of the activity instance:

   private Activity activity;
   public override void OnAttach (Activity activity) {
   base.OnAttach (activity);
   this.activity = activity;
   }
6. Add the following code blocks to the `OnCreateView()` method before the return statements to initialize the progress bar:

```csharp
progressBar = view.FindViewById<ProgressBar>(Resource.
Id.progressBar);
```

7. Create a new method named `DownloadPoisListAsync` (the same as declared in `POIListActivity`). We will be using the same logic we wrote earlier for `POIListActivity` with a small modification as follows. You will get a compilation error wherever the `this` keyword is used. Replace this with the activity instance:

```csharp
public async void DownloadPoisListAsync(){
    POIService service = new POIService();
    if (!service.isConnected(activity)) {
        Toast toast = Toast.MakeText(activity, "Not connected
to internet. Please check your device network settings.",
ToastLength.Short);
        toast.Show();
    } else {
        progressBar.Visibility = ViewStates.Visible;
        poiListData = await service.GetPOIListAsync();
        progressBar.Visibility = ViewStates.Gone;

        poiListAdapter = new POIListViewAdapter(activity,
        poiListData);
        thisListAdapter = poiListAdapter;
    }
}
```

Notice that in the preceding code blocks, we use the `ListAdapter` property defined in the `ListFragment` class to set the adapter instance.

8. Override the `OnResume()` life cycle method and call `DownloadPoisListAsync()` to start the download when a fragment is resumed:

```csharp
public override void OnResume ()
{
    DownloadPoisListAsync ();
    base.OnResume ();
}
```
9. Note that `POIListFragment` contributes to the action bar menu items and enables this option for the fragment by adding the following statement to `OnCreateView()` before the return statement:

```csharp
SetHasOptionsMenu (true);
```

10. Override the fragment version of the `OnCreateOptionsMenu` and `OnOptionsItemSelected` methods and paste in the following code. We are just reusing the same logic from `POIListActivity`:

```csharp
public override void OnCreateOptionsMenu (IMenu menu, MenuInflater inflater)
{
    inflater.Inflate(Resource.Menu.POIListViewMenu, menu);
    base.OnCreateOptionsMenu (menu, inflater);
}

public override bool OnOptionsItemSelected (IMenuItem item)
{
    switch (item.ItemId)
    {
    case Resource.Id.actionNew:
        Intent intent = new Intent (activity, typeof(POIDetailActivity));
        StartActivity (intent);
        return true;
    case Resource.Id.actionRefresh:
        DownloadPoisListAsync ();
        return true;
    default :
        return base.OnOptionsItemSelected(item);
    }
}
```

11. Override the `OnListItemClick()` method from `ListFragment` to detect the list row click action. The following code block is a copy of the same logic we wrote for the `POIClicked()` method in `POIListActivity`:

```csharp
public override void OnListItemClick (ListView l, View v, int position, long id)
{
    PointOfInterest poi = poiListData[position];
    Intent poiDetailIntent = new Intent(activity, typeof(POIDetailActivity));
```
string poiJson = JsonConvert.SerializeObject(poi);
poiDetailIntent.PutExtra("poi", poiJson);
StartActivity(poiDetailIntent);
}

At this point, the list fragment will display the list of POIs and is ready to be added to POIListActivity.

Adding POIListFragment to POIListActivity

For POIListFragment to work, it has to be added to POIListActivity. In the previous section, you learned how to add the fragment dynamically. Now let's add the POIListFragment using a declarative approach.

Modifying the POI list activity layout

A fragment can be added to an activity view hierarchy using the <fragment> tag declaration in its layout. The layout_width and layout_height attributes of the <fragment> tag are used to control the LayoutParams provided when attaching the fragment's view to the activity layout.

Make the following changes to the POI list activity layout (POIList.axml) to add POIListFragment statically:

```xml
<string version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:minWidth="25px"
    android:minHeight="25px"
    android:orientation="horizontal">
    <fragment class="POIApp.POIListFragment"
        android:id="@+id/listFragment"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

The value of the <fragment> attribute should be a fully qualified fragment class name. The class attribute indicates that the specified fragment class will be attached to the activity's content layout.
Adding POIListFragment to POIListActivity

Now that we are ready with POIListFragment, and we have attached it to the POI list activity content layout, we do not have to do anything much inside POIListActivity. Update the existing POIListActivity class with the following code blocks:

```csharp
using Android.App;
using Android.Os;

namespace POIApp
{
    [Activity (Label = "POI List")]
    public class POIListActivity : Activity
    {
        protected override void OnCreate (Bundle bundle)
        {
            base.OnCreate (bundle);
            SetContentView (Resource.Layout.POIList);
        }
    }
}
```

So far, we have done a great deal of work by creating two fragments that hold the logic to download and display the POI details. Now it's time to compile and run the app. Compile and run the application on the Android emulator:
Notice that the app will download the data from the POI web service and display the list of POIs on the scrollable list view. It works in exactly the same way as it did when we built it in Chapter 5, Adding a Details View, but now it works with fragments.

Creating a multi-pane layout for tablets

By now, you have a fair idea of how fragments work, and we have refactored the existing POIApp to work with fragments. The following sections will walk you through the key steps required to build the multi-pane layout and make the application compatible with Android tablets.

In Chapter 6, Making App Orientation Aware, we discussed the various configuration qualifiers to be considered while adding alternative layouts in Android. One of the qualifiers is the device's screen size. The Android powered device sizes ranges from small, normal, large, xlarge to layout-xxlarge. Since Android 3.2, Android has recommended that you use the sw<\(N\)>dp configuration qualifier to define an extra large layout for a tablet.

Let's assume that we require at least 600dp of screen width to build a multi-pane layout for the POIApp. For this, we need add a new layout-sw600dp subdirectory under the Resources directory. The Android runtime will pick the layout placed under this directory if the device configuration matches 600dp of width.

For tablet configuration, the POIListActivity layout will host both POIListFragment and the POIDetailFragment in a single activity. The POI list fragment will be placed to the left of the screen covering 40 percent of the total width, and the details fragment is docked to the right, covering the remaining 60 percent of the total width.

Xamarin Studio provides an easy-to-use layout designer that allows you to alternatively add layouts for various device configurations without any hassle. You don't need to remember or manually add the different folder names for different configuration qualifiers. The designer takes care of creating, editing, and deleting alternative layouts for different configurations.
The following section shows the usages of the Xamarin Studio layout designer to add alternative layouts for Android tablets:

1. Open the `POIList.axml` layout file and click on the **Content** tab to open the layout designer.

2. Click on the **Alternative Layouts** button located in the top-left corner. This enables a special editor to manage the alternative layouts. The following screenshot depicts the special editor provided in Xamarin Studio to edit and manage the alternative layouts for different configurations:

3. Click on the **+New Version** button to add another version of the layout for a given configuration.
4. This will open a dialog asking for different configuration parameters. For now, we will consider the minimum width for the tablet layout to be $600\text{dp}$. Enter the smallest screen width value as $600$, and then click on **Add**:

![Create Layout Variation dialog](image)

5. Notice that a new folder with the name `layout-sw600dp`, and that a new version of the `POIList` layout will be created in the Project Explorer.

6. Select the `sw600dp` layout to edit.

7. Search for `FrameLayout` in the Toolbox and drag it to the simulator window to add it as a root element. This will be used to add `POIDetailFragment` dynamically when a user selects any POI from the list.

8. Click on `FrameLayout` in the **Document Outline** view and then click on the **Properties** window. Set the view ID to `poiDetailLayout`.
9. Now we have a list fragment and a frame layout in the screen. Let’s assign 40 percent of the screen width for the list to appear and 60 percent for the details layout. This can be done using the `weight` property.

Select `listFragment` from the **Document Outline** view and change the `weight` property to 2 and the `layout_width` property to 0dp. This allows the weight property to decide the width of the view. Now, select `detailsLayout` from the document outline view, and change the weight property to 3 and the `layout_width` property to 0dp.

10. Set the **Padding left** and **Padding right** properties to 20dp for both `listFragment` and `detailsLayout`.

The following screenshot depicts the layout built so far:

11. Click on the **Source** tab and notice that the following code gets generated:

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:minWidth="25px"
    android:minHeight="25px"/>
```
Updating the POIApp to support the multi-pane split layout

In an Android tablet, the POI activity layout is declared to host both the POI list and POI details fragments by sharing the same screen space. The POIListFragment is statically added to the layout, and POIDetailFragment will be added dynamically when a user selects any POI item from the list. The FrameLayout with the detailsLayout ID is used as a placeholder to host the POIDetailFragment. However, in mobile devices, when a user clicks on any POI list item, it will continue to start POIDetailActivity.

The following steps will take you through the changes required to make the POIApp work for the multi-pane layout in Android powered tablets:

1. First, we need to find out if the device is running in the multi-pane mode. This is useful in order to understand whether you want to start a new activity or update the fragment content in the same activity. Let's declare a static Boolean variable that holds the information about whether or not an app is running in the dual mode:

   ```java
   public static bool isDualMode = false;
   ```
2. If the detailsLayout is not null and visible, then we can assume that the device is running in the multi-pane view mode. Add the following code blocks to your POIListActivity in the OnCreate() method to initialize the isDualMode variable:

```java
var detailsLayout = FindViewById(Resource.Id.poiDetailLayout);
if (detailsLayout != null && detailsLayout.Visibility == ViewStates.Visible) {
    isDualMode = true;
} else {
    isDualMode = false;
}
```

3. Currently, the new action in POIListFragment takes the user to POIDetailActivity. Now for tablets, instead of invoking another activity, the POIDetailFragment will be shown on the right-hand side of the same activity.

Add the following code blocks to the OnOptionsItemSelected() method under a new action:

```java
case Resource.Id.actionNew:
    if (POIListActivity.isDualMode) {
        var detailFragment = new POIDetailFragment();
        FragmentTransaction ft = FragmentManager.BeginTransaction();
        ft.Replace(Resource.Id.poiDetailLayout, detailFragment);
        ft.Commit();
    } else {
        Intent intent = new Intent(activity, typeof(POIDetailActivity));
        StartActivity(intent);
    }
    return true;
```

Note that, as we are not passing any details of the POI, the POIDetailsFragment will show the empty fields and allow the user to add a new POI.

4. As in the previous steps, we need the same implementation in the OnListItemClick() method. When a user clicks on the POI list item, instead of taking them to another activity, the POI details will be shown on the same activity in the right-hand details pane.
Update the `OnListItemClick()` method with the following code snippets:

```csharp
public override void OnListItemClick (ListView l, View v, int position, long id)
{
    PointOfInterest poi = poiListData[position];
    if (POIListActivity.isDualMode) {
        var detailFragment = new POIDetailFragment();
        detailFragment.Arguments = new Bundle();
        FragmentTransaction ft = FragmentManager.BeginTransaction();
        ft.Replace (Resource.Id.poiDetailLayout, detailFragment);
        ft.Commit();
    } else {
        Intent poiDetailIntent = new Intent(activity, typeof(POIDetailActivity));
        poiDetailIntent.PutExtra("poi", JsonConvert.SerializeObject(poi));
        StartActivity(poiDetailIntent);
    }
}
```

We are almost done! Let's now run the POIApp in the Android tablet emulator. You will notice that the POI app shows the multi-pane layout, as shown in the following screenshot:
Working with DialogFragment

In this chapter, we already covered how to use the fragment and ListFragment to optimize layout for Android tablets. Let's now discuss another specialized fragment subclass, DialogFragment. The current implementation of the POIApp uses AlertDialog to display the confirmation dialog to a user before deleting the POI. In this section, we will replace the AlertDialog implementation with DialogFragment.

DialogFragment is used to display a fragment as a floating dialog window that pops up on top of the current window. DialogFragment is the subclass of fragment and was introduced in Android 3.0 API level 11. Google recommends that you use DialogFragment for implementing the dialog because of the following reasons:

- Like normal fragments, DialogFragment manages its own life cycle. Events such as the user pressing the device's Back button or rotating the screen are handled in DialogFragment.
- A DialogFragment UI can be reused and embedded inside another activity. For example, if you wish your dialog to appear differently on different screen sizes, you can embed the DialogFragment inside the activity layout, which is similar to normal fragments.
- Although DialogFragment was introduced in Android 3.0, Google has released the Android support library, using which you can use the DialogFragment class for devices running on Android 1.6 or higher. The subsequent section (fragment for older Android devices) will describe how to add the Android Support Library to the Xamarin Studio solution.

Keeping the preceding points in mind, we will replace the AlertDialog and use DialogFragment in the POIDetailFragment class. Creating a dialog fragment is similar to the process of creating normal fragments. You need to first create a class that inherits from DialogFragment and overrides one of the following methods to provide the view hierarchy for your dialog:

- OnCreateView(): This method is used to inflate the layout for the dialog. It is the same as the Fragment.onCreateView() callback. If you want your fragment UI to be reusable / embedded inside another view, you must override OnCreateView() and inflate the layout.
- OnCreateDialog(): This method returns an instance of Dialog. This method is called automatically while creating the layout for the dialog. As we have already used AlertDialog in the POIApp; in this example, we will override the OnCreateDialog() to create the DialogFragment.
Chapter 7

The following steps will show you how to use the DialogFragment class to show the Delete confirmation dialog to a user:

1. Create a new fragment class and name it DeleteDialogFragment.
2. Inherit the DeleteDialogFragment class from DialogFragment:
   ```java
   public class DeleteDialogFragment : DialogFragment
   {
   }
   ```
3. Override the OnCreateDialog() method and add the following code snippets. The code blocks used in the OnCreateDialog() method are similar to the code we used in POIDetailFragment in the DeletePOI() method:
   ```java
   public override Dialog OnCreateDialog(Bundle savedInstanceState)
   {
Activity);
       alertConfirm.SetTitle("Confirm delete");
       alertConfirm.SetCancelable(false);
       alertConfirm.SetPositiveButton("OK", delegate {});
       alertConfirm.SetNegativeButton("Cancel", delegate {});
       alertConfirm.SetMessage("Are you sure you want to delete?");
       return alertConfirm.Create();
   }
   ```
4. We have constructed a basic version of DialogFragment. Now we need to initialize the fragment and use the FragmentTransaction to display the dialog box. Add the following code snippets to the POIDetailFragment DeletePOI() method:
   ```java
   FragmentTransaction ft = FragmentManager.BeginTransaction();
   DeleteDialogFragment dialogFragment = new DeleteDialogFragment();
   dialogFragment.Show(ft, "dialog");
   ```
   Like any other fragment, the operations on the dialog fragments are done using fragment transactions. The Show() method is used to add the dialog fragment to the activity view hierarchy and then commit the transaction. This takes two parameters: an instance of the fragment transaction, and a tag. The string value tag can be used later to retrieve the fragment instance from the fragment manager.
5. Now run and test the POIApp. Notice that the dialog fragment works great. However, it doesn’t display the name of the POI that the user wants to delete.
6. To display the name of the POI in the **Delete** confirmation dialog, we need to pass the POI name to `DeleteDialogFragment`. This can be accomplished using the fragment `Arguments` property.

Add the following code snippets to the `DeletePOI()` method after the fragment initialization and before calling `dialogFragment.Show()`:

```java
Bundle bundle = new Bundle();
bundle.putString("name", _poi.Name);
dialogFragment.Arguments = bundle;
```

7. Now we need to make the following changes to the `DeleteDialogFragment` class in the `OnCreateDialog()` method to retrieve and display the name of the POI:

```java
public override Dialog OnCreateDialog(Bundle savedInstanceState)
{
    string poiName = Arguments.GetString("name");
    Activity);
    ...
    ...
    alertConfirm.SetMessage(String.Format("Are you sure you want to
delete {0}?", poiName));
    return alertConfirm.Create();
}
```

8. Run the POIApp and notice that the **Delete** confirmation dialog now displays the name of the POI to be deleted. However, when you click on the **OK** button, the dialog is dismissed but the delete action is not working. We will pass the event callback from the fragment dialog to the `POIDetailFragment`. This can be easily accomplished using the `TargetFragment` property.

While adding the dialog fragment, we need to provide the target fragment information to the dialog by calling the `SetTargetFragment()` method. This method accepts two parameters: an instance of fragments that is the target of the dialog, and an optional integer request code.

The following code snippet shows the changes required in the `DeletePOI()` method:

```java
protected void DeletePOI()
{
    FragmentTransaction ft = FragmentManager.BeginTransaction();
```
DeleteDialogFragment dialogFragment = new DeleteDialogFragment();
    dialogFragment.SetTargetFragment (this,0);
    Bundle bundle = new Bundle();
    bundle.PutString("name", _poi.Name);
    dialogFragment.Arguments = bundle;
    dialogFragment.Show(ft, "dialog");
}

9. Now we can access the instance of POIDetailsFragment from the
dialog fragment using the TargetFragment property, and call the
DeletePOIAasync() method to initiate the delete POI web service request.

The following code shows what should be present in the
DeleteDialogFragment class:

    public class DeleteDialogFragment : DialogFragment
    {
        public override Dialog OnCreateDialog (Bundle savedInstanceState)
        {
            POIDetailFragment targetFragment = (POIDetailFragment)
            TargetFragment;

            string poiName = Arguments.GetString("name");

            AlertDialog.Builder alertConfirm = new AlertDialog.
            Builder(this.Activity);
            alertConfirm.SetTitle("Confirm delete");
            alertConfirm.SetCancelable(false);
            alertConfirm.SetPositiveButton("OK", (sender, e) => {
                targetFragment.DeletePOIAasync();
            });
            alertConfirm.SetNegativeButton("Cancel", delegate {});
            alertConfirm.SetMessage(String.Format("Are you sure you want
to delete {0}?", poiName));
            return alertConfirm.Create ();
        }
    }

Now run and test the POIApp. Notice that the DeletePOI is fully functional with
FragmentDialog.
Fragments for older Android devices

As we discussed earlier, the Fragment API was added in Android 3.0 API level 11. In the POIApp example discussed so far, I have assumed that the user who tests my application will be using Android 3.0 or later. However, if your business demands that you support your app for older devices, then you can still take advantage of the new Fragment API classes.

As there is some portion of the market still running on 3.0, Google provides support libraries that allow you to implement fragments while maintaining backward compatibility. The Fragment features are available in the V4 support library, and it works in Android 1.6 API level 4.

You cannot directly use the Fragment class as we have used it in this chapter. You have to add the Support package to your solution. For users of Xamarin Studio 5.9.x or above, while creating a new solution, you can select Maximum Compatibility for the targeted platform selection to include support packages in your project.

Alternatively, you can add Android support packages by navigating to Packages | Add packages. This will open the NuGet package browser, as shown in the following screenshot. You can search for Support packages and click on the Add Package button to add the selected package to your project:
After the Android Support packages are added to the project, you can start implementing the fragment features for older Android devices. The Support package fragment classes work pretty much similar to the native Android 3.0 compatible fragment classes. However, the following are some of the changes required by your app for using the supported compatible APIs:

- Your activity that will host the support fragment must extend now and inherit from `Support.V4.App.FragmentActivity`
- Use `SupportFragmentManager` instead of `FragmentManager`

Handling configuration changes with fragments

In Chapter 6, Making Your App Orientation-aware, we discussed how Android activities respond to device configuration changes. In this section, we will take a look at the fragment behavior in the event of configuration changes.

Like activities, fragment provides the `OnSaveInstanceState()` method that can be used to save the state of your fragment. This works in pretty much the same way as its activity counterpart. The `OnSaveInstanceState()` method supplies a Bundle instance into which we can dump the data bundles. The saved fragment instance can be retrieved from the `OnCreate()`, `OnActivityCreated()`, or `OnCreateView()` callback.

The following steps will guide you through saving and retaining the POI list scroll position when the device configuration changes:

1. Get the index of the first visible list item and save it to a bundle in the `OnSaveInstanceState()` method:

   ```csharp
   public override void OnSaveInstanceState (Bundle outState)
   {
       base.OnSaveInstanceState (outState);
       int currentPosition = ListView.FirstVisiblePosition;
       outState.PutInt ("scroll_position", currentPosition);
   }
   
   In the preceding code snippet, the string `scroll_position` is used as the key to save the current scroll position of `ListView`.```
Designing for Multiple Screen Sizes

2. Restore the saved list's scrolling position in `OnCreate()`. Note that we have to use the same key that we used to save:

```csharp
public override void OnCreate(Bundle savedInstanceState)
{
    base.OnCreate(savedInstanceState);
    if (null != savedInstanceState) {
        scrollPosition = savedInstanceState.GetInt("scroll_position");
    }
}
```

3. Add the following code snippet to the `DownloadPoisListAsync()` method in the `POIListFragment` class. This queues a message to scroll the POI list view to the previously saved scroll position:

```csharp
public async void DownloadPoisListAsync()
{
    POIService service = new POIService();
    if (!service.isConnected(activity)) {
        ...
    } else {
        ...
        this.ListAdapter = poiListAdapter;
        ListView.Post(() => {
            ListView.SetSelection(scrollPosition);
        });
    }
}
```

Build and run the app on the Android device or emulator. Change your device orientation and notice that the list view scroll position is now retained.
Summary
In this chapter, we covered a lot of ground on creating and managing fragments, including the following topics:

- The `Fragment` class and what it does, and creating and managing fragments
- How to use a declarative approach for adding fragments to activities using the `<fragment>` tag
- Using `ListFragment` with `ListAdapter` to populate the data
- Using `ListFragment` with `ListAdapter` to populate the data
- How to add alternative layout resources from Xamarin Studio layout designer
- How to reuse the fragments to create a multi-pane split view layout for Android tablets
- How to create a dialog using `DialogFragment`, and create its view hierarchy using the `OnCreateDialog()` method
- Optimizing the `POIApp` layout to support large-screen Android tablets
- Dealing with configuration changes to save and retain the fragment state

The next chapter will guide you to deal with data persistence using SQLite in Android.

Let’s take a look at the following references:

- [http://android-developers.blogspot.in/2012/11/designing-for-tablets-were-here-to-help.html](http://android-developers.blogspot.in/2012/11/designing-for-tablets-were-here-to-help.html)
We now turn our attention to data storage requirements. We need a way to store a list of **Point of Interest (POI)** data fetched from the server and make the list accessible when a device runs offline. This chapter shows how to use the built-in SQLite database engine to store and retrieve the POI data in a cross-platform way. The following topics will be covered in this chapter:

- Approaches to data storage solutions
- Different app storage options in Android
- Using preferences to store key-value pairs
- SQLite database storage in Xamarin.Android
- Using the SQLite.NET ORM component
- Performing database CRUD operations
- Unit testing the Android app using NUnitLite
- Implementing caching to POIApp

Data storage solutions mainly fall into two general categories: storing in Cloud using web services or using the device's local storage. Saving data in the Cloud provides great advantages over local data storage options, but in certain cases, such as apps that run offline or for games, generally you are required to store the data locally in the device's storage.

The current POIApp uses web services to store the POI list in the server, and the mobile app retrieves the list by making REST API calls. So far, it works great. However, it always requires an Internet connection to fetch and display the POI list. Let's overcome this problem by storing the list of POIs locally and accessing them even when the device is offline.
Data storage in Android

Android supports several solutions that can be used for persisting data locally. The persisted data can be accessible even when the application shuts down or when a device restarts. The examples of such data include the device settings, contacts list, browser bookmarks, or any such app-specific data.

The following table depicts the different data storage options available in the Android platform:

<table>
<thead>
<tr>
<th>Storage options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared preferences</td>
<td>This is the private data storage for your application that persists primitive key-value data pairs only. When the user uninstalls the application, this data is removed.</td>
</tr>
<tr>
<td>Internal storage</td>
<td>This stores data in the device internal memory and is available until the app is uninstalled from the device. This data is private to your application and other apps cannot access it.</td>
</tr>
<tr>
<td>External file storage</td>
<td>This stores data in shared external storage such as an external SD card. The data stored is public. Other applications or users can access these files by connecting the device to a computer.</td>
</tr>
<tr>
<td>SQLite storage</td>
<td>This is a structured private data store. SQLite databases created from an app can only be accessed from the same application.</td>
</tr>
</tbody>
</table>

This chapter briefly covers the shared preferences and SQLite storage options only. You can research the internal and external storage options on your own from the official Xamarin website.

Shared preferences

Shared preferences are persistence key/value data pairs that are used to store primitive data pairs, such as bool, float, int, string, and long. The data saved in the Android preference is persisted across different app sessions and is private to the application it has created. Any other application cannot access it.
For using shared preferences to save data pairs, you first need to get an instance of the ISharedPreferences interface. A shared preference can be specific to an activity or made global to all the activities in the application. If you want to create a single preference file specific to an activity, you can use Activity.GetPreferences to get an instance of the ISharedPreferences interface, or you can call the GetSharedPreferences method on the application context by passing the preference name and mode of operation to get the application level preference:

```csharp
ISharedPreferences prefs = Application.Context.GetSharedPreferences("PREF_NAME", FileCreationMode.Private);
```

Now, let's call the Edit() method to get an instance of ISharedPreferencesEditor. This matches all of the changes made to values in shared preferences and only saves when a call to Commit() or Apply() is made:

```csharp
ISharedPreferencesEditor editor = prefs.Edit();
editor.PutInt("your_key1", 10);
editor.PutString("your_key2", "Xamarin Example");
editor.Apply();
```

To read values from shared preferences, we can use the GetXX() methods by providing the same key that was used while saving data, where xx represents the supported primitive types. The following code snippet retrieves the values stored in the previous step:

```csharp
var value1 = prefs.GetInt("your_key1", 0);
var value2 = prefs.GetString("your_key2", null);
```

**The SQLite database storage**

SQLite is an open source, lightweight, and transactional database engine shipped with mobile platforms including Android, iOS, and Windows phones. According to the official documentation, Android is using SQLite Version 3.4.0. It is a widely used, standalone, relational database engine that requires no separate server process.

The following are some of the advantages that make the SQLite database engine one of the most shipped database engines today:

1. It is an open source project and a large open source community is actively working on it.
2. It doesn't have a server, so it requires no server infrastructure.
3. It is a lightweight engine that provides a set of classes to perform database transactions.

4. The complete database is a single file stored in the device memory that is private to your application. The privacy of the SQLite database boils down to the platform filesystem privacy.

As such, we have decided to go with SQLite for caching the POI list. When the list is downloaded successfully, we will save the POI in the database and update it periodically whenever the download is successful. When a device runs offline or is unable to fetch the updated list from the server, the cached POI list can be presented on the screen.

Although SQLite is built in Android and provides a set of APIs to perform all the database CRUD (Create, Read, Update, and Delete) operations, we will use the SQLite.ORM component from the component store. The following section walks you through the steps to perform different database operations using the SQLite.ORM component.

**Using the SQLite.net ORM component**

SQLite.NET Object Relational Mapping (ORM) is an open source library that allows .NET and Mono applications to store data objects in the SQLite database. It is a lightweight and easy to use component that allows you to save and retrieve data objects, without having to worry about writing any SQL query. This is a free component available in the Xamarin Component Store. It was earlier designed to work for MonoTouch on iPhone, but later it was extended to support other platforms, including Android, Windows, and Silverlight platforms. Thanks to Frank A. Krueger for developing and maintaining this component.

There are three ways to add SQLite.NET ORM to the Xamarin application. Either you can add it by downloading the required files from the GitHub open source code repository, NuGet, or the Xamarin Component Store:

- **SQLite.NET is a very thin library with a single file. Download the SQLite.cs file from GitHub and add it to your application. The SQLite.NET library binds directly to the SQLite database engine on each platform.**

- **To add it from the Xamarin Component Store, you need to follow the same steps we used while adding the UrlImageViewHelper component in Chapter 4, Adding a List View.**
• To add it as a NuGet package bundle, you can navigate to Packages | Add packages from your Solution Explorer to open the NuGet gallery window. Search for SQLite.NET and select it to add it to your project.

Once the reference to the SQLite.NET ORM component is added to the project, we can focus on performing database operations.

**Using attributes to mark POIApp to be persisted**

SQLite.NET ORM brings a set of attributes that allows you to mark classes and fields that are to be persisted to the database. You can apply any of the following attributes to control the construction of a table:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>By default, the class names are used as table names. You can use this attribute if you want to specify your own table name. This is an optional attribute applied to a class.</td>
</tr>
<tr>
<td>Column</td>
<td>By default, the property names are used as column names. This attribute provides you the control to specify your own column name.</td>
</tr>
<tr>
<td>PrimaryKey</td>
<td>This property is the primary key of the table. Only single-column primary keys are supported.</td>
</tr>
<tr>
<td>AutoIncrement</td>
<td>This property is automatically generated by the database upon insert. The property type should be an integer and should also be marked with the PrimaryKey attribute.</td>
</tr>
<tr>
<td>Indexed</td>
<td>This property should have an index created for it.</td>
</tr>
<tr>
<td>Unique</td>
<td>This property will be unique in the table.</td>
</tr>
<tr>
<td>MaxLength</td>
<td>This property specifies the varchar max size. The default max length is 140.</td>
</tr>
<tr>
<td>Ignore</td>
<td>This property will not be in the table.</td>
</tr>
<tr>
<td>NotNull</td>
<td>This property cannot be left null.</td>
</tr>
</tbody>
</table>

For POIApp, let's use some of the attributes from the preceding list of the PointOfInterest class:

```csharp
using SQLite;
namespace POIApp
{
```
Creating Data Storage Mechanisms

```csharp
[Table("POITable")]
public class PointOfInterest
{
    [PrimaryKey, AutoIncrement, Column("_id")]
    public int Id { get; set; }

    [NotNull]
    public string Name { get; set; }

    [MaxLength(1000)]
    public string Description { get; set; }

    [MaxLength(150)]
    public string Address { get; set; }

    public string Image { get; set; }
    public double? Latitude { get; set; }
    public double? Longitude { get; set; }
}
```

In the preceding code snippet, notice that we declared that the table name will be `POITable`, the `Id` field is marked as the primary key, and the `Name` field cannot be null.

### Adding a database helper class

Now we need to create a standard class that will define the set of methods to allow basic CRUD operations. This class is likely to be accessed multiple times; therefore, we will mark this class as a singleton class to prevent multiple instances of the same class.

To create the database helper implementation class, perform the following steps:

1. Create a class called `DBManager`.
2. To make the `DBManager` class a singleton class, first declare the default constructor private:

   ```csharp
   public class DBManager
   {
       private DBManager()
       {
       }
   }
   ```
Now declare a private static instance of the \texttt{DBManager} class and do the early initialization. We need to add a static getter method to make the \texttt{DBManager} class accessible. Add the following code snippet to the \texttt{DBManager} class:

```csharp
private static readonly DBManager instance = new DBManager();
public static DBManager Instance
{
    get
    {
        return instance;
    }
}
```

Note that there are different ways to create a singleton class. I have chosen to use the singleton class using the \textit{early initialization} method. For more information on singleton class implementation, follow the MSDN developer site.

Before saving the data, we need to first create a database. Let's create a blank database or open an existing one by passing the file path to the \texttt{SQLiteConnection} class constructor. Add the following \texttt{CreateTable()} method to your \texttt{DBManager} class:

```csharp
SQLiteConnection dbConn;
private const string DB_NAME = "PointOfInterest_DB.db3";

public void CreateTable()
{
    var path = System.Environment.GetFolderPath(System.Environment.SpecialFolder.Personal);
    dbConn = new SQLiteConnection(System.IO.Path.Combine(path, DB_NAME));
    dbConn.CreateTable<PointOfInterest>();
}
```

Notice the following in the preceding code:

- The \texttt{DB_NAME} constant defines a name for the database. The database file will be created with this name and saved to the location specified in the path variable.
- The SQLite database file path can be different for different platforms. Both Android and iOS use the environment class to construct a valid path.
- The \texttt{SQLiteConnection} class constructor opens the database if it already exists with the specified name, or else it creates a new database.
- The \texttt{CreateTable()} method opens the table if it exists, or else it creates a new table.
Creating Data Storage Mechanisms

Creating or updating the POI record

Once your database connection is open, we can perform different database operations. To begin with, let's create a method named `SavePOI()` to save the POI object in the database. The SQLiteConnection class provides methods, such as `Insert`, `InsertOrReplace`, `InsertAll`, `Update`, and `UpdateAll`, to create or update a record in the database.

We will use the `InsertOrReplace` method, as it is convenient for us. This method looks up for the same record in the database, updates it if it already exists, or else inserts a new one. For the `InsertOrReplace()` method to work, your table must have a primary key.

Let's add the following code snippet to the `DBManager` class in the `SavePOI` method:

```csharp
public int SavePOI(PointOfInterest poi)
{
    int result = dbConn.InsertOrReplace (poi);
    Console.WriteLine ("{0} record updated!", result);
    return result;
}
```

Reading POI details from the database

The POIApp reads the list of POI objects while the device runs in the offline mode, or if the app has failed to download the POI list data from the server. Add the following `GetPOIListFromCache()` method to retrieve all the records from the database:

```csharp
public List<PointOfInterest> GetPOIListFromCache()
{
    var poiListData = new List<PointOfInterest> ()
    INumberable<PointOfInterest> table = dbConn.Table<PointOfInterest> ()
    foreach (PointOfInterest poi in table)
    {
        poiListData.Add (poi);
    }
    return poiListData;
}
```
The `GetPOIListFromCache` method returns the list of all POIs available in the database. The following method will help you if you are looking for a particular POI by its ID:

```csharp
public PointOfInterest GetPOI(int poiId)
{
    PointOfInterest poi = dbConn.Table<PointOfInterest>().Where(a => a.Id.Equals(poiId)).FirstOrDefault();
    return poi;
}
```

**Deleting the POI data from the database**

Like reading records from a database, we can either clear all of the database records or delete records one by one by the POI ID. Add the following methods to the `DBManager` class. They are straightforward and quite self-explanatory:

```csharp
public int DeletePOI(int poiId)
{
    int result = dbConn.Delete<PointOfInterest>(poiId);
    Console.WriteLine("{0} record effected!", result);
    return result;
}

public int ClearPOICache()
{
    int result = dbConn.DeleteAll<PointOfInterest>();
    Console.WriteLine("{0} records effected!", result);
    return result;
}
```

Now that we have defined all the methods to perform different database operations. The next step is to validate each method by writing unit test cases before using them in `POIApp`.

**Using the Xamarin.Android NUnitLite**

You may be familiar with a process called **test-driven development (TDD)**. At a high-level, the approach proposes that you create automated unit test cases to test the features that your software will need to support, and use these test cases to drive the development and unit testing cycle.
This chapter will not cover the concepts behind test-driven development in detail, but we will introduce a feature delivered with Xamarin.Android, which supports teams using TDD. This feature is NUnitLite. NUnitLite is a lightweight, open source testing framework, which is based on the same ideas as NUnit. This has been designed to use minimal resources, making it ideal for embedded and mobile software development.

When working with NUnitLite, you create classes called test fixtures. These classes contain test methods that are used to test the various facets of the testing target; in our case, the DBManager class. To designate a class as a test fixture or a method as a test method, NUnitLite uses the .NET attributes. Once the test fixtures and test methods have been created, Xamarin.Android provides a user interface that allows the tests to be executed within the Android emulator or on a device.

To start using NUnitLite, we need to create a test project in the Solution pad that we have been working with.

To create a test project, perform the following steps:

1. Select the POIApp solution from the Solution pad in Xamarin Studio.
2. Right-click on it and select Add New Project.
3. On the left-hand side of the New Project dialog box, go to C# | Android.
4. In the template list, in the middle of the dialog box, select the Android Unit Test project.
5. Enter POITestApp for the name and click on OK. The new unit test project is created and added to the POIApp solution.
6. Go to the Options dialog box for the new project, and set the Package name to POITestApp, and verify that the Target framework is set to the latest available Android SDK framework.

You will notice that the new unit test project has the following files:

- MainActivity.cs: This activity inherits from TestSuiteActivity and provides a test suite user interface when we run our tests. Basically, it allows us to run our tests and see the results.
- TestsSample.cs: This class acts as a test fixture and allows us to add test methods that will exercise the features provided by DBManager.

Now, we need to create the test methods in order to exercise the features of our database operation performed by the DBManager class.
Setting up for tests

NUnitLite provides a place to execute any initialization code that may be required by the tests. In our case, we need to create an instance of the DBManager class, which the test methods will later interact with. The Setup() method is the perfect place since it will be called before each test. The following will steps help you to set up a new test case in Xamarin Studio:

1. Rename the TestsSample.cs file to POITestFixture.cs. Rename the corresponding class inside the file as well.

2. In POITestApp, select References, right-click on it, and select Edit References. Select the Projects tab in the Edit References dialog box, check on the POIApp project, and click on OK. POITestApp needs to reference POIApp so that it can work with the DBManager class.

   Some version of Xamarin Studio have a bug, where the project references are not linked even after adding them to a test project. In such cases, you need go to the .NET Assembly tab in the Edit References dialog box and browse manually to the POIApp/bin/Debug/POIApp.dll file.

3. Open the POITestFixture class and delete all the other methods except Setup().

4. In the Setup() method, call the CreateTable() method from the DBManager class:

   [TestFixture]
   public class POITestFixture
   {
      [SetUp]
      public void Setup ()
      {
         DBManager.Instance.CreateTable ();
      }
   }

   [ 199 ]
Creating the test methods

Now the real work begins; we need to create test methods to test each significant scenario. In the case of the data service, we need to be sure that we cover the following topics:

- Creating a new POI
- Updating an existing POI
- Deleting an existing POI

There are many more scenarios we could choose to test, but the preceding small set should help you verify that the basics of our database logic are functioning.

The CreatePOI test

The first test method we will start with is CreatePOI() and, as the name implies, we will test the process of creating and saving a new POI. To accomplish this, we need to perform the following steps:

1. Create a new instance of PointOfInterest and fill out some attributes.
2. Call the SavePOI() method on the DBManager class.
3. The save method returns an integer, indicating the number of records updated. Ensure that it returns the value 1.
4. Call GetPOI() to retrieve the POI, based on the saved ID.
5. Use the Assert class to ascertain that the POI was retrieved (the reference is not null) and the name of the POI is what it was expected to be.

The following code shows an implementation of CreatePOI():

```csharp
[Test]
public void CreatePOI ()
{
    int testId = 1091;
    PointOfInterest newPOI = new PointOfInterest();
    newPOI.Id = testId;
    newPOI.Name = "New POI";
    newPOI.Description = "POI to test creating a new POI";
    newPOI.Address = "100 Main Street\nAnywhere, TX 75069";

    //Saving poi record
    int recordsUpdated = DBManager.Instance.SavePOI (newPOI);

```
// Check if the number of records updated are same as expected
Assert.AreEqual (1, recordsUpdated);

// verify if the newly create POI exists
PointOfInterest poi = DBManager.Instance.GetPOI (testId);
Assert.NotNull (poi);
Assert.AreEqual (poi.Name, "New POI");
}

The DeletePOI test
Next, we will implement DeletePOI(). Again, we want DeletePOI() to be independent of other tests, so we will first need to create a POI which will be deleted later.

On calling DeletePOI(), the following steps will be performed:

1. Create a new instance of PointOfInterest and fill out some attributes.
2. Call the SavePOI() method on the DBManager class.
3. GetPOI() to retrieve the POI, based on the saved ID.
4. DeletePOI() to delete the POI file and remove it from the database.
5. GetPOI() to retrieve the POI, based on the saved ID.
6. Use the Assert class to ensure that the POI is not found (the reference is null).

The following code shows an implementation of DeletePOI():

[Test]
public void DeletePOI ()
{
    int testId = 1019;
    PointOfInterest testPOI = new PointOfInterest ();
    testPOI.Id = testId;
    testPOI.Name = "Delete POI";
    testPOI.Description = "POI being saved so we can test delete";
    testPOI.Address = "100 Main Street\nAnywhere, TX 75069";
    DBManager.Instance.SavePOI (testPOI);

    PointOfInterest deletePOI = DBManager.Instance.GetPOI (testId);
    Assert.NotNull (deletePOI);
Creating Data Storage Mechanisms

```csharp
DBManager.Instance.DeletePOI(testId);

PointOfInterest poi = DBManager.Instance.GetPOI(testId);
Assert.Null(poi);
```

Additionally, we will implement `ClearCache()` to verify that the call to `ClearPOICache()` clears all the database records. We will perform the following steps will be performed in the `ClearCache` test method:

1. Call `ClearPOICache()` on the `DBManager` class to delete all records from the database.
2. Call `GetPOIListFromCache` on the `DBManager` class to get a list of records from the database.
3. Use the `Assert` class to ensure that the number of records fetched from the server is 0.

The following code shows an implementation of `ClearCache()`:

```csharp
[Test]
public void ClearCache()
{
    DBManager.Instance.ClearPOICache();
    List<PointOfInterest> poiList = DBManager.Instance.GetPOIListFromCache();
    Assert.AreEqual(0, poiList.Count);
}
```

**Executing the tests**

Now that the tests have been developed, we are ready to execute them. To do this, we simply run the test app using the Android emulator or a physical device. To execute the tests in an emulator, perform the following steps:

1. Run `POITestApp` using the Android emulator. Note that `POITestApp` is not set as the startup project, so you will need to select the project when you select Run. You can make `POITestApp` the startup project by selecting it, right-clicking on it, and choosing Set as Startup Project. Once `POITestApp` is running, you should see the following screen when the app has been deployed and started:
2. Execute the tests by clicking on the **Run Tests** label. You should see a green message label, indicating that all the tests have passed.

3. If the test case fails, the message will be shown in red, and you can further drill down into the tests to see the failure details.

So far, we have created the `DBManager` class and have a suite of automated tests to test the CRUD methods. It is now time to focus on filling in the logic so that `POIApp` can save and fetch the data from the database.

NUnitLite is a great framework to implement the unit testing strategy for the Android application development, but it's not all about testing. You need to implement some sort of UI automation testing framework to test most of your application components, including the user interfaces.

Xamarin brings yet another platform; Xamarin Test Cloud makes it possible to test mobile apps written in any language on a collection of real devices from around the world. You can write your test scripts using the Xamarin testing framework and automate your app testing from CI systems. This book doesn't cover the UI automation frameworks and Xamarin Test Cloud service. You can visit the official [http://xamarin.com/](http://xamarin.com/) website for more information.
Implementing the caching logic to POIApp

The DBManager class is now tested and ready to use straightaway on POIApp. The POIApp will save the POI records when the download is completed, and the list will be retrieved later when the app is unable to fetch the updated list from the server. Perform the following steps to integrate the DBManager class with POIApp:

1. Call the CreateTable() method on the POIListActivity class to initiate the database:
   ```csharp
   DBManager.Instance.CreateTable();
   ```

2. Open the POIListFragment class and go to the DownloadPoisListAsync() method. We have used this method to download the POI list from the server. The following updates are required here:
   1. Clear the database cache when the POI downloads from the server are successful.
   2. Save the newly fetched POI data to the database.
   3. When a device is not connected to the network, return the cached data from the database.

   The following code snippet depicts the updated version of the DownloadPoisListAsync() method:
   ```csharp
   public async void DownloadPoisListAsync(){
       POIService service = new POIService();
       if (!service.is_connected(activity)) {
           Toast toast = Toast.MakeText(activity, "Not connected to internet. Please check your device network settings.", ToastLength.Short);
           toast.Show();
           poiListData = DBManager.Instance.GetPOIListFromCache();
       } else {
           progressBar.Visibility = ViewStates.Visible;
           poiListData = await service.GetPOIListAsync();

           //Clear cached data
           DBManager.Instance.ClearPOICache();

           //Save updated POI data
   ```
In the preceding code block, we save all of the POI list data received from the server. However, when a user edits and saves any of the POI data, we need to update the local database for that POI.

To do this, let’s call the `SavePOI()` method from the `POIDetailFragment` class in the `CreateOrUpdatePOIAsync()` method:

```csharp
private async void CreateOrUpdatePOIAasync(PointOfInterest poi){
    .....  
    .....  
    .....  
    if (!string.IsNullOrEmpty (response)) {
        Toast toast = Toast.MakeText (activity, String.Format ("{0} saved.", _poi.Name), ToastLength.Short);
        toast.Show();

        DBManager.Instance.SavePOI (poi);

        if(!POIListActivity.isDualMode)
            activity.Finish ();
    } else {
        Toast toast = Toast.MakeText (activity, "Something went Wrong!", ToastLength.Short);
        toast.Show();
    }
}
```

When the POI record is deleted from the server in the `POIDetailFragment` class, in the `DeletePOIAasync()` method we need to call the `DeletePOI()` method to delete the same POI from the local device database.
Creating Data Storage Mechanisms

The following code snippet depicts the updated version of the `DeletePOIAsync()` method:

```csharp
public async void DeletePOIAsync()
{
    ....
    ....
    string response = await service.DeletePOIAsync (_poi.Id);
    if (!string.IsNullOrEmpty (response)) {
        Toast toast = Toast.MakeText (activity, String.Format ("{0} deleted.", _poi.Name), ToastLength.Short);
        toast.Show();
        DBManager.Instance.DeletePOI (poi);
        if (!POIListActivity.isDualMode)
            activity.Finish ();
    } else {
        Toast toast = Toast.MakeText (activity, "Something went Wrong!", ToastLength.Short);
        toast.Show();
    }
}
```

Now we have implemented the database logic for the POIApp to work offline. Run the app; when connected to the network, it will download and cache the POI list. Now restart the application while the device is offline and notice that the same old cached POI list is presented on the screen.

**Summary**

In this chapter, we used SQLite.ORM to save the POI records to the database and created a series of unit tests to verify that the database operations were functioning well.

In the next chapter, we will continue integrating with device capabilities by adding camera support to capture and save the POI image.
Making POIApp Location Aware

One of the most interesting aspects of mobile development is interacting with device capabilities such as motion sensors, cameras, and location sensors. Such features are contextual for most applications and carry a great deal of value to the users. In this chapter, we will walk you through adding location awareness to POIApp. We will cover the following topics:

- Setting application permissions
- Obtaining the current longitude and latitude
- Obtaining the address for a longitude and latitude
- Calculating the distance between two geo points
- Displaying a POI within the map app

Working with location services

Integrating the location service in the Android platform is more complicated than it looks.

You need to consider different location providers, location accuracy, user movements, and most importantly the nominal device battery power consumption. While working with location services on the Android platform, you will primarily work with an instance of LocationManager. The LocationManager class provides you with the ability to obtain periodic updates of the device geo location or fires an event when the device enters the proximity of a given geographical location.
Android devices generally provide two different means of determining a location: **GPS** and **Network**. When requesting location change notifications, you may specify the provider you wish to receive updates from. The Android platform defines a set of string constants for the following providers:

<table>
<thead>
<tr>
<th>Provider name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPS_PROVIDER (GPS)</td>
<td>This provider determines a location using satellites. Depending on conditions, this provider may take a while to return a location fix. This requires the ACCESS_FINE_LOCATION permission.</td>
</tr>
<tr>
<td>NETWORK_PROVIDER (network)</td>
<td>This provider determines a location based on the availability of a cell tower and Wi-Fi access points. Its results are retrieved by means of a network lookup. This requires the ACCESS_COARSE_LOCATION permission.</td>
</tr>
<tr>
<td>PASSIVE_PROVIDER (passive)</td>
<td>This provider can be used to passively receive location updates when other applications or services request them without actually having to request for the locations yourself. It requires the ACCESS_FINE_LOCATION permission. If the GPS is not enabled, this provider might only return coarse fixes.</td>
</tr>
</tbody>
</table>

The process of integrating the location service to an Android application includes the following steps:

1. Obtain a reference to an instance of `LocationManager`.
2. Use the instance of `LocationManager` to request location change notifications, either ongoing or a single notification.
3. Process the `LocationListener` callback methods. These methods will be fired only if the location is requested with the `RequestLocationUpdates(string, long, float, ILocationListener)` method.

Accessing the location service in Android applications requires specific permissions to be added, depending on the provider you want to use.
Setting the app permissions

To access location services in Android, you must provide permissions to the application AndroidManifest.xml file. Android applications use two permissions to access the location API: \texttt{ACCESS\_COARSE\_LOCATION} and \texttt{ACCESS\_FINE\_LOCATION}. The \texttt{ACCESS\_FINE\_LOCATION} includes the permission for both the GPS\_PROVIDER and NETWORK\_PROVIDER providers. The \texttt{ACCESS\_COARSE\_LOCATION} permission includes permission only for NETWORK\_PROVIDER.

To add the appropriate permissions to your application descriptor, perform the following steps:

1. Double-click on Properties/AndroidManifest.xml in the Solution pad. The file will be opened in the manifest editor. There are two tabs at the bottom of the screen, Application and Source, which can be used to toggle between viewing a form for editing the file or the raw XML.

2. In the Required permissions list, check AccessCoarseLocation, AccessFineLocation, and Internet. Navigate to File | Save:
3. Switch to the **Source** view to view the XML as follows:

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionName="1.0" package="com.packt.poiapp">
  <uses-sdk android:minSdkVersion="10" android:targetSdkVersion="19" />
  <application android:label="POIApp" android:icon="@drawable/ic_launcher">
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
  </application>
</manifest>
```

### Configuring the emulator

To use an **emulator** for development, this chapter will require the emulator to be configured with Google APIs so that the address lookup and navigation to map the app works.

To install and configure **Google APIs**, perform the following steps:

1. From the main menu, navigate to **Tools** and open **Android SDK Manager**.
2. Select the platform version you are using, check **Google APIs**, and click on **Install 1 package...**, as shown in the following screenshot:
3. After the installation is complete, close the Android SDK Manager, and from the main menu, navigate to Tools | Open Android Emulator Manager.

4. Select the emulator you want to configure and click on Edit.

5. In Target, select the Google APIs entry for the API level you want to work with.

6. Click on OK to save.

Obtaining an instance of LocationManager

The LocationManager class is a system service that provides access to the location and bearing of a device, if the device supports these services. You do not explicitly create an instance of LocationManager; instead, you request an instance from a Context object using the GetSystemService() method. In most cases, the Context object is a subtype of activity. The following code depicts declaring a reference of a LocationManager class and requesting an instance:

```java
LocationManager locMgr;
...
locMgr = (LocationManager) getSystemService(Context.LocationService);
```

Requesting location change notifications

The LocationManager class provides a series of overloaded methods that can be used to request location update notifications. If you simply need a single update, you can call RequestSingleUpdate(); to receive ongoing updates, call RequestLocationUpdate().

Prior to requesting location updates, you must identify the location provider that should be used. In our case, we simply want to use the most accurate provider available at the time. This can be accomplished by specifying the criteria for the desired provider using an instance of Android.Location.Criteria. The following code example shows how to specify the minimum criteria:

```java
Criteria criteria = new Criteria();
criteria.Accuracy = Accuracy.NoRequirement;
criteria.PowerRequirement = Power.NoRequirement;
```

Now that we have the criteria, we are ready to request updates as follows:

```java
locMgr.RequestSingleUpdate (criteria, this, null);
```
Implementing ILocationListener

You will notice that the second parameter to `RequestSingleUpdate()` must be an object that implements `ILocationListener`, which defines the following methods:

```java
void OnLocationChanged (Location location);
void OnProviderDisabled (string provider);
void OnProviderEnabled (string provider);
void OnStatusChanged (string provider, Availability status, Bundle extras);
```

For the most part, we will create blank stubs for all of the methods except `OnLocationChanged()`. While writing more sophisticated applications, it will be useful to provide implementations for some of the other methods. For example, you might call `RequestLocationUpdate()` to begin receiving updates and then receive a notification via `OnProviderEnabled()` that a preferred provider is now available, in which case you would want to stop the updates and start them again using the preferred provider.

Adding location services to POIApp

In POIApp, we have the following two different scenarios for requesting location updates:

- **In the POI list**, we need to calculate the distance of each of the listed POIs. In this scenario, we want to request location change notifications on an ongoing basis and use the most current location to calculate the distance.
- **In POIDetailFragment**, we would like to request the current location when adding a new POI. In this scenario, we would want to request a single location change notification.

Adding location services to the POI list

Now that we have some idea of how to add location services to an app, let's add location services to `POIListFragment` as follows:

1. Declare a private instance of `LocationManager` and obtain a reference in the `OnCreateView()` method as follows:

   ```java
   LocationManager locMgr;
   ...
   public override View OnCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)
   ```
{  
   View view = inflater.Inflate(Resource.Layout.POIListFragment,  
   container, false);  
   progressBar = view.FindViewById<ProgressBar> (Resource.  
   Id.progressBar);  
   SetHasOptionsMenu (true);  
   locMgr = (LocationManager) Activity.GetSystemService (Context.  
   LocationService);  
   return view;  
}

2. Include the Android.Locations namespace in POIListFragment and  
   implement the ILocationListener interface:  
   public class POIListFragment: ListFragment, ILocationListener  
   {  
   ...  
   }

3. From the code editor right-click on ILocationListener interface, select  
   Refactor | Implement interface to implement the stub methods. Remove  
   any code placed by default in the stub methods; we will provide logic for  
   OnLocationChange()

4. In OnResume(), obtain the best location provider and call  
   RequestLocationUpdates() to start receiving updates as follows:  
   public override void OnResume ()  
   {  
      base.OnResume ();  
      DownloadPoisListAsync ();

      Criteria criteria = new Criteria ();  
      criteria.Accuracy = Accuracy.NoRequirement;  
      criteria.PowerRequirement = Power.NoRequirement;  
      string provider = locMgr.GetBestProvider (criteria, true);  
      locMgr.RequestLocationUpdates (provider, 2000, 100, this);
   }

5. Override the OnPause() method on the POIListFragment class and add a  
   call to RemoveUpdates(). This eliminates unnecessary processing of location  
   changes when the POIListFragment class is not visible, as shown in the  
   following code:  
   protected override void OnPause ()  
   {  
      base.OnPause ();  
      locMgr.RemoveUpdates (this);
   }
6. Add a `CurrentLocation` property to `POIListViewAdapter`. The `POIListFragment` class will use this property to communicate location changes to the adapter:

   ```csharp
   public Location CurrentLocation { get; set; }
   ```

7. Add a logic to `OnLocationChanged()` to set `CurrentLocation` on `POIListViewAdapter` when a location change is received and call `NotifyDataSetChanged()` to cause the `ListView` to be refreshed as follows:

   ```csharp
   public void OnLocationChanged (Location location)
   {
       (this.ListAdapter as POIListViewAdapter).CurrentLocation = location;
       this.ListView.InvalidateViews ();
   }
   ```

8. Add a logic to the `GetView()` method on `POIListViewAdapter` to calculate the distance between the `CurrentLocation` and a POI's location properties and update `distanceTextView` with the results. The calculation should only be done if `CurrentLocation` is not null, and the Latitude and Longitude properties for the POI being added to the `ListView` are not null. If any of these values are null, simply place `??` in the distance field to indicate that it cannot be calculated at this time as follows:

   ```csharp
   var distanceTextView = view.FindViewById<TextView> (Resource.Id.distanceTextView);
   if (((CurrentLocation != null) && (poi.Latitude.HasValue) && (poi.Longitude.HasValue))
   {
       Location poiLocation = new Location ("");
       poiLocation.Latitude = poi.Latitude.Value;
       poiLocation.Longitude = poi.Longitude.Value;
       float distance = CurrentLocation.DistanceTo (poiLocation) * 0.000621371F;
       distanceTextView.Text = String.Format("{0:0,0.00} miles", distance);
   }
   else
   {
       distanceTextView.Text = "??";
   }
   ```

Now, run POIApp and view the results in POIListView.
Adding location services to POI details

The steps for adding location services to POIDetailFragment will be very similar to the previous section, but will be slightly simpler.

Updating the user interface

Prior to adding the logic, we need to add a few buttons to our app; one to get our location and one to navigate to the map, which we will cover later in this chapter. We can add these as a row of buttons at the bottom of POIDetailFragment.axml, as shown in the following screenshot:

Using ImageButton allows a drawable class to be specified. To add the ImageButton widgets, perform the following steps:

1. Add a LinearLayout instance to the bottom of POIDetailFragment.axml just below the TableLayout used to arrange the latitude and longitude content. The orientation should be horizontal.

2. On the LinearLayout instance, the orientation should be horizontal, the content should be wrapped (both height and width), and it should be centered horizontally in the parent. The layout gravity can be used to center the content horizontally within its parent. The top and bottom padding of 10dp will provide a good spacing for the buttons.

3. Add the two ImageButton widgets: locationImageButton and mapImageButton, in the LinearLayout instance. Images for these buttons can be found in the drawable folder in the assets location.
4. The following XML code shows the result:

```xml
... </TableLayout>
<LinearLayout
    android:orientation="horizontal"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:minWidth="25px"
    android:minHeight="25px"
    android:layout_marginTop="10dp"
    android:layout_marginBottom="10dp">
    <ImageButton
        android:src="@drawable/ic_locate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/locationImageButton" />
    <ImageButton
        android:src="@drawable/ic_map"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/mapImageButton" />
</LinearLayout>
```

**Adding the code**

Now that we have buttons on the UI, we can add the code to obtain the location as follows:

1. Declare a private instance of `LocationManager` and obtain a reference in `OnCreate()` in the same way we did for `POIListView` in the previous section:

   ```csharp
   locMgr = (LocationManager) Activity.GetSystemService(Context.LocationService);
   ``

2. Add a `GetLocationClicked` event handler and hook it up to the `ImageButton` as follows:

   ```csharp
   _locationImageButton = FindViewById<ImageButton>(Resource.Id.locationImageButton);
   _locationImageButton.Click += GetLocationClicked;
   ```
3. Add a call to RequestSingleUpdate() in GetLocationClicked(). The RequestSingleUpdate() method allows a Criteria object to be passed in so that we do not need a separate call to GetBestProvider() as follows:

```csharp
protected void GetLocationClicked(object sender, EventArgs e)
{
    Criteria criteria = new Criteria();
    criteria.Accuracy = Accuracy.NoRequirement;
    criteria.PowerRequirement = Power.NoRequirement;
    locMgr.RequestSingleUpdate (criteria, this, null);
}
```

4. Specify that POIDetailFragment implements Android.Locations. The ILocationListener implements stub methods using the Refactor | Implement interface. Remove any code placed in the stub methods; we will provide a logic for OnLocationChange().

5. Add a logic to OnLocationChange() to update the location fields as follows:

```csharp
public void OnLocationChanged (Location location)
{
    _latEditText.Text = location.Latitude.ToString();
    _longEditText.Text = location.Longitude.ToString();
}
```

**Mocking location data for testing**

Run POIApp and try adding a new POI and getting the location. While running the app in the emulator, you will notice that apparently nothing happens when you click on the location button. The app is actually waiting for a callback to OnLocationChanged() from the location manager. To trigger this callback, you must use the Android Device Monitor (ADM) to mock the location data.

To trigger OnLocationChanged(), perform the following steps:

1. Start ADM and select the emulator instance on the left-hand side of the screen.
2. Click on the **Emulator Control** tab on the right-hand side of the screen. If the **Emulator Control** tab is not present, navigate to Window | Show View to display the tab. Notice that, at the bottom of the panel, there is a nested tab titled **Location Controls**, as shown in the following screenshot:

3. Select the **Manual** tab, adjust **Longitude** and **Latitude**, if desired, and click on **Send**. This will cause the `OnLocationChanged()` method to be fired on `POIDetailFragment`.

Note the other two tabs under **Location Controls**: GPX and KML. These tabs can be used to load a series of location updates from a file and play them back to your app to test more sophisticated scenarios.

**Getting an address for a location**

Another useful feature provided by the Android platform is called **Geocoding**. This is the process of obtaining the location in terms of latitude and longitude from a known address. Android also supports reverse geocoding, which is what you would expect from the name, obtaining an address from a known location.
The `Android.Locations.Geocoder` class is the class used to perform geocoding and reverse geocoding. It’s a very straightforward process to use, as shown in the following steps:

2. Call `GetFromLocation()` by passing in the location you would like to find the address for.
3. Process the `IList<Address>` collection returned. The collection of addresses returned from `GetFromLocation()` vary in specifics, which means that some are specific street addresses, some specify a city, country, and so on. The first address is always the most specific, so we will automatically choose it using the following code:

   public void OnLocationChanged (Location location)
   {
       _latEditText.Text = location.Latitude.ToString();
       _longEditText.Text = location.Longitude.ToString();
       Geocoder geocdr = new Geocoder(activity);
       IList<Address> addresses = geocdr.GetFromLocation (location.
       Latitude, location.Longitude, 5);
       if (addresses.Any()) {
           UpdateAddressFields (addresses.First());
       }
   }

Notice that the number 5 parameter in the `GetFromLocation()` method call represents the maximum number of results that are known to describe the area, surrounding the latitude and longitude.

You can see that we chose to call a method to format the address information. The `FeatureName` property may contain a title such as *Golden Gate Bridge* or *Empire State Building*. More times than not, `FeatureName` will simply contain the street number. The address contains a list of address lines, which we combine and place in `_addrEditText` as follows:

   protected void UpdateAddressFields(Address addr)
   {
       if (String.IsNullOrEmpty(_nameEditText.Text))
           _nameEditText.Text = addr.FeatureName;
       if (String.IsNullOrEmpty(_addrEditText.Text))
       {
           for (int i = 0; i < addr.MaxAddressLineIndex; i++)
           {
               }
Making POIApp Location Aware

```csharp
if (!String.IsNullOrEmpty(_addrEditText.Text))
_addrEditText.Text += addr.GetAddressLine (i);
```

Now, run POIApp and try adding a new POI and getting the address for a location.

**Keeping the user informed**

After using the get location button, you will notice that requests for location information take some amount of time to process; generally, a few seconds or more. It would be best to keep users informed that processing is taking place so that they don't continually click on the button.

This can be achieved in two ways. You can either attach a `ProgressBar` directly to the details fragment layout as we did for POIListFragment, or we can use a `ProgressDialog` class. The `ProgressDialog` class provides a simple means of displaying a dialog with a spinning progress widget and text description of what process is taking place. Ever since the `DialogFragment` class is added to Android, Google recommends that you use the `DialogFragment` class instead of `ProgressDialog`. Here, in this example, we will create a `DialogFragment` class that displays the loading progress that is similar to the `DeleteDialogFragment` class.

The following steps will guide you to add a dialog fragment to display the loading progress:

1. Let's add a new fragment named `ProgressDialogFragment` and extend it from the `DialogFragment` class.
2. Override the `OnCreateDialog()` method and add the following code blocks:

   ```csharp
   public class ProgressDialogFragment : DialogFragment
   {
       public override Dialog OnCreateDialog (Android.OS.Bundle savedInstanceState)
       {
           Cancelable = false;
           ProgressDialog _progressDialog = new ProgressDialog (Activity);
           _progressDialog.SetMessage ("Getting location...");
           _progressDialog.Indeterminate = true;
           _progressDialog.SetProgressStyle (ProgressDialogStyle.Spinner);
           return _progressDialog;
       }
   }
   ```
Chapter 9

3. We are not ready with the ProgressDialogFragment fragment. Let's add the following code snippets to the GetLocationClicked() method to display the loading progress when a user clicks on the location button from POIDetailFragment:

   FragmentTransaction ft = FragmentManager.BeginTransaction();
   var dialogFragment = new ProgressDialogFragment();
   dialogFragment.Show(ft, "progress_dialog");

4. Now we need to remove the loading progress dialog once the location is retrieved from the location manager. Add the following code snippets to the OnLocationChanged() callback to remove the progress dialog:

   FragmentTransaction ft = FragmentManager.BeginTransaction();
   ProgressDialogFragment dialogFragment = (ProgressDialogFragment)FragmentManager.FindFragmentByTag("progress_dialog");
   if (dialogFragment != null) {
       ft.Remove (dialogFragment).Commit();
   }

Now, run POIApp and check the new progress dialog, as shown in the following screenshot:
Making POIApp Location Aware

Adding map integration
Maps are another truly cool part of mobile computing. They provide a means of navigation, finding points of interest in an area, as well as supporting many other useful scenarios.

There are two basic approaches to interfacing with maps from an app as follows:

- Navigate to the existing Android map app installed on the device to display a point of interest. Most recent Android devices are preinstalled with the Google Map application. However, it is not guaranteed.
- Integrate with the Google Maps API. This approach integrates the map view directly to your application using Google Play services. You must create an application in the Google Developer Console and obtain a copy of the API key for your app.

The first option is much easier to implement, whereas the second option allows tighter integration and control of the maps at the cost of more code and complexity. The second option requires very specific versions of the Xamarin.Android binding libraries corresponding to Google Play libraries. We chose to go with the first option for the POIApp example for the following reasons:

- It is very difficult to get the second option working inside an emulator, which means that you would have to test and view the results of the code on an actual device, which may not be an option for all readers
- We would need to dedicate more time than we have available in this chapter to get the second option up and running

Xamarin's official website contains articles with all the details required to get the second option working.

Navigating to the Map app
To navigate to the Map app, we will rely on the Intent class we used earlier in the book; however, rather than specifying the Activity class we want to start, we will specify the type of information we would like to view using a URI. Android contains a registry of apps that can display different types of information and will launch the most appropriate app.
The Android platform defines a set of `Intent` classes that can be used to launch Google apps on Android devices. The following table summarizes the `Intent` classes related to locations:

<table>
<thead>
<tr>
<th>URI</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>geo:latitude,longitude</code></td>
<td>This action opens the Map application centered at a latitude or longitude</td>
</tr>
<tr>
<td><code>geo:latitude,longitude?z=zoom</code></td>
<td>This action opens the Map application centered at a latitude or longitude and zoomed to the specified level</td>
</tr>
<tr>
<td><code>geo:0,0?q=my+street+address</code></td>
<td>This action opens the Map application to the location of a street address</td>
</tr>
<tr>
<td><code>geo:0,0?q=business+near+city</code></td>
<td>This action opens the Map application and displays the annotated search results</td>
</tr>
</tbody>
</table>

In our case, we have a street address, latitude and longitude, or both. If the street address is present, we should build the `Intent` class with it because this will cause the street address to appear in the Map app, making it more user friendly. If the street address is not present, we will build the `Intent` class using latitude and longitude. The following code shows the logic for building the `Intent` class:

```java
Android.Net.Uri geoUri;
if (String.IsNullOrEmpty(_addrEditText.Text)) {
    geoUri = Android.Net.Uri.Parse(String.Format("geo:{0},{1}", _poi.Latitude, _poi.Longitude));
} else {
    geoUri = Android.Net.Uri.Parse(String.Format("geo:0,0?q={0}", _addrEditText.Text));
}
Intent mapIntent = new Intent(Intent.ActionView, geoUri);
```

Prior to launching the `Intent` class, we need to be sure there is an app that can handle the `Intent` class; otherwise, we might end up with an unhandled exception from `StartActivity()`.
Checking for registered map apps

Apps provide information about any capabilities they provide (the Intent classes) in their manifest files as an <intent-filter/> element. Since we are relying on an external map app to display our location for us, we should check whether such an app exists on the device we are running on. We can accomplish this with a few calls to the PackageManager class. The PackageManager class allows you to retrieve various types of information about the application packages installed on a device. The QueryIntentActivities() method allows you to check whether there are any apps available to handle a specific Intent class. The following code demonstrates the use of QueryIntentActivities():

```
PackageManager packageManager = Activity.PackageManager;
IList<ResolveInfo> activities = packageManager.QueryIntentActivities(mapIntent, 0);
if (activities.Count == 0) {
    Toast.MakeText(activity, "No map app available.", ToastLength.Short).Show();
} else {
    StartActivity(mapIntent);
}
```

Create a MapClicked() event handler and attach it to the _mapImageButton. The following code snippet represents the complete MapClicked() code that is used to open a Map application:

```
protected void MapClicked(object sender, EventArgs e){
    Android.Net.Uri geoUri;
    if (String.IsNullOrEmpty(_addrEditText.Text)) {
        geoUri = Android.Net.Uri.Parse(String.Format("geo:{0},{1}", _poi.Latitude, _poi.Longitude));
    } else {
        geoUri = Android.Net.Uri.Parse(String.Format("geo:0,0?q={0}", _addrEditText.Text));
    }
    Intent mapIntent = new Intent(Intent.ActionView, geoUri);
    PackageManager packageManager = Activity.PackageManager;
    IList<ResolveInfo> activities = packageManager.QueryIntentActivities(mapIntent, 0);
}
```
if (activities.Count == 0) {
    Toast.MakeText (activity, "No map app available.", ToastLength.
    Short).Show ();
} else {
    StartActivity (mapIntent);
}

Run POIApp and click on the Map button from the POI details page. You will notice
that the Map app will be opened with the POI location. You have the option of
choosing to navigate to it from your current location.

**Summary**

In this chapter, we saw how to use device location services to find the current
position. We also integrated POIApp with the native device map to position the
POI on the map. In the next chapter, we will continue integrating with device
capabilities by adding integration with the camera.
Another exciting feature of mobile computing is that most Android devices have some types of camera that can be used to capture photos and/or videos. This chapter will walk you through the steps required to add the ability to capture and upload a picture of a POI and will include the following topics:

- Approaches to integrate with a device camera
- Camera permissions and features
- Capturing and displaying a photo
- Uploading an image using an HTTP multipart form upload

### Picking an integration approach

The Android platform provides two different ways to integrate a device's camera feature to your app:

- Using existing camera apps to integrate using the `Intent` method
- Creating your own custom activity that interacts directly with the camera using the Android API

The second approach allows a high degree of control over how the camera view is presented to the user, and how the user interacts with the view. However, the first approach is very straightforward to implement as it reuses the existing device camera apps to capture the picture. We will go with the `Intent` method as it represents a very practical way to add camera integration.
Permissions and features

Before getting into the details of integrating the device camera feature, we will discuss more about the general permissions and features related to the camera in more depth. The following table contains the various permissions that may be required. In our case, we need not specify any of these because we are using the Intent method, where the external camera app will capture the picture for us. The external camera app needs to specify the required camera permissions:

<table>
<thead>
<tr>
<th>Permission</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAMERA</td>
<td>This is required for apps that request permission to use the device's camera; this is not required if you make use of the Intent method</td>
</tr>
<tr>
<td>WRITE_EXTERNAL_STORAGE</td>
<td>This permission is required to save images or videos to the device's external storage (SD card)</td>
</tr>
<tr>
<td>RECORD_AUDIO</td>
<td>If your app records audio with video capture</td>
</tr>
</tbody>
</table>

Camera-specific features can be set up in an app's manifest file using the <uses-feature> element. The <uses-feature> declaration is used to inform about the set of hardware and software features on which your application depends. You may specify the required=true to declare that your application cannot function without the declared feature. The feature declaration is used for information purpose only. There is no validation done by the Android system before installing the app.

The following table depicts the set of features you may define in your application manifest declaration:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>android.hardware.camera</td>
<td>The application uses the device's camera. If the device supports multiple cameras, the application uses the camera that faces away from the screen.</td>
</tr>
<tr>
<td>android.hardware.camera.autofocus</td>
<td>Subfeature. The application uses the device camera's autofocus capability.</td>
</tr>
<tr>
<td>android.hardware.camera.flash</td>
<td>Subfeature. The application uses the device camera's flash.</td>
</tr>
<tr>
<td>android.hardware.camera.front</td>
<td>Subfeature. The application uses a front-facing camera on the device.</td>
</tr>
<tr>
<td>android.hardware.camera.any</td>
<td>The application uses at least one camera, which can face in any direction. Use this in preference to android.hardware.camera if a back-facing camera is not required.</td>
</tr>
</tbody>
</table>
In our case, we will not specify any features as requirements, but at runtime, we will check whether an external app is available to capture a photo.

**Configuring the emulator**

If you are using the emulator for development, you will need to configure it to have a camera. If the computer you are using has a webcam, the emulator can use it as the camera; otherwise, you can choose to have an emulated camera.

To configure the emulator for a camera, perform the following steps:

1. From the main menu, navigate to **Tools** and open **Android Emulator Manager**.
2. Select the emulator you have been working with and choose **Edit**.
3. In the middle of the Edit AVD dialog, you will see two dropdowns; one for **Front Camera** and one for **Back Camera**. Make your selections and click on **OK**.
Extending the data service

Since we have decided to use an external camera app to capture the picture, it will take care of saving the picture after it has been captured. We will have to provide the storage path where the image will be saved. To save the POI images, we will use a naming scheme such as `poiimage<poi id>.jpg`.

Let's now extend the `POIService` class with the following additional methods.

**Implementing GetFileName()**

Let's implement the `GetFileName()` method in `POIService.cs` that will take care of providing the absolute path for saving the images in device memory. The absolute path includes the location and the filename. The image files will be named as `poiimage<poi id>.jpg`. The following listing shows how the filename can be constructed:

```csharp
public static string GetFileName(int poiId)
{
    String storagePath = System.IO.Path.Combine
        (Android.OS.Environment.ExternalStorageDirectory.Path, "POIApp");
    String path = System.IO.Path.Combine (storagePath, "poiimage" +
        poiId + ".jpg");
    return path;
}
```

**Implementing GetImage()**

As discussed in the previous sections, the saving of an image can be done by the camera intent. However, we need to write two helper methods to read and delete the image from device storage.

Let's add the `GetImage()` helper method that will be used to read the POI image from the device memory location, where it was saved earlier. Add the following listing to the `POIService.cs` class:

```csharp
public static Bitmap GetImage(int poiId)
{
    string filename = GetFileName (poiId);
    if (File.Exists (filename)) {
        Java.IO.File imageFile = new Java.IO.File (filename);
        return BitmapFactory.DecodeFile (imageFile.Path);
    }
    return null;
}
```
Implementing DeleteImage()

After a user deletes the poi object by clicking on the delete action bar button, currently the POI will be deleted from the server. It is a good idea to delete the corresponding POI image from device memory once the delete from the server side is successful.

The following helper method is used to delete the image based on its POI ID:

```csharp
public void DeleteImage (int poiId)
{
    String filePath = GetFileName (poiId);
    if (File.Exists (filePath)) {
        File.Delete (filePath);
    }
}
```

In this chapter, we are not dealing with the delete POI; however, to incorporate the changes as described earlier, you need to call the `DeleteImage()` method in `DeletePOIAsync()`:

```csharp
... if (response != null || response.IsSuccessStatusCode){
    DeleteImage (poi.Id.Value);
    string content = await response.Content.ReadAsStringAsync();
    return content;
} 
...```

Capturing an image from POIDetailFragment

We are now ready to take on the task of capturing a photo. This will involve the following tasks:

- Adding new user interface widgets to initiate capturing a photo and displaying it
- Building a photo intent to navigate to an external camera app to capture a photo
- Processing the results of the photo intent and displaying a photo once it is successfully captured

The following sections describe the details of each step.
Adding UI elements

There are a few new UI elements that we will need to add to support capturing an image; we need an ImageButton element to initiate the process of capturing an image, and we also need an ImageView element to display the captured POI image. The ImageButton widget can be added adjacent to the location and map buttons, and the ImageView element can be placed as the first widget above the Name field.

The following list shows the definition of the ImageView, which should be placed inside the LinearLayout before the POI name text field:

```xml
<LinearLayout
  ...
  <ImageView
    ...
    p1:layout_width="200dp"
    p1:id="@+id/poiImageView"
    p1:layout_gravity="center_horizontal"
    p1:scaleType="centerCrop"
    p1:layout_marginBottom="10dp" />
  <TextView
    ...
    p1:layout_width="fill_parent"
    p1:layout_height="wrap_content"
    p1:id="@+id/textView10" />

  ...
  </LinearLayout>
```

Create a private reference object in POIDetailFragment and assign the reference in OnCreateView():

```java
ImageView _poiImageView;
...
_poiImageView = view.FindViewById<ImageView>
(Resource.Id.poiImageView);
```

Now, we need a button that initiates the camera. We will start by copying the ic_new_picture.png icon from the assets folder to the project's drawable folder and adding it to the project in the same manner, as we did in the previous chapters.
Add the following button definition to the `LinearLayout` that contains the other two buttons:

```xml
<ImageButton
    android:src="@drawable/ic_new_picture"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/photoImageButton" />
```

The output of the preceding layout changes should look like the following screenshot:

![Screenshot of the modified layout](image)

Create a private reference object in `POIDetailFragment` and assign the reference in `OnCreateView()` as follows:

```csharp
ImageButton _photoImageButton;
...
    _photoImageButton = view.FindViewById<ImageButton>(Resource.Id.photoImageButton);
    _photoImageButton.Click += NewPhotoClicked;
```

Notice that in the preceding code snippets, we are assigning the `NewPhotoClicked()` event listener to the `photoImageButton` button. We will complete the `NewPhotoClicked()` method implementation in the following sections in this chapter.
Adding the Camera App Integration

Creating the camera Intent

To start an external camera app to capture a photo, we rely on the `Intent` class again, this time combined with an action. The following listing depicts creating an `Intent` class with the image capture action:

```java
Intent cameraIntent = new Intent(MediaStore.ActionImageCapture);
```

The `MediaStore` class contains the metadata for all the available media on both internal and external storage devices. The `MediaStore.ActionImageCapture` action tells the Android platform that you want to capture a photo and are willing to use any existing app that provides these capabilities.

Checking for registered camera apps

In Chapter 9, Making POIApp Location Aware, we used `PackageManager` to check whether there is a map app installed to handle our intent. We now need to perform the same check for an app that can handle our `ActionImageCapture` intent. The following listing shows the logic we need:

```java
PackageManager packageManager = PackageManager; 
IList<ResolveInfo> activities = packageManager.QueryIntentActivities(cameraIntent, 0); 
if (activities.Count == 0) {
    //display alert indicating there are no camera apps
} else {
    //launch the camera Intent
}
```

Providing additional information with the Intent

Before we start the intent, we need to provide some information to the camera app that processes our request; specifically, a filename and location, and the maximum size of the resulting photo. You must be careful about providing the value of the maximum size; this can be a potential threat that can cause out of memory exceptions. We do this by adding `Extras` to the intent. The `MediaStore` class defines a number of standard `Extras` that can be added to an intent to control how an external app fulfills the intent.
Providing a filename and location

The MediaStore.ExtraOutput extra can be used to control the filename and location that the external camera app should use in order to capture an image. We previously added the GetFileName() method in the POIService class to provide the file path information. However, the camera app expects the file path as an instance of Android.Net.Uri; and hence, we will need to convert the string path to an instance of Android.Net.Uri.

This is a two-step process. First, we create a Java.IO.File object using the string path from the data service and then create an Android.Net.Uri object. The following listing shows how to accomplish the construction of the URI and set up the MediaStore.ExtraOutput extra:

```java
Java.IO.File imageFile = new Java.IO.File(POIData.Service.GetFilename(_poi.Id.Value));
Android.Net.Uri imageUri = Android.Net.Uri.FromFile (imageFile);
cameraIntent.PutExtra (MediaStore.ExtraOutput, imageUri);
```

Providing a size limit

The MediaStore.ExtraSizeLimit extra limits the image size. It is much more straightforward to set up as shown here:

```java
cameraIntent.PutExtra (MediaStore.ExtraSizeLimit, 1.5 * 1024);
```

Starting the Intent

We are now ready to invoke the camera app by starting the intent by calling the StartActivity() method. Unlike the map app, in the case of a camera intent, we expect the results back from the activity. We are expecting the camera app to provide either a photo or a notification that the user cancelled to capture the photo. This can be accomplished by calling StartActivityForResult() by passing in the intent. The StartActivityForResult() method works in conjunction with an OnActivityResult() activity callback to communicate the results of the intent.

The following listing depicts the calling of StartActivityForResult():

```java
const int CAPTURE_PHOTO = 0;
...
StartActivityForResult(cameraIntent, CAPTURE_PHOTO);
```
Adding the Camera App Integration

Notice the second parameter of `StartActivityForResult()`. It is an integer value named `requestCode` that will be returned as a parameter to `OnActivityResult()` in the callback and help you identify the original reason for launching an intent. The best practice is to define a constant value to pass in for each `requestCode` that could potentially cause `OnActivityResult()` to be called.

Note that when we call the `StartActivityForResult()` method from the fragment, the result will always be returned to the `OnActivityResult()` method of the activity that hosts `POIDetailsFragment`. However, in this case, we are looking for the results in `POIDetailsFragment` itself. For this, we need to override the `OnActivityResult()` method in all of the hosting activities, and call `base.OnActivityResult` to fire the callback to the `OnActivityResult()` method declared on the fragment.

Completing the NewPhotoClicked() method

We have covered a number of topics related to starting the camera app in a somewhat fragmented fashion. The following listing is the complete implementation of `NewPhotoClicked()`:

```csharp
int CAPTURE_PHOTO = 100;

void NewPhotoClicked (object sender, EventArgs e)
{
    if (_poi.Id <= 0){
        Toast.MakeText (activity, "You must save the POI before attaching a photo.", ToastLength.Short).Show ();
        return;
    }

    Intent cameraIntent = new Intent (MediaStore.ActionImageCapture);
    PackageManager packageManager = Activity.PackageManager;
    IList<ResolveInfo> activities = packageManager.QueryIntentActivities (cameraIntent, 0);
    if (activities.Count == 0) {
        Toast.MakeText (activity, "No camera app available.", ToastLength.Short).Show ();
    } else {
        string path = POIService.GetFileName (_poi.Id);
        Java.IO.File imageFile = new Java.IO.File (path);
        Android.Net.Uri imageUri = Android.Net.Uri.FromFile (imageFile);
```
cameraIntent.PutExtra (MediaStore.ExtraOutput, imageUri);
cameraIntent.PutExtra (MediaStore.ExtraSizeLimit, 1 * 1024 * 1024);
StartActivityResult (cameraIntent, CAPTURE_PHOTO);
}

Processing the results of the Intent

The initiating activity is notified of the results of an intent via the OnActivityResult() callback method. Let’s add the following callback to both POIListActivity and POIDetailActivity to trigger the callback back to POIDetailFragment:

protected override void OnActivityResult (int requestCode, Result resultCode, Intent data)
{
    base.onActivityResult (requestCode, resultCode, data);
}

Now, let’s override the OnActivityResult method in the POIDetailFragment class. The following listing shows the signature of the OnActivityResult() method:

public override void OnActivityResult (int requestCode, Result resultCode, Intent data)

We discussed requestCode in the previous section. The resultCode parameter indicates the result of the intent that was launched and is of type Result, which can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESULT_OK</td>
<td>The activity completed the request successfully</td>
</tr>
<tr>
<td>REQUEST_CANCELED</td>
<td>The activity was canceled, generally, by a user action</td>
</tr>
<tr>
<td>REQUEST_FIRST_USER</td>
<td>The first value that can be used for a custom meaning</td>
</tr>
</tbody>
</table>

The third parameter, data, is of type Intent and can be used to pass additional information back from the activity that was launched. In our case, we are only concerned with requestCode and resultCode. The following listing shows the implementation of OnActivityResult() in POIDetailFragment:

public override void OnActivityResult (int requestCode, Result resultCode, Intent data)
{

Adding the Camera App Integration

```java
if (requestCode == CAPTURE_PHOTO) {
    if (resultCode == Result.Ok) {
        Bitmap bitmap = POIService.GetImage (_poi.Id.Value);
        _poiImageView.SetImageBitmap (bitmap);
        if (bitmap != null) {
            bitmap.Dispose ();
            bitmap = null;
        }
    } else {
        Toast.MakeText (Activity, "No picture captured.",
                        ToastLength.Short).show();
    }
} else {
    base.OnActivityResult (requestCode, resultCode, data);
}
```

Notice that when `resultCode` is `RESULT_OK`, we load the photo that was captured into a `Bitmap` object and then set the image for `_poiImageView`. This causes the image to be displayed at the top of the `POIDetailFragment` layout. If `resultCode` is not `RESULT_OK`, we display a toast message to the user, indicating that the action was canceled.

We are making use of `GetImage()` in `POIService` to retrieve the image from the memory. It is a simple utility method that accepts a POI ID and loads `Android.Graphics.Bitmap` using the `Android.BitmapFactory`.

So far, we have added a lot of code. Run `POIApp`, and from the `POIDetails` page, click on the add image button to invoke the camera. Notice that the captured image will be displayed on the `POIDetails` screen.

Upload an image using an HTTP multipart

The last functionality we are left to incorporate in the `POIApp` sample application is to upload the captured POI image to the server. Currently, we are saving only the POI details on Cloud, and the POI images are stored locally on the user's device memory. It would be nice to upload the image to the server so that we don't lose the images when a user deletes them locally from a device.

If the user has captured the photo of the POI and it is available locally on the device memory, then the save action will post the image along with the data using multipart form upload. Otherwise, it will post just the POI JSON data for the save operation.
Before we create a multipart form data request, let's first understand the web service API specification. The following block describes the web service API requirements for a multipart form upload:

Endpoint: com.packt.poiapp/api/poi/upload
Request-Type: multipart/form-data POST

-----BOUNDARY
Content-Disposition: form-data; name="file"; filename="poiimage1.jpg"
Content-Type: application/octet-stream

-----BOUNDARY
Content-Disposition: form-data; name="poi";
Content type = application/json, Encoding=UTF8
Body=
{"id":1,"latitude":"51.5033","longitude":"0.1197","name":"London Eye","description":...,"image":""}

-----BOUNDARY

Notice that the preceding API specification uses the multipart/form-data encoding type. Earlier in this book, for uploading the POI JSON object we used the HTTP POST method, but now we need to send the POI data along with the Bitmap stream, which cannot be uploaded with the POST method. We need to use multipart/form-data encoding type through which we can attach an image, along with several other message contents.

A standard delimiter called boundary separates each part of a multipart message. Each part of the message can define its own standard headers, such as Content-Type and Content-Disposition, providing the name of the file whose value it contains. The multipart delimiters and header fields are always 7-bit ASCII, in any case. The content of the image is present right below the Content-Type header, followed by the value of the POI JSON string.

For more detailed information on multipart Content-Type, search for RFC 1341(MIME) and visit http://www.w3.org/Protocols/rfc1341/7_2_Multipart.html.
Adding the Camera App Integration

Now that we understand the API specification to upload the image along with the POI data, let's proceed with the implementation:

1. Create an overloaded method for `CreateOrUpdatePOIAsync()` in the `POIService` class that takes two arguments: object of `PointOfInterest`, and the POI image bitmap:

   ```csharp
   public async Task<string> CreateOrUpdatePOIAsync(PointOfInterest poi, Bitmap bitmap)
   {
       ...
   }
   ```

2. Add the following snippets to convert the POI object to a JSON object and then to `StringContent`. I don't need to give you an explanation as we have already done a similar thing while posting the POI data:

   ```csharp
   var settings = new JsonSerializerSettings();
   settings.ContractResolver = new POIContractResolver();
   var poiJson = JsonConvert.SerializeObject(poi, Formatting.None, settings);
   var stringContent = new StringContent(poiJson);
   ```

3. Now convert the Bitmap image to a byte array in order to send it as `ByteArrayContent` via the multipart form upload:

   ```csharp
   byte[] bitmapData;
   var stream = new MemoryStream();
   bitmap.Compress(Bitmap.CompressFormat.Jpeg, 0, stream);
   bitmapData = stream.ToArray();
   var fileContent = new ByteArrayContent(bitmapData);
   ```

   Notice that the `bitmap.Compress()` writes a compressed version of the Bitmap to the specified stream. The second integer argument indicates the compression quality ranging from 0 to 100, where 0 means low quality and 100 being the maximum (lowest quality).

4. Add the following additional content headers such as media content type and content disposition. The POI image name is sent in the `poiimage<poid>.jpg` format. The server that saves the image in the database will use this name:

   ```csharp
   fileContent.Headers.ContentType = MediaTypeHeaderValue.Parse("application/octet-stream");
   fileContent.Headers.ContentDisposition = new ContentDispositionHeaderValue("form-data")
   ```
5. Now that we are ready with both the POI data and and image content, we can add both the chunks to the MultipartFormDataContent. The boundary is a random string used as a delimiter to separate each part of the message body:

```csharp
string boundary = "---8d0f01e6b3b5daf";
MultipartFormDataContent multipartContent = new MultipartFormDataContent(boundary);
multipartContent.Add(fileContent);
multipartContent.Add(stringContent, "poi");
```

6. Declare a string constant that represents the web service endpoint to upload the POI image using multipart:

```csharp
```

7. Let's now proceed to upload the form content to the server using the `HttpClient` class in the `PostAsync()` method. The following code snippet demonstrates posting data to the server using the `HttpClient` class:

```csharp
HttpClient httpClient = new HttpClient();
HttpResponseMessage response = await httpClient.PostAsync(UPLOAD_POI, multipartContent);
if (response.IsSuccessStatusCode) {
    string content = await response.Content.ReadAsStringAsync();
    return content;
}
return null;
```

The result of the upload request is sent back to the `POIDetailFragment`.

Now that we have the method ready to upload the image along with the POI JSON data, let's change the logic in the `POIDetailFragment` class in the `SavePOI()` method. Currently, the save action validates the user input and calls `CreateOrUpdatePOIAsync()` by passing the POI object.
Adding the Camera App Integration

Now if the user has captured the POI image using the camera, let's instead call the newly created overridden version of `CreateOrUpdatePOIAsync()` by passing the Bitmap, POI, and activity instance. Add the following logic to the `SavePOI()` method:

```csharp
POIService service = new POIService();
Bitmap bitmap = null;
if (_poi.Id > 0) {
    bitmap = POIService.GetImage (_poi.Id);
}

string response;
if (bitmap != null) {
    response = service.CreateOrUpdatePOIAsync (_poi, bitmap);
} else {
    response = service.CreateOrUpdatePOIAsync (_poi);
}
```

To make our application memory efficient, it is a good idea to dispose the Bitmap instance after you're done with it. Add the following code to dispose the Bitmap memory:

```csharp
if (bitmap != null) {
    bitmap.Dispose();
    bitmap = null;
}
```

We have now completed the `POIApp` app that exercises many of the Xamarin.Android features you will need to utilize for developing a professional app. While the app is relatively simple in nature, we hope that it provides a good starter guide for the Android app development. I wish you good luck with the Xamarin.Android development! Happy coding!

**Summary**

In this chapter, we completed `POIApp` by adding integration with the camera. We now have an app that demonstrates many of the features of the Android platform.

In the last chapter, we will discuss the various app distribution channels and steps to prepare your application for deployment.
An app is useful if everyone can enjoy it, and this means finding a way to make it available to the masses. In this chapter, we will take a look at the various aspects of getting your app ready for deployment and discuss the various options available for publishing the Android application to market. In this chapter, we will cover the following topics:

- Preparing application for release
- Publishing a signed APK for uploading
- Different app distribution options
- Publishing on Google Play

Preparing the application for release
As your application is fully developed, the moment when you put a significant amount of effort to verify each of the key functional modules is crucial. Many developers prefer to test the app using different test automation frameworks (such as Robotium, Appium, Xamarin Test Cloud, and so on), or some developers use manual testing to verify the app on targeted devices. Once you're convinced that the application doesn't contain any obvious glitches and works as smoothly as expected, you're ready to deploy the app to the app store.

The following sections discuss the various aspects to be considered prior to producing a release APK.
Disabling the debug mode

During the development of an application, Xamarin Studio supports debugging Xamarin.Android apps using **Java Debug Wire Protocol (JDWP)**. This is great for development purposes but poses security risks for deployed applications and thus needs to be disabled in the released apps. There are two different ways to accomplish this:

2. Using the `AssemblyInfo.cs` conditional directives.

To remove the debug mode using the Android `AndroidManifest.xml` descriptor, you need to set the value of the `android:debuggable` property to `false`. The following listing shows how to turn off the JDWP debugging from the manifest file:

```xml
<application ...
  android:debuggable="false" ...
</application>
```

Another way of disabling the debug mode is using the conditional directive in the `AssemblyInfo.cs` file. The following listing shows how to use a conditional directive to turn the JDWP debugging off or on based on the configuration that is selected. This approach has the advantage of being based on the currently selected configuration:

```csharp
#if RELEASE
  [assembly: Application(Debuggable=false)]
#else
  [assembly: Application(Debuggable=true)]
#endif
```

Configuring linking options

By default, the release mode turns off using shared runtime and turns on linking so that your distribution APK only contains the portions of the Xamarin.Android runtime required by your app. The linker does this by performing a static analysis of your compiled code in order to determine which assemblies, types, and type members are used by your application. All unused assemblies, types, and members are striped out in order to reduce the overall size of the application.


**Linker Options** can be viewed and set in the **Project Options** dialog under the **Android Build** section:

When viewing and adjusting **Linker Options**, make sure to first select **Release** from the **Configuration** drop-down box. Xamarin.Android provides the following linking behaviors:

- **Don't link**: This turns off the linker; no linking will be performed.
- **Link SDK assemblies only**: This will only link the assemblies that are required by Xamarin.Android. Other assemblies will not be linked; they will be distributed as separate assemblies.
- **Link all assemblies**: This will link all assemblies that are required by the application and not just the ones required by Xamarin.Android.
Side effects of linking

In some cases, linking can have some unintended side effects, including needed types and members being accidentally discarded. It is very important for an application that is compiled and linked in a release mode to be put through a thorough testing cycle in order to ensure that the app does not suffer from this side effect. In fact, in most cases, testing beyond the initial developer's testing should be conducted using an APK file and produced in the release mode.

If you encounter runtime exceptions or build failure issues related to missing types or have trouble locating specific methods, you may need to provide a custom linker file that gives explicit instructions concerning specific types or members to the linker.

The following listing is an example of a custom linking file that directs the linker to always include a specific type and specific set of members for the type:

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<linker>
  <assembly fullname="Mono.Android">
    <type fullname="Android.Widget.AdapterView">
      <method name="GetGetAdapterHandler" />
      <method name="GetSetAdapter_Landroid_widget_Adapter_Handler" />
    </type>
  </assembly>
</linker>
```

A custom linking file can be added to a project as a simple XML file. After adding the file to the project, select the file, open the Properties pad, and choose LinkDescription from the Build action menu, as shown in the following screenshot:
Selecting supported ABIs

Android supports several different CPU architectures. The Android platform defines a set of Application Binary Interfaces (ABIs) that correspond to different CPU architectures. By default, Xamarin.Android assumes that `armeabi-v7a` is appropriate in most circumstances. If you need to support additional architectures, then you must check each one that applies. This will cause the build process to generate the code that will run on all the target ABIs as well as include native libraries that are appropriate for each architecture.

Supported ABIs can be specified in the Project Options dialog under the Android Build section:

One scenario where I have run into the need of specifying additional ABIs is the testing process. I have worked with a group of testers many times; some of which have physical devices and some of which use emulators. To support the use of the x86 emulator, you will need to include x86 in the supported ABI's list.
Verifying the package name, icon, and app version

There are several additional manifest properties that you need to verify before preparing a release build for uploading it to the Android market. This section will take you through some of those key items.

The package name

A package name is a unique identifier for your application in Google Play. An application package name is defined in the `AndroidManifest.xml` file. Once your application is deployed to the market with this name, you cannot change it later. Changing the app package name means that it will be treated as a completely new application.

The Android application package name should be unique and the name may contain the uppercase or lowercase letters ('A' through 'Z'), numbers, and underscores ('_'). Usually, it is a best practice to use your Internet domain ownership as the basis for your package names. For example, applications published by Yahoo will start with `com.yahoo.<app id>`.

Most of the app settings can be either controlled directly by editing the `AndroidManifest.xml` file or from the project Options window. To modify the package name in the `AndroidManifest.xml`, you need to use the `package` attribute, as shown in the following code snippet:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.packt.poiapp">
  ...
</manifest>
```
Right-click on your project in the Xamarin Solution Explorer and select **Options** to open the **Project Options** window. The following screenshot depicts the project options window in Xamarin Studio:

**The application name and icon**
Making an app that works in a device isn't just enough. You will need to provide a high-quality app icon that represents your brand or product. After all, the app icon is the first thing that a user will notice after the app is installed on the device. You must prepare your app launcher icon for devices of mdpi, hdpi, xhdpi, and xxhdpi density.
Once the launcher icons are ready, add them to their respective drawable resource folder for different screen densities. You can set the app name by setting the `android:label` property to the `<application>` element in the `AndroidManifest.xml` file, and for setting the application icon, you can use the `android:icon` property:

```xml
<manifest ......>
  <application
    android:label="POIApp"
    android:icon="@drawable/ic_launcher">
  </application>
</manifest>
```


### The application versioning scheme

The application versioning scheme is used to track different build versions. This also helps notify a user when there is an upgrade version of the application available for download. Android uses the following two different manifest attributes to define the build version information:

- **android:versionCode**: This is an integer value that represents the current build version of the application. Internally, Google Play uses this property to handle the build update process such as notifying the user when a new version of the app is available. You must increase the value of `versionCode` for each successive release.

- **android:versionName**: This is a string that represents the release version of the application. This version code will be displayed publicly to the user on Google Play. Note that this string is not made mandatory on Google Play and is not used internally, but it is just displayed it to the user.

Both of these attributes can be set either from the **Project Options** window or from the application manifest file:

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="1"
  android:versionName="1.0"
  package="com.packt.POIApp">
  ....
  ....
</manifest>
```
Review user permissions

The Android application must declare the list of permissions it requires for accessing certain features in the app. Android app permissions are declared in the AndroidManifest.xml file using the <uses-permission> tag. When an application is being installed on the user device, the Android system reads the manifest file and displays the list to the user. The user must decide to allow the permissions for the installation to complete. The following screenshot depicts the app permissions dialog while downloading the Facebook mobile app:

While preparing a release version build of your application, verify that your app manifest file defines only the permissions that are required for your application. For example, a camera application that requests for the read call log permission could be a reason for annoyance with a high degree of a chance for the user to not download your application.
Publishing a signed APK for uploading

Once we have completed all the previous steps, we will proceed to the final step of preparing a signed APK to be published on different app stores. The following sections discuss the steps of producing a signed APK from within Xamarin Studio.

Android keystores

A keystore is a database of security certificates created and managed by the keytool program from the Java SDK. The keystore is an important aspect of creating a release version of the Android app. The Android device cannot run applications that have not been digitally signed. This may come as some surprise because we have been running our app for some time now. During the development process, Xamarin. Android uses a debug keystore that signs the build while running the application from an IDE, and as such the apps to run on the device. This keystore works for debugging purposes only, but will not be recognized as a valid keystore for the distribution of released apps.

A keystore must be kept safe as the same keystore key is required for publishing all of the future versions of an application. If a keystore is lost, it will not be possible to publish updates to an app on Google Play. The only solution would be to create a new keystore and publish the new version as a completely new app.

A keystore can be created using a keytool and jarsigner command-line tool available in Java JDK. The following command is used to create a keystore using the Java keytool utility:

```
keytool -genkey -v -keyalg RSA -keysize 2048 -alias MyKeyPair -validity 10000
Enter key password:
Re-enter new password:
What is your first and last name? [MyName]: Neel Lakshman
What is the name of your organizational unit? [MyOrgName]: Packt
What is the name of your company or organization? [MyCompany]: Packt
What is the name of your city or locality? [MyCity]: London
What is the name of your state or province? [MyProvince]: UK
What is the two-letter country code for this unit? [MyCountry]: UK
Is CN=MyName, OU=MyOrgName, O=MyCompany, L=London, ST=MyProvince, C=MyCountry correct? [yes]: yes
Generating 2048 bit RSA key pair and self-signed certificate (SHA1withRSA) with a validity of 10,000 days for: CN=Neel Lakshman, OU=Packt, O=Packt, L=London, ST=UK, C=UK
Enter key password for <MyKeyPair>
```

[252]
Note that the preceding command prompts the user to provide the password and other required details in order to generate the `release.keystore` key.

While the command-line keytool utility can be used directly to create and manage keystores, Xamarin.Android provides a user interface to the tool, which is integrated into the publishing process. The following section will guide you with the steps to create a keystore and prepare a signed Android build for publishing.

**Publishing from Xamarin.Android**

The following steps will guide you through creating a new keystore as part of the process while creating a signed `.apk` file:

1. In the Run Configuration drop-down box, select the Release option:

![Screenshot of Xamarin Studio showing Run Configuration]

2. Navigate to the Build | Achieve for Publishing option in the Xamarin Studio toolbar. You will see a list of the latest POIApp builds, as shown in the following screenshot. If you have never build your code, it will build and generate one for you:

![Screenshot of Xamarin Studio showing build history]
Publishing an App to the App Store

3. Select the latest build from the list and click on the Sign and Distribute... button from the bottom-right corner. You will see two options: Ad-Hoc or Google Play. If you select the Ad-Hoc option, it will create a build and save it in your disk. You need to manually publish the app to Google Play. However, the Google Play option allows you to distribute your app on Google Play directly from Xamarin Studio. However, it is always a good idea to create a signed build and save it in your computer so that you can verify all the features before publishing it to Google Play:

4. Let's select Ad-Hoc and click on Next. Notice that it will open the Android Signing Identity wizard with two buttons: Create a New Key and Import an Existing Key. If you have created the keystore using the Java keytool command-line utility as explained earlier, you will need to directly select Import Existing Key. In this example, I have selected Create a New Key button to demonstrate how Xamarin Studio facilitates the process of creating a new keystore.

5. Select Create a New Key button.
6. Fill in all of the required details, such as alias, password, full name, validity, and so on, and then select **OK** to confirm:

![Create New Certificate dialog](image)

7. It will create a new key and display the **Android Signing Identity** dialog. Select the key and click on **Next**.

8. Notice that it will display a confirmation dialog, as shown in the following screenshot:

![Publish as Ad Hoc dialog](image)
9. Select **Publish**. It will prompt you to select the location to save the apk file. As prompted, you need to provide the keystore password while generating the signed APK:

The resulting APK is ready for final testing and potential distribution.

**Repubhlishing the app**

The keystore that was created while deploying the very first version of an application is very important and should be kept safe. The keystore, alias, and passwords should be kept secure so that only those authorized to publish a new version of the app will have access to them.
Different app distribution options
Android developers have a number of options for distributing their applications, which include the following:

- Self-publishing via a private Cloud or via website links
- Distributing via e-mail attachments
- Publishing in Google Play
- Publishing in third-party app stores

Self-publishing via website links or e-mails
Website links and e-mail attachments are pretty straightforward, easy to accomplish, and may be suitable for some apps that are used internally by a company or by a small group of associates.

Prior to installing apps from a website link or an e-mail attachment, you must first update your device security settings to allow apps to be installed from Unknown sources. Let's take a look at the following screenshot:

After enabling this option, you will be prompted to install an app when selecting an e-mail attachment or a web link that is an APK.
Note that for general consumers, website links and e-mails as a means of distribution are not ideal. Marketplaces such as Google Play and Amazon Appstore provide significant advantages over self-publishing methods. The following are some of the advantages of distributing your application via an app store:

- Most app stores provide a review process to ensure that apps are not malicious in nature. Google has recently introduced an app store review process that verifies the violation of the Android developer policy, build size, permissions, and so on. The review process strengthens the Android community and helps to build consumers’ trust.
- Provides a robust infrastructure to reach out to millions of consumers by distributing apps on App Store.
- Promotes apps and allow content rating from consumers.
- Handles financial settlements for applications that use subscriptions or purchases.

**Publishing on Google Play**

Google Play is an ideal marketplace to upload and distribute your application to a wider range of audience. It is a single platform that allows an app publisher to distribute, advertise, make dollars by selling your app, and analyze the usage and stats of your application. On the other hand, every Android device is shipped with a Google Play Android app through which your published application will be discovered and downloaded to users.

Before your application is pushed into the wild, it is always valuable to get some real-world feedback from users. Developers prefer to distribute an alpha/beta version of their application to a group of users and gather feedback. Beta testing helps you get the early feedback from real users and address them before heading to the production move.

Google Play allows you to set up and distribute the alpha beta staging release of your app. After the release version of your application is prepared and uploaded to the beta stage, you can invite a group of testers to whom you would like to distribute your application. The testers with an invitation can download the app, provide the feedback, and rate your application. This book does not include the detailed steps to use the Google Play beta testing program. For more details, you can visit http://developer.android.com/distribute/tools/launch-checklist.html.
To publish your application to Google Play, you need to follow some of the specific steps described as follows:

1. To distribute apps on Google Play, you will require a developer account. To register as a developer, there is a one time registration fee of $25 USD. You can follow the on-screen instructions to register as a developer, which is available at https://play.google.com/apps/publish/signup/.

2. Prepare the promotional assets such as graphics, video, and promotional messages required while uploading your application. The following is the list of:
   - Prepare the promotional messages, including an application title, a short description, and a full description
   - High-resolution application icons of size 512 w x 512 h
   - Feature graphic of size 1024 w x 500 h
   - Promo graphic of size 180 w x 120 h
   - An optional promo YouTube video showcasing your application

3. Once you are ready with the release version of APK and the promotional contents, the application can be uploaded to Google Play. This is done by logging to the Google Play publisher dashboard:
4. Click on the **Add new application** button to help you with the process of uploading an APK:

![Add New Application](image)

5. Provide a **Title** for your app to be displayed on Google Play and click on **Upload APK**.

6. Once the APK is uploaded, you need to complete the **Store listing** details by uploading the required screenshots and other promotional materials.

7. While uploading, all the apps must be rated according to the Google Play ratings system. To learn more about the new content rating system, you can refer to [https://support.google.com/googleplay/android-developer/answer/188189](https://support.google.com/googleplay/android-developer/answer/188189).

8. The **Pricing and Distribution** tab on the dashboard allows you to select the geo location for your app availability.

9. Once you furnish all the particulars required, you can now click on the **Publish app** button to make your application live for download.

---

**Third-party app stores**

Google Play is not the only option available to distribute your application to the world. You should consider using other distribution channels such as Mall.Soc.io ([http://soc.io/apps](http://soc.io/apps)), GetJar ([http://www.getjar.com/](http://www.getjar.com/)), Amazon Appstore, and so on. There is no restriction imposed on Google for redistributing the same application on other marketplaces. Consider all the available options to make the most of your efforts.
Summary

In this chapter, we briefly discussed the process to prepare and sign the release version of an APK. We also discussed the different app distribution channels that are available to developers for distributing the Android apps.

We have now completed the POIApp app that exercises many of the Xamarin. Android features you will need while developing a professional Android application. However, there are tons of great features in Android that remained undiscovered in this book, which include audio, video, face recognition, Bluetooth, NFC, and so on.

In the context of this starter guide, we simply do not have enough space to cover all features that are supported in Android. Make sure that you explore the http://developer.android.com/index.html and http://developer.xamarin.com/ to enhance your knowledge on the other interesting Android features.

I wish you good luck with the Xamain.Android development! Happy coding!
Index

A

ActionBar
   actions, adding to 102
activity
   about 12
   events 14
   life cycle 13
ActivityManager 10
activity state, for configuration changes
   POI list scroll position, retaining 144, 145
   saving 142, 143
   saving manually 143, 144
adapter layouts
   about 22
   GridView 22
   ListView 22
Ahead-of-Time (AOT) 10
Android Application Framework (AAF) 10
Android application orientation
   activity state, saving for configuration changes 142, 143
   Android behavior, on configuration changes 139, 140
   dynamically requesting 141
   locking 140, 141
   orientation-aware layout, building 145-147
   orientation behavior, handling
      manually 148, 149
Android bindings design
   about 33
   constants to enumerations 34
   C# properties 33
   delegates 34
   principles 33

Android Cloud to Device Messaging (C2DM) service 6
Android device
   apps, running on 67
   debugging with 66
   peer object 67, 68
   USB debugging, enabling 67
   USB driver, installing 67
Android Device Monitor (ADM) 217
Android emulator
   speeding up 50
   third-party emulator solutions 50
   x86 emulator, using 50
Android.Locations.Geocoder class 219
AndroidManifest.xml file 68, 69
Android packages (.apk) 11
Android platform
   about 2
   Android runtime 9
   Linux kernel 9
   native libraries 9
   versions 2-8
Android Runtime (ART)
   about 31
   URL 10
Android SDK Build-tools 45
Android SDK Platform-tools 45
Android SDK Tools 45
Android tablet emulator
   creating 152-154
Android tablets 152
android:verisonName 250
android:versionCode 250
Android Virtual Device (AVD)
   about 46, 47
   creating 47-49
app distribution options
about 257
publishing in Google Play 258-260
self-publishing, via website links or
  e-mail 257, 258
  third-party app stores 260
app icon
setting 62, 63
Application Binary Interfaces (ABIs) 247
application framework
about 10
application layer 11
application manifest 12
application, preparing for release
about 243
application icon, verifying 249
application name, verifying 249
application versioning scheme,
  verifying 250
debug mode, disabling 244
linking options, configuring 244, 245
linking, side effects 246
package name, verifying 248
review user permissions 251
supported ABIs, selecting 247
app, publishing to app store
about 243
app distribution options 257
application, preparing for release 243
signed APK, publishing for uploading 252
async await 131
asynchronous programming
with async keyword 98, 99
with await keyword 98, 99
attributes, SQLite.NET ORM
AutoIncrement 193
Column 193
Ignore 193
Indexed 193
MaxLength 193
NotNull 193
PrimaryKey 193
Table 193
Unique 193
B
building blocks, Android application
about 11
activities 12
adapter layouts 22
Android packages (.apk) 11
application manifest 12
broadcast receivers 18
common layouts 20, 21
content providers 18
declarative, versus programmatic
  View creation 19
fragments 15
  intents 24
resources 24
R.java file 24
services 18
user interface widgets 19
ViewGroups 19
Views 19
XML layouts, using from activities 24
XML layout files 22
C
caching logic
  implementing, to POIApp 204-206
camera app integration
  adding 227
data service, extending 230
  emulator, configuring 229
  features 228
  image, capturing from
    POIDetailFragment 231
  image uploading, HTTP multipart
    used 238-242
  integration approach, selecting 227
  permissions 228
Common Language Runtime (CLR) 30, 98
collection changes
  handling, with fragments 185, 186
considerations, for designing Android app user interfaces
resolution 152
screen density 152
screen orientation 152
screen size 152
content providers 10, 18
C# properties 33
CreateOrUpdatePOIAsync method
   adding, to POIService 126-128
CreatePOI test 200

d
Dalvik Virtual Machine (Dalvik VM) 9
data
receiving, in POIDetailActivity 120
database helper class
creating 194, 195
POI data, deleting from database 197
POI details, reading from database 196
POI record, creating 196
POI record, updating 196
data service, extending
DeleteImage(), implementing 231
GetFileName(), implementing 230
GetImage(), implementing 230
data storage, in Android
about 190
external file storage 190
internal storage 190
shared preferences 190, 191
SQLite storage 192
default layouts, ListView widget
ActivityListItem 76
SimpleListItem1 76
SimpleListItem2 76
TwoLineListItem 76
delete action
adding 123
adding, in POIService 125
disabling 124
delete confirmation prompt
adding 136, 137
DeleteImage() method
   implementing 231
DeletePOIAsync method
   adding, to POIService 129
DeletePOI() method
   creating 132, 133
DeletePOI test 201
deserializing
   with Json.NET 100
development environments
   about 35
   compatibility 38
   IDE comparison 37
   Xamarin, for Visual Studio 36
   Xamarin Studio 35
DialogFragment
   working with 180-183
e
EditText.Error property
   using 134, 135
EditText‘s InputType
   working with 114
emulator
   configuring, for camera 229
f
features, Xamarin for Visual Studio
   IntelliSense 36
   Visual Designer 36
fragment life cycle events
onActivityCreated method 17
onAttach method 17
onCreate event 17
onCreateView event 17
onDestroy method 18
onDestroyView method 18
onDetact method 18
onInflate event 17
onPause method 17
onResume method 17
onStart method 17
onStop method 17
fragments
  about 15, 16, 155
  creating 155-157
  creating, to display POI list 167-170
  for older Android devices 184, 185
  life cycle 16, 17
  using, for displaying POIDetails 159
fragment subclasses
  DialogFragment 155
  ListFragment 155
  PreferenceFragment 155

G
Geocoding 218
GetFileName() method
  implementing 230
GetImage() method
  implementing 230
GetJar
  URL 260
Google app icon design guidelines
  URL 250
Google Play
  about 258
  publishing on 258

I
IDE
  cons 37, 38
  pros 37, 38
ILocationListener
  implementing 212
image, capturing from POIDetailFragment
  about 231
  additional information, providing with
    Intent 234
  camera Intent, creating 234
  filename, providing 235
  Intent results, processing 237, 238
  Intent, starting 235
  location, providing 235
  NewPhotoClicked() method,
    completing 236
registered camera apps, checking for 234
  size limit, providing 235
UI elements, adding 232, 233
Internet permissions
  adding 91, 92

J
Java Debug Wire Protocol (JDWP) 244
Java Native Interface (JNI) 31
Json.NET
  about 100
  used, for deserializing 100, 101
  used, for serializing 100, 101
jrsigner command-line tool 252
Json.NET component 119
Just in Time (JIT) 9

K
keytool 252
key/value pairs 119

L
launcher activity 63, 64
layout managers
  linear layout 21
  relative layout 21
  table layout 21
Linux kernel 9
ListFragment
  about 166
  key concepts 166
  POIListFragment layout, creating 166, 167
  using, for displaying POI list 166
list thumbnail image
  populating 89
ListView click event
  handling 105, 106
ListView item
  populating 85
ListView widget
  default layouts 76
location change notifications
requesting 211
LocationManager 11
location services
adding, to POIApp 212
adding, to POI details 215
adding, to POI list 212-214
address for location, obtaining 218, 219
app permissions, setting 209, 210
code, adding 216, 217
debugger, configuring 210, 211
debugging, settings 210
location data, mocking for testing 217, 218
LocationManager class, obtaining 211
user interface, updating 215
users, keeping informed 220, 221
working with 207, 208

M
Mall.Soc.io
URL 260
Managed Callable Wrappers (MCW) 31
Map app
navigating to 222, 223
registered map apps, checking for 224
map integration
adding 222
menus
setting, in OnCreateOptionsMenu() 104
menu XML file
defining 103, 104
Model-View-Controller (MVC) pattern 12
Mono
about 30
Dalvik, working with 30, 31
multi-pane layout
creating, for tablets 173-176

N
navigation
adding, to POIDetailActivity 117
Near Field Communication (NFC) 6
no network condition
handling 106, 107
NotificationManager 11
NUnit 198
NUnitLite
about 198
CreatePOI test 200
DeletePOI test 201, 202
setting up, for tests 199
test methods, creating 200
tests, executing 202, 203
using 198

O
onActivityCreated method 17
onAttach method 17
onCreate event 17
OnCreateOptionsMenu() method
menus, setting in 104
onCreateView event 17
onDestroy method 18
onDestroyView method 18
onDetach method 18
onInflate event 17
OnOptionsItemSelected() method
selection, handling in 104, 105
onPause method 17
onResume method 17
onStart method 17
onStop method 17

P
PackageManager 11
package name
about 248
setting 62, 63
peer objects 31
platforms
samples for SDK 46
SDK Platform 46
system images 46
platforms and tools, Android
Android platform APIs 45
Extras section 46
installing 43
Tools section 44, 45
POI
creating, web service used 125
deleting, web service used 128
updating, web service used 125

POIApp
caching logic, implementing 204-206
creating 52-55
debugging 64-66
defining 39
define class, creating 84, 85
features 39
location services, adding to 212
making compatible, with
Android tablets 157, 158
running 64-66
updating, to support multi-pane
split layout 177-179

POIList item click
navigating on 118-120

POIDetailActivity
creating 117
data, receiving in 120
navigation, adding to 117
refactoring, to add
POIDetailFragment 164, 165

POIDetailFragment
adding, to POIDetailActivity 163
creating 159-161
delete action, working with 161, 162
save action, working with 161, 162

POIDetail layout
creating 109, 110
ScrollView, adding 111-113

POI details activity layout
modifying 163

POIDetailsFragment layout
creating 159

POIListActivity
creating 75, 76
refreshing 137, 138
update, 101, 102

POI list activity layout
modifying 171

POIListFragment
adding, to POIListActivity 171-173

POI list row layout
creating 76, 77
distance TextView, adding 82, 83
Image widget, adding 79
LinearLayout widget, adding 80
name and address TextView classes,
adding 80, 81
RelativeLayout view group, adding 78, 79

POIListViewAdapter
constructor, implementing 86
Count { get }, implementing 87
creating 86
GetItemld(), implementing 87
GetView(), implementing 88
hooking up 92-94
index getter method, implementing 87
row Views, populating 88
row Views, reusing 88

POI ListView layout
creating 71-74

Points Of Interest app. See POIApp

POIService
CreateOrUpdatePOIAAsync method,
adding 126-128
delete action, adding 125
DeletePOIAAsync method, adding 129
save action, adding 125

POIService class
creating 97

POI web service
deploying 95

Project Options view
about 57
project structure, defining 57-59

R
release.keystore key 253

Resources Manager 11
Resources directory
defining 59
REST 95
REST web services
consuming, asynchronously 96

RFC 1341(MIME)
URL 239
R.java file 24
row Views
  populating 88
  reusing 88

S
save action
  adding 123
  adding, in POIService 125
SavePOI() method
  creating 130, 131
scale-independent Pixels (sp) 81
ScrollView
  about 110
  adding 111-113
  features 110
selection
  handling, in OnOptionsItemSelected() method 104, 105
serializing
  with Json.NET 100
services 18
shared preferences
  about 190
  using 191
signed APK, publishing
  Android keystores 252
  app, republishing 256
  from Xamarin.Android 253-256
SOAP 95
SQLite
  about 191
  advantages 191, 192
SQLite database storage 191
SQLite.Net ORM component
  about 192
  attributes, used for marking POIApp to be persisted 193, 194
  database helper class, adding 194, 195
  using 192
  ways, for adding to Xamarin application 192
SQLite Version 3.4.0 191

T
TableLayout manager
  using 113
TelephonyManager 11
test-driven development (TDD) 197
third-party emulator solutions
  Genymotion 51
  Xamarin Android Player 50, 51
Toast
  about 108
  creating 108
tools, Android platform
  Android Support Library 46
  Google Play services 47
  Google USB Driver 47

U
Uniform Resource Identifier (URI) 130
UrlImageViewHelper
  using 89, 90
USB driver
  installing 67
  URL 67
user interfaces, Android applications
  considerations, for designing 152
user interface widgets
  populating 121, 122

V
validation
  adding 133
variables
  binding, to controls 121
ViewGroup 15
View system 10
virtual device
  cloning 49
web service
  about 95
  consuming 94
  POIService class, creating 97
  POI web service, deploying 95
  REST web services, consuming 96
WindowManager 10

X

Xamarin
  about 27
  defining 27, 28
  Development IDE 28
  URL 40
  Xamarin platform 27
  Xamarin Test Cloud 28
Xamarin.Android
  about 29
  benefits 29
  drawbacks 29
  installing 40-43
  URL, for documentation 141
Xamarin.Android application packaging 32
Xamarin.Android NUnitLite. See NUnitLite

Xamarin platform
  Xamarin.Android 28
  Xamarin.Forms 28
  Xamarin.iOS 28
  Xamarin.Mac 28
  Xamarin.Windows 28
Xamarin Studio 35
Xamarin Studio IDE 56
Xamarin Studio layout designer
  about 59
  designer content layout 60
  Document Outline pad 61
  Properties pad 61
  target framework, setting 61, 62
  Toolbox pad 61
XML layout files
  about 22
  element and attribute names 23
  View and layout identifier 23
Thank you for buying
Xamarin Mobile Application Development for Android
Second Edition

About Packt Publishing
Packt, pronounced 'packed', published its first book, Mastering phpMyAdmin for Effective MySQL Management, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike.
For more information, please visit our website at www.packtpub.com.

About Packt Open Source
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt
We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.
Xamarin Cross-platform Application Development

Second Edition


Develop production-ready applications for iOS and Android using Xamarin

1. Write native iOS and Android applications with Xamarin.iOS and Xamarin.Android respectively.
2. Learn strategies that allow you to share code between iOS and Android.
3. Design user interfaces that can be shared across Android, iOS, and Windows Phone using Xamarin.Forms.

iOS Development with Xamarin Cookbook


Over 100 exciting recipes to help you develop iOS applications with Xamarin

1. Explore the new features of Xamarin and learn how to use them.
2. Step-by-step recipes give you everything you need to get developing with Xamarin.
3. Full of useful tips and best practices on creating iOS applications.

Please check www.PacktPub.com for information on our titles
**Xamarin Mobile Application Development for iOS**  
ISBN: 978-1-78355-918-3  Paperback: 222 pages

If you know C# and have an iOS device, learn to use one language for multiple devices with Xamarin

1. A clear and concise look at how to create your own apps building on what you already know of C#.
2. Create advanced and elegant apps by yourself.
3. Ensure that the majority of your code can also be used with Android and Windows Mobile 8 devices.

---

**Learning Xamarin Studio**  

Learn how to build high-performance native applications using the power of Xamarin Studio

1. Get a full introduction to the key features and components of the Xamarin 3 IDE and framework, including Xamarin.Forms and iOS visual designer.
2. Install, integrate and utilise Xamarin Studio with the tools required for building amazing cross-platform applications for iOS and Android.
3. Create, test, and deploy apps for your business and for the app store.

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles