Applications with Multiple Screens

Xamarin Android Level 1, Chapter 2

Xamarin Inc.

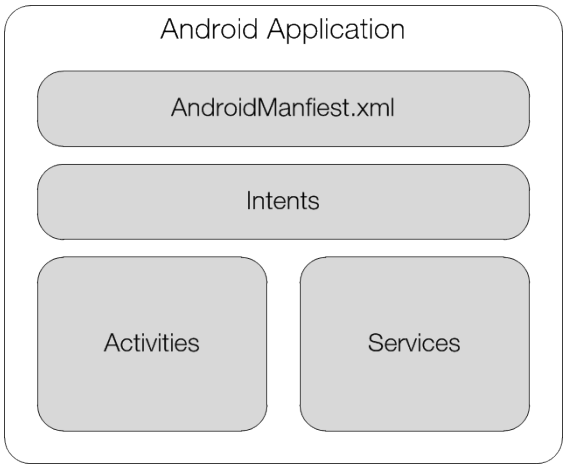
Overview

In this chapter we'll look at how to create applications that have more than one Screen using Xamarin.Android and walk through the creation of a simple multiscreen app. We'll introduce Intents and show how they can be used to load additional Activities. However, before we dive into creating the application, let's examine the constituent pieces of an Android application

Anatomy of an Android Application

Android applications are very different from traditional client applications found on platforms such as Windows and Mac OSX, or even those that run on mobile platforms such as iOS. These platforms have a single-entry point into the application which creates an instance of an application that then launches, loads, and manages its screens. By contrast, Android applications consist of a set of loosely coupled screens, represented by `Activity` classes, and `Service` classes, which are long-running background processes. This is somewhat similar to web applications (in that they can be started from various URLs), except that typically even web applications retain an application instance.

The following diagram illustrates the components of a basic Android application:



This loosely coupled architecture presents an interesting problem for multiscreen applications. Since each Activity is essentially decoupled from all others, there needs to be a way to launch additional Activities and optionally pass data to them. On Android this is accomplished by using *Intents*. Intents are classes that describe a message: both what the desired action of the message is and a data payload to send along with it.

Let's explore Activities and Intents a little more.

Activities

As mentioned, Activities are classes that provide an interface. An Activity is given a window in which to add a User Interface. This means that creating multiscreen applications involves creating multiple Activities and transitioning between them.

The `Activity` class inherits from the abstract `Context` class.

CONTEXT

`Context` is the closest Android gets to a reference to the current application that provides a mechanism for accessing the Android system. A `Context` is needed to perform many operations in Android such as:

- ➔ Accessing Android services
- ➔ Accessing preferences
- ➔ Creating views
- ➔ Accessing device resources

An Android application needs a `Context` to know what permissions the application has, how to create controls, to access preferences, etc. However, because (as we described above) Android apps are a set of loosely coupled Activities and Services, no single static application `Context` exists. Therefore, each Activity and Service inherits from the `Context` class, which has all the information the application needs. Every time a new Activity is created, it's passed a `Context` from the Activity that created it.

Since Activity derives from `Context`, any call that takes a `Context` as an argument can take an Activity.

ACTIVITY LIFECYCLE OVERVIEW

Every Activity has a lifecycle associated with it, ranging from creation to destruction. Activities can be paused or destroyed by the system at any time. The lifecycle provides a way for Activities to handle the various lifecycle methods that Android will call and gives them an opportunity to save or rehydrate state, so screens can continue to operate normally. After completing this Getting Started series, we highly recommend checking out the Activity Lifecycle document for a detailed discussion on the subject.

In short, the Android activity lifecycle comprises a collection of methods exposed within the Activity class that provide the developer with a resource management framework. This framework allows implementers to meet the unique state management requirements of each activity within an application. The activity lifecycle thus assists the developer by providing a consistent framework in which to handle resource management within the application.

Intents

Intents are used throughout Android to make things happen by sending messages. Intents are most commonly used to launch Activities within applications. To launch a new Activity, we create a new Intent, set the `Context`

and the `Activity` classes to launch, and then tell the OS to handle the Intent, which launches the Activity.

Additionally, Intents can be used to tell the OS to launch external Activities as well. For example, an application can have the intention to dial a phone number when the user taps a button. The way an application announces this intention is via an Intent for a phone-dialing action. However, the actual dialing of the number is handled by an Activity in the Phone Dialer Application.

AndroidManifest.xml

Every Android application needs to include a file called *AndroidManifest.xml.* This file contains information about the application such as:

- **Component Registration** - The components that make up the app, including registration of Activities and Intents.
- **Required Permissions -** The permissions the app requires.
- **OS Version Compatibility -** The minimum Android API level the application supports.

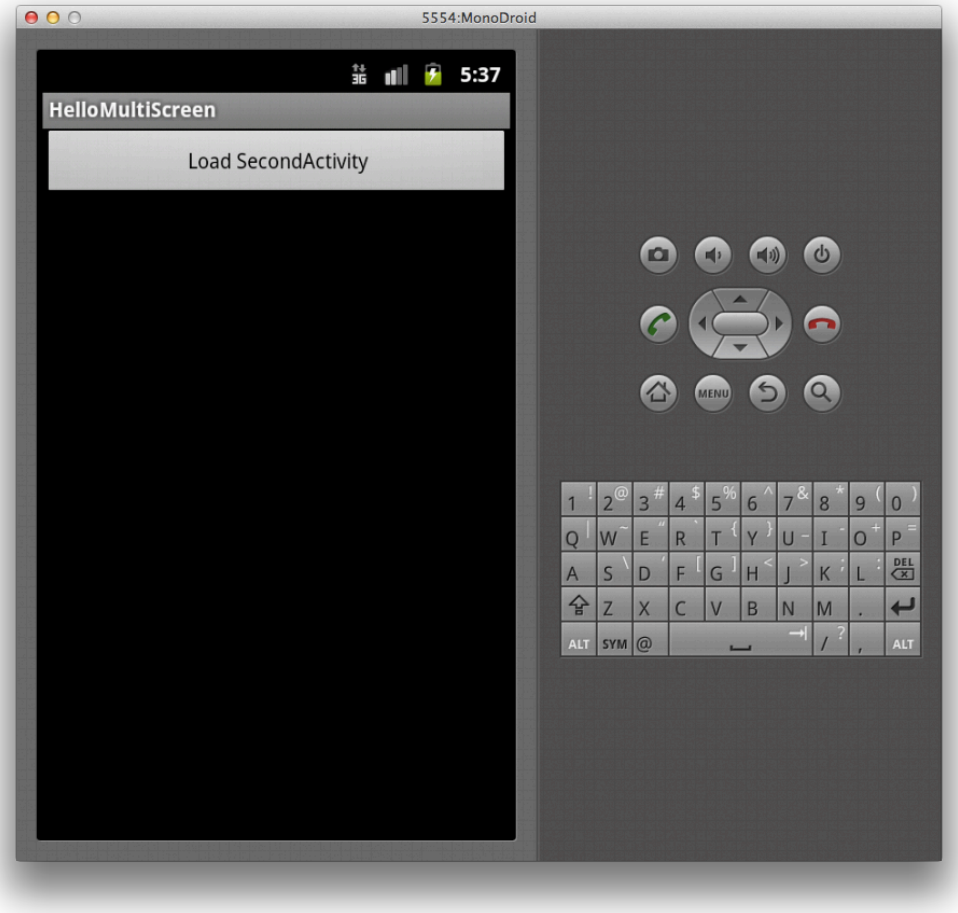
We'll examine the manifest more later on in this document during the application walkthrough. For more details, see the AndroidManifest.xml documentation on the Android Developers site.

Application Walkthrough

Now that we've covered some concepts that help us understand the strangeness of Android applications, let's create a simple, multiscreen application.

We are going to make a new application where the first Activity will include a button. Tapping the button will result in another Activity loading. For this example, we'll demonstrate loading a second Activity explicitly from a class defined within the application.

Here is a screenshot showing the application after it is launched:



Creating the Application

To get started, let's use the **Mono for Android Application** template to create a new Xamarin.Android application named `HelloMultiScreen`. Please refer to the Hello, Android tutorial if you need a review of how to create a project. The project template will create an application with an `Activity` class named `Activity1` in the file `Activity1.cs`. Open the file and right-click on the `Activity1` class name, selecting **Refactor > Rename** from the Context menu. Rename the `Activity1` class included by the template to `FirstActivity`. If you are using Visual Studio, also rename the file in the **Solution Explorer**.

Before we add any code, let's examine the `ActivityAttribute` that the template added to the `FirstActivity` class.

Using ActivityAttribute

Notice that the `FirstActivity` class is decorated with an *ActivityAttribute* as shown below:

```
[Activity (Label = "HelloMultiScreen", MainLauncher = true)]
public class FirstActivity : Activity
```

We mentioned earlier that Activities have to be registered in the `AndroidManifest.xml` file in order for them to be located by the OS, and the `ActivityAttribute` aids in doing just that. During compilation, the Xamarin.Android build process gathers all the attribute data from all the class files in the project and then creates a new `AndroidManifest.xml` file. This new file is based on the `AndroidManifest.xml` file in the project, but the attribute data is merged into it. It then bundles that manifest into the resulting compiled application (apk file).

By using C# attributes, Xamarin.Android is able to keep the declarative information that is used to register an Activity together with the class definition. Failure to include this attribute will result in Android not being able to locate the Activity, since your Activity would not be registered with the system.

Additionally, the attribute has `MainLauncher` set to `true`. This results in the Activity being registered as launchable. During installation of an app, Android will populate the application launcher with the icon of the launchable Activity.

Manifest File

When the `ActivityAttribute` is merged into the manifest, it will result in XML similar to the following:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="hellomultiscreen.hellomultiscreen" >
   <uses-sdk android:minSdkVersion="11" />
   <application android:icon="@drawable/icon"
      android:label="HelloMultiScreen">
      <activity android:name="FirstActivity"
         android:label="HelloMultiScreen">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>
   </application>
</manifest>
```

In the XML above, `FirstActivity` uses an intent filter to register with the system as being enabled for launch, with the intent filter's action set to `android.intent.action.MAIN` and its category set to `android.intent.category.LAUNCHER`. Intent filters register what actions a particular Activity can support, such as being able to handle application launch and serving as the entry point (the intention of the Activity in this case).

Adding the UI Markup

In the Hello, Android tutorial, we saw how to create our UI in either code or declaratively by using a special XML format called Android XML saved in files with an `axml` file name extension. Let’s use the declarative approach here.

We are going to add a button control to a `LinearLayout`. Recall from the Hello, Android tutorial that Android uses layout classes to group and position controls on the screen. Any controls we add, such as the `Button`, will be children of the layout. The `LinearLayout` class is used to align controls one after another, either horizontally, or vertically.

To include the button in the `FirstActivity`, add the following code to `Main.axml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
    <Button
            android:id="@+id/showSecond"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="Load SecondActivity"/>
</LinearLayout>
```

Implementing the First Activity

Now that we have our UI defined, we want the button included in `Main.axml` to be created by `FirstActivity` when the application loads.

When the application loads, `FirstActivity's OnCreate` method is called. Whether we create our UI programmatically or declaratively, we need to load it in `OnCreate`. Add the following code to load the UI from `Main.axml` and to handle the button's click event:

```
protected override void OnCreate (Bundle bundle)
{
    base.OnCreate (bundle);

    //Use UI created in Main.axml
    SetContentView (Resource.Layout.Main);

    var showSecond = FindViewById<Button> (Resource.Id.showSecond);

    showSecond.Click += (sender, e) => {
        StartActivity (typeof(SecondActivity));
    };
}
```

Let's break this code down line-by-line.

The code first loads the UI created in `Main.axml` with this line:

```
SetContentView (Resource.Layout.Main);
```

Next, it gets a reference to the button with `id showSecond` that we added in `Main.axml` by using the `FindViewById` method as shown:

```
var showSecond = FindViewById<Button> (Resource.Id.showSecond);
```

Using the button reference, an event handler is created to start an instance of `SecondActivity`, as shown below:

```
showSecond.Click += (sender, e) => {
    StartActivity(typeof(SecondActivity));
};
```

This code uses the overloaded version of `StartActivity` that takes the type of the Activity to start. Under the hood, an Intent is created, with the current Activity passed as the `Context`. This method is an optimization built on top of the underlying Android API, which requires an explicit Intent to be created and the `Context` to be passed, as in the following code:

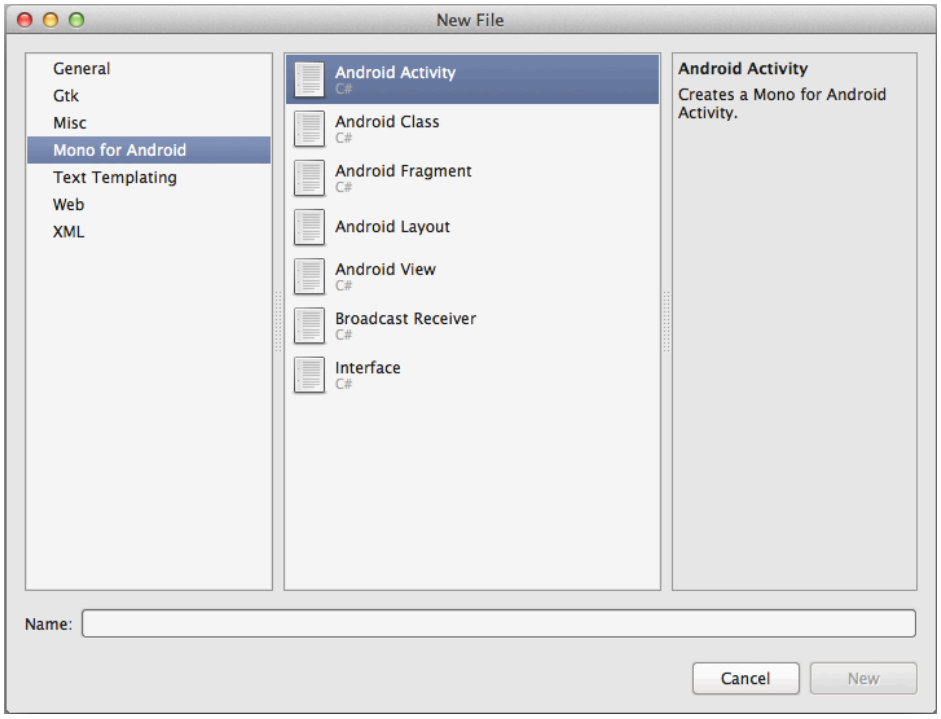
```
var second = new Intent(this, typeof(SecondActivity));
StartActivity(second);
```

This longer form is useful when we need control over the Intent instance, such as we'll see later when we pass data between Activities.

Creating a Second Activity

With the code in place to launch an instance of `SecondActivity`, let's now create the `SecondActivity` class.

The `SecondActivity` class is defined by the application. To create it, use Xamarin Studio's **File > New > File** menu to add a new Activity. Then, in the **New File** dialog, choose **Xamarin.Android** in the left pane, and then select **Android Activity** for the file template, as shown below:



Name the file `SecondActivity`, and then click **New** to create it.

Adding a Layout

The `SecondActivity` will include a `TextView` inside a `LinearLayout`. Add a new file named `Second.axml` to the layout folder in Xamarin Studio and include the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
     <TextView
             android:id="@+id/screen2Label"
             android:layout_width="fill_parent"
             android:layout_height="wrap_content"
             android:text="Hello second activity" />
</LinearLayout>
```

In order for the controls declared in XML to be programmable from code, the `Second.axml` file must have it's build action set to *AndroidResource*, which you can do by right-clicking the file in the solution explorer and selecting **Build Action > AndroidResource** from the context menu.

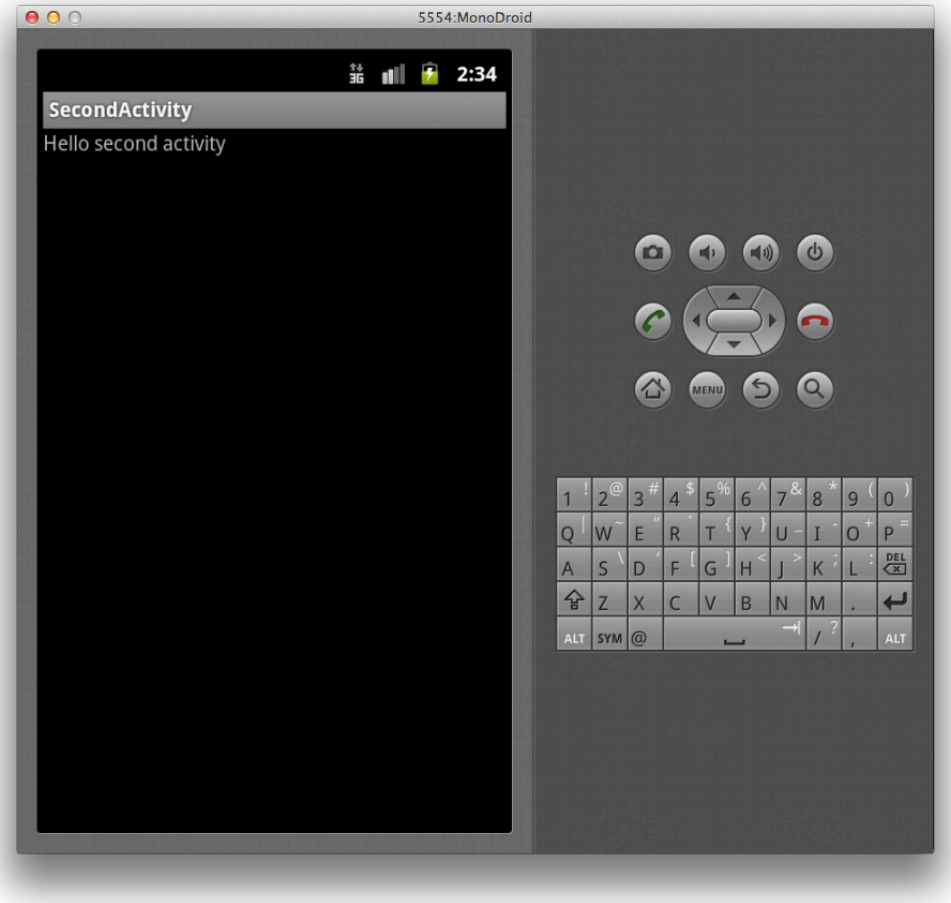
Loading the UI from the Second Activity

Finally, to load the UI from `Second.axml`, first load `SecondActivity`. Next, we'll call `SetContentView` in `SecondActivity's OnCreate` method as shown below:

```
[Activity (Label = "SecondActivity")]
public class SecondActivity : Activity
{
    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);

        // Create your application here
        SetContentView (Resource.Layout.Second);
    }
}
```

Running the application and clicking the **showSecond** button results in `SecondActivity` loading as shown in the screenshot below:



Now that we've seen how to load multiple Activities, let's go a step further and show how to pass data between Activities.

Sending Data Between Activities

We can use the `PutExtra` method of an Intent to send a data payload with Intent. For example, change the button-click event code in the `FirstActivity` class as follows:

```
showSecond.Click += (sender, e) => {
    var second = new Intent(this, typeof(SecondActivity));
    second.PutExtra("FirstData", "Data from FirstActivity");
    StartActivity(second);
};
```

In the code above we use the longer form of creating Intent and passing it to `StartActivity`. The other new thing introduced here is the line:

```
second.PutExtra("FirstData", "data from FirstActivity");
```

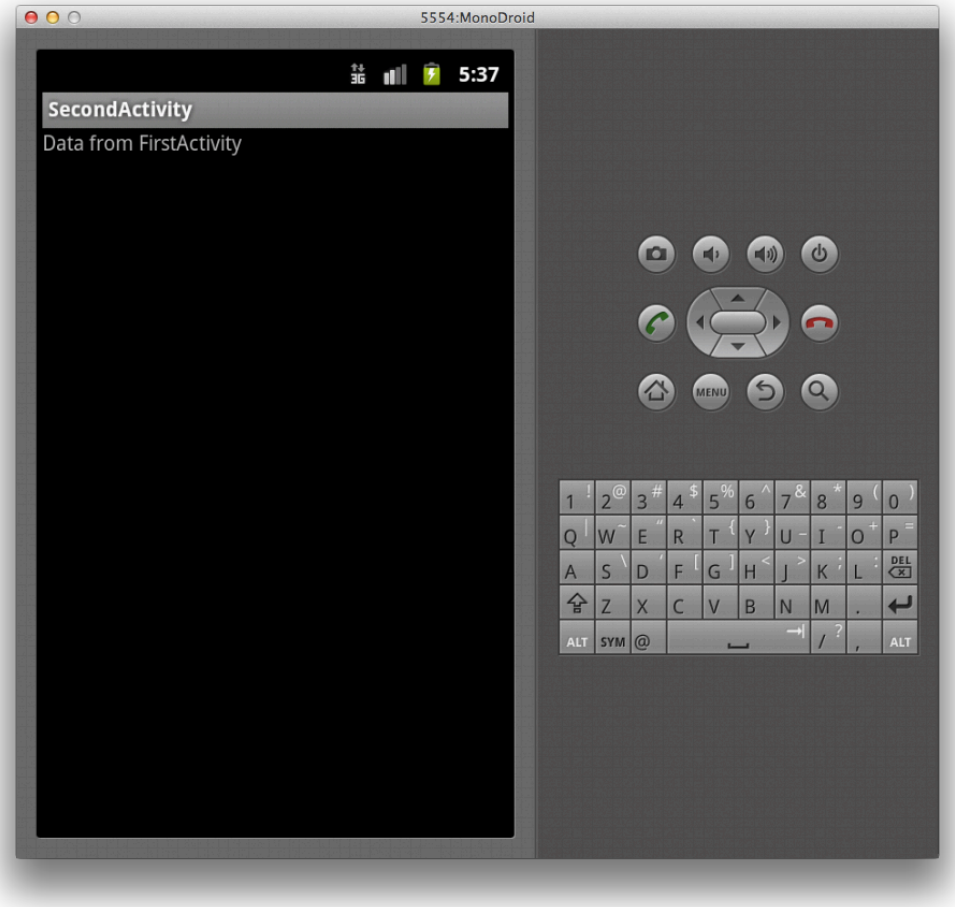
This code adds a string to include with the Intent. This design is similar to how web applications pass data between pages by using query strings or form data. In addition to strings, the `PutExtra` method also includes several overloads for passing data of various types.

In the `SecondActivity` class, we simply use the `TextView` to show the string sent with the Intent as follows:

```
var label = FindViewById<TextView> (Resource.Id.screen2Label);
label.Text = Intent.GetStringExtra("FirstData") ?? "Data not available";
```

We call `Intent.GetStringExtra("FirstData")` to pull out the data that we passed over from the `FirstActivity`. We used the *null coalescing operator* ("??") there because when accessing Intent data, it may be null.

When the Activity loads, the text from the `FirstActivity` is displayed, as shown below:



Summary

In this chapter we examined the constituent parts that make up an Android app, and how they work together. We showed how Activities can be used to represent the screens in an Android application and how to use Intents to navigate between them. We also introduced the AndroidManifest.xml file and showed how Xamarin.Android uses C# attributes to make registering Activities in the manifest more concise. Finally, we examined how to include a data payload with an Intent in order to pass data between Activities.