Introduction to Android Resources

Xamarin Android Level 1, Chapter 3

Xamarin Inc.

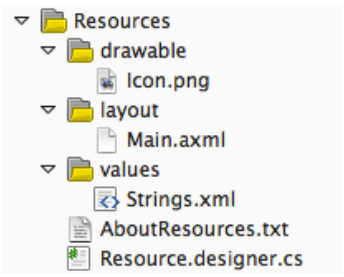
Overview

An Android application seldom comprises only source code. Many other files make up a typical application; a partial list of common components would include video, images, and audio files—just to name a few. Collectively, these non-source code files are referred to as *resources* and are compiled (along with the source code) during the build process, and then are packaged as an *APK* for distribution and installation onto devices:

Resources offer several advantages to an Android application:

- **Code-Separation** – Separates source code from images, strings, menus, animations, colors, etc. Separating resources from source code in this way can help streamline localization.
- **Targeting of multiple devices** – Provides simpler support of different device configurations without requiring code changes.
- **Compile-time Checking** – Allows resource usage to be checked at compile time, when it is easier to catch and correct mistakes, as opposed to runtime when it is more difficult to locate errors and costlier to correct them. This is possible because resources are static and are compiled into the application.

When a new Xamarin.Android project is started, a special directory called Resources is created, along with some subdirectories:



In the image above, the application resources are organized into subdirectories, according to their types. Images go in the **drawable** directory; views go in the **layout** subdirectory, etc.

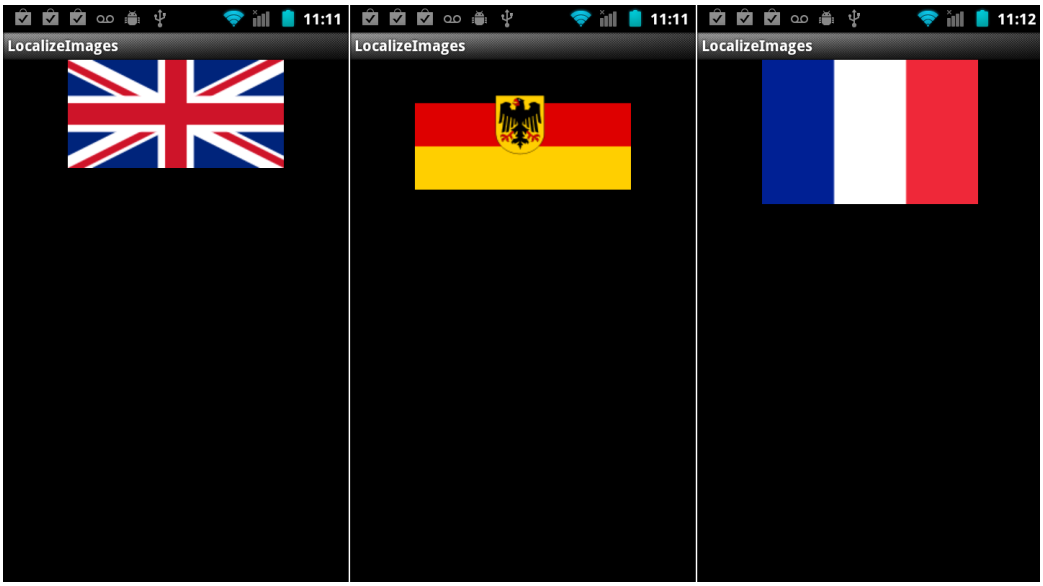
There are two ways to access these resources in a Xamarin.Android application: *programmatically* in code and *declaratively* in XML, using a special XML syntax.

These resources are called *default resources* and are used by all devices, unless a more specific match is specified. Additionally, all types of resources may optionally

have *alternate resources* that Android may use to target specific devices. For example, resources may be provided to target the user's locale, the screen size, or if the device is rotated 90 degrees from portrait to landscape, etc. In each of these cases, Android loads the resources for use by the application without any extra coding effort by the developer.

Alternate resources are specified by adding a short string—called a *qualifier*—to the end of the directory that is holding a given type of resources.

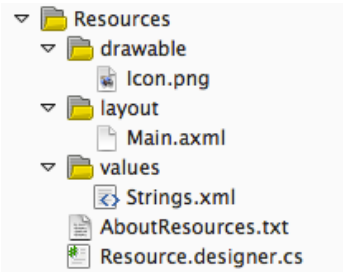
For example, `resources/drawable-de` specifies the images for devices that are set to a German locale, while `resources/drawable-fr` would hold images for devices set to a French locale. The image below shows an example of how the resources change in response to changes in the locale of the device:



This chapter gives an introduction of the basic resource concepts and how to use them in your application. For comprehensive coverage on resources, please see the Resources in Android guide at Xamarin's web site.

Android Resource Basics

Almost all Android applications use some sort of resources; at a minimum, applications often code the user interface layouts in the form of XML files. In fact, when a Xamarin.Android application is first created, default resources are set up by the Xamarin.Android project template as reflected in the following screenshot:

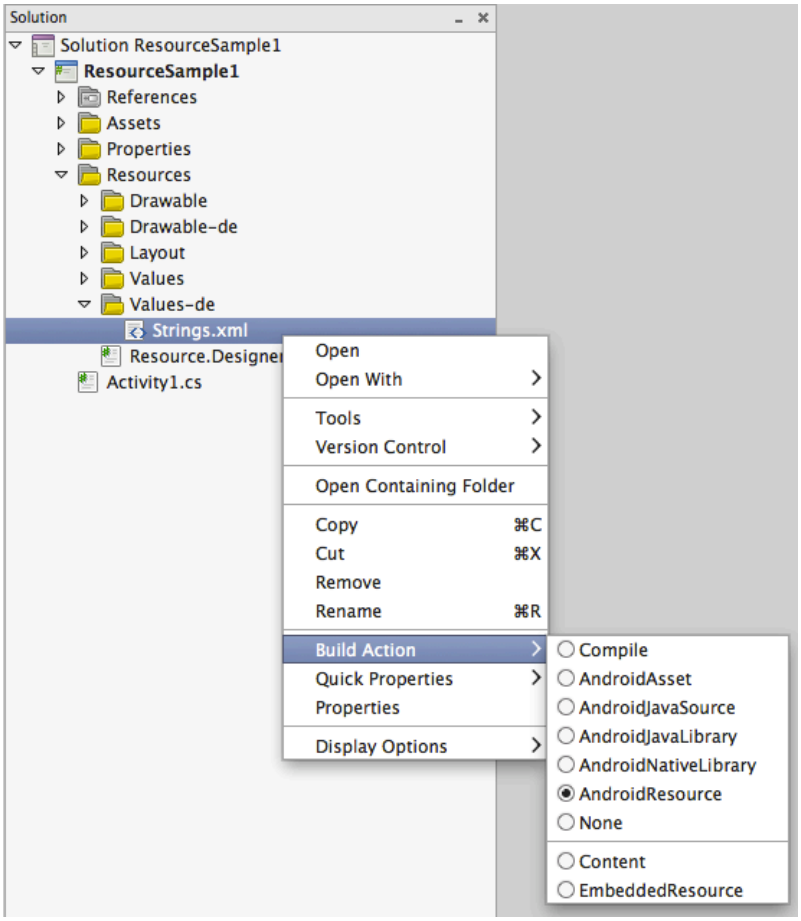


The five files that make up these default resources are created in the Resources folder. Following are descriptions of these files:

- Icon.png – The default icon for an application.
- Main.axml – The default user interface layout file for an application. Note that while Android uses the `.xml` file extension, Xamarin.Android uses the `.axml` file extension.
- Strings.xml – A string table to help with localization of an application.
- Resource.designer.cs – Xamarin.Android automatically generates and maintains this file. The file contains the unique IDs assigned to each resource. The purpose of this file is very similar to the purpose of the `R.java` file that an Android application written in Java would have. Xamarin.Android tools not only automatically create this file, but also regenerate it from time to time.
- AboutResources.txt – This file is not necessary and may safely be deleted. It provides a high-level overview of the Resources folder and the files in that folder.

Creating and Accessing Resources

Creating resources is as simple as adding files to the directory for the resource type in question. The screenshot below shows how string resources for German locales were added to a project. When Strings.xml was added to the file, the Build Action was automatically set to *Android Resource* by the Xamarin.Android tools:



This configuration allows the Xamarin.Android tools to properly compile and embed the resources in the APK file. However, if the *Build Action* is not set to **AndroidResource**, then the files will be excluded from the APK, and any attempt to load or access the resources will result in a run-time error and the application will crash.

Also, it's important to note that while Android only supports lowercase file names for resource items, Xamarin.Android is more forgiving; it will support both uppercase and lowercase file names.

After resources have been added to a project, there are two ways to use them in an application—programmatically (inside code), or from XML files.

REFERENCING RESOURCES PROGRAMMATICALLY

At compile time, the Xamarin.Android and Android tools assign a unique resource ID to each resource in a Xamarin.Android project. The resource ID is an integer defined in a special class called `Resource`, which is found in the file **Resource.designer.cs**, and looks something like this:

```
public partial class Resource
{
    public partial class Attribute
    {
    }
    public partial class Drawable {
```

```
            public const int Icon=0x7f020000;
        }
        public partial class Id
        {
            public const int Textview=0x7f050000;
        }
        public partial class Layout
        {
            public const int Main=0x7f030000;
        }
        public partial class String
        {
            public const int App_Name=0x7f040001;
            public const int Hello=0x7f040000;
        }
    }
```

Each resource ID is contained inside a nested class that corresponds to the resource type. For example, when the file **Icon.png** was added to the project, Xamarin.Android updated the `Resource` class, creating a nested class called `Drawable` with a constant inside it named `Icon`. This structure allowed the file `Icon.png` to be referred to in code as `Resource.Drawable.Icon`. However, note that the `Resource` class should not be manually edited, as any changes that are made to it will be overwritten.

When referencing resources programmatically (in code), they can be accessed via the Resources class hierarchy that uses the following syntax:

```
[<PackageName>.]Resource.<ResourceType>.<ResourceName>
```

- **PackageName** – The package that is providing the resource. This is only required when resources from other packages are being used.
- **ResourceType –** This is the nested resource type that is within the `Resource` class described above.
- **Resource Name –** This is the file name of the resource (without the file name extension) or the value of the `android:name` attribute for resources that are in an XML element.

REFERENCING RESOURCES FROM XML

Resources in an XML file are accessed by following a special syntax: `@[<PackageName>:]<ResourceType>/<ResourceName>`.

- **PackageName** – The package that is providing the resource. This is only required when resources from other packages are being used.
- **ResourceType –** This is the nested resource type that is within the `Resource` class.
- **Resource Name –** This is the file name of the resource (without the file name extension) or the value of the android:name attribute for resources that are in an XML element.

For example, the contents of a layout file, `Main.axml`, are as follows:

```
<?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">
    <ImageView android:id="@+id/myImage"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/flag" />
</LinearLayout>
```

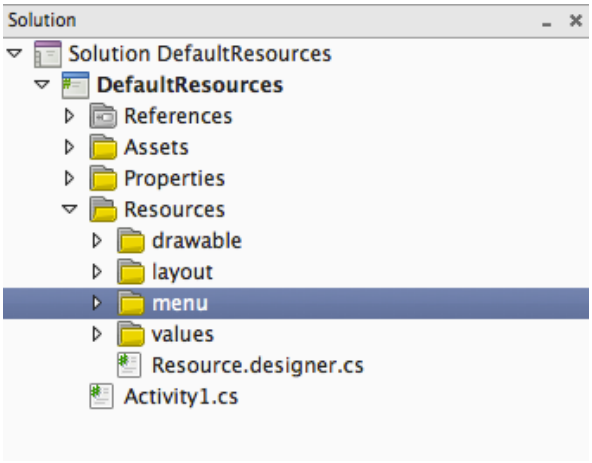
This example has an ImageView that requires a drawable resource named `flag`. There are two things to notice about the XML element for the ImageView:

- The `id` attribute is set using the special syntax `@+id/myImage`. This tells Android to generate the constant `Resource.Id.myImage` which can be used to programmatically reference the view.
- The `ImageView` has its `src` attribute set to `@drawable/flag`. When the Activity starts, Android looks inside the directory `Resource/Drawable` for a file named `Activity1Image` and loads that file and displays it in the `ImageView`. When this application is run, it looks like the following image:

Default Resources

Default resources are items that are not specific to any particular device or form factor, and therefore are the default choice by the Android OS if no more specific

resources can be found. Because of this, they're the most commonly created type of resource. They're organized, according to their resource type, into subdirectories of the Resources directory:



In the image above, the project has default values for drawable resources, layouts, menus, and values (XML files that contain simple values).

The table below lists some of the more commonly used resource types in an Android application. This is a subset of the full list of resource types that are available in an Android application. For a more comprehensive list consult Xamarin's Resource Document on default resources:

Directory	Description
color	XML files that describe a state list of colors. To understand color state lists, consider a UI widget such as a *button*. A button may have different states such as *pressed* or *disabled*, and the button may change color with each change in state. The list is expressed in a state list.
drawable	Drawable resources are graphics that can be compiled into an application and then accessed by API calls or referenced by other XML resources. Some examples of drawables are bitmap files (`.png`, `.gif`, `.jpg`), special resizable bitmaps known as *Nine-Patches*, state lists, generic shapes defined in XML, etc.
layout	XML files that describe a user interface layout, such as an Activity or a row in a list.

menu	XML files that describe application menus, such as *Options Menus*, *Context Menus*, and *submenus*.
values	XML files that contain simple values. An XML file in the values directory does not necessarily define only a single resource, but instead can define multiple resources. For example, one XML file may hold a list of string values, while another XML file may hold a list of color values.

Adding Resources to a Project

Let's create an application that demonstrates how to add some images and strings to an application as resources. The application will then use those resources declaratively and programmatically.

1. Begin by creating a new **Xamarin.Android** project named `DefaultResources`.
2. To see an example of how string resources are used, let's look at the contents of the file of `Main.axml`. Notice the `android:text` attribute of the `Button`. This attribute is already set to use a string resource named `hello`.

```
<?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/myButton"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />
</LinearLayout>
```

When Android inflates a View using this layout file, it will look in the file `/Resources/values/Strings.xml` for a string element which has the `name` attribute set to `hello`. The system will display the value of that element in the Button. The contents of `Strings.xml` is provided for reference:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
        <string name="hello">Hello World, Click Me!</string>
        <string name="app_name">DefaultResources</string>
</resources>
```

3. We want the application to use a string resource to display the number of times the button was clicked. Add a string resource by editing the file `/Resources/values/Strings.xml`, and add the following:

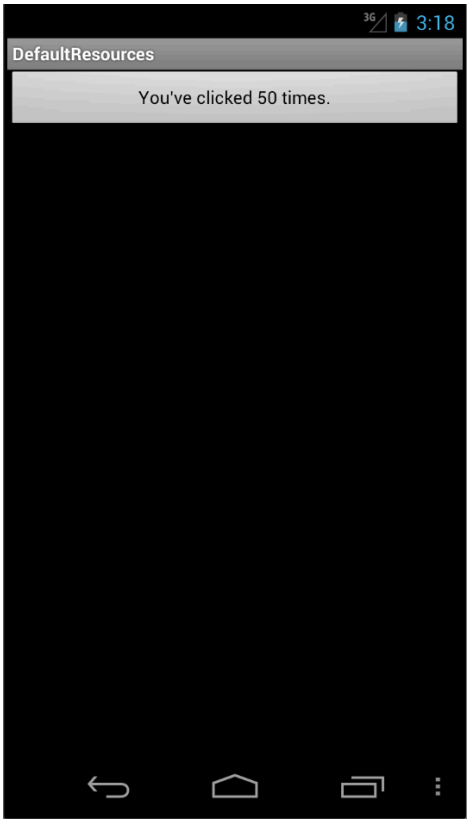
```
<string name="button_clicked_text">You\'ve clicked %d times.</string>
```

This creates a parameterized string named `button_clicked_text` (notice that the ' was escaped with a \). This string has a placeholder, `%d`, which will be replaced with some a value that is programmatically provided by our code. How this is done will be covered in the next step.

4. Finally we have to change the Activity so that it will use the string resource that was just added. Open the file `Activity1.cs`, and look for the code that sets the button text. Replace that line of code with the following line:

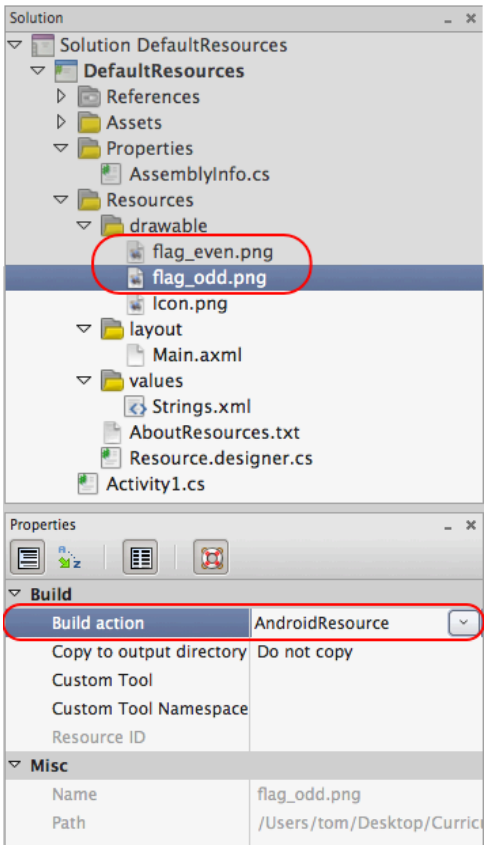
```
button.Text= GetString(Resource.String.button_clicked_text, count++);
```

`GetString` is a helper method on an Activity that will load the string that is referenced by `Resource.String.button_clicked_text`, replace the parameter placeholder with the value of `count`, and increment the value of `count`. If the application is run, the text displayed in the button will change with each click, similar to what is displayed in the following screenshot:



In the next part of this walkthrough some bitmaps will be added to the project and the Activity will display a different image depending on the value of `count`.

5. Download the file `03 - Introduction to Android Resources - Drawables.zip`, and add the two files to the directory `/Resources/drawable`. Ensure that the **Build Action** of each file is set to **AndroidResource**, as shown in the following screenshot:

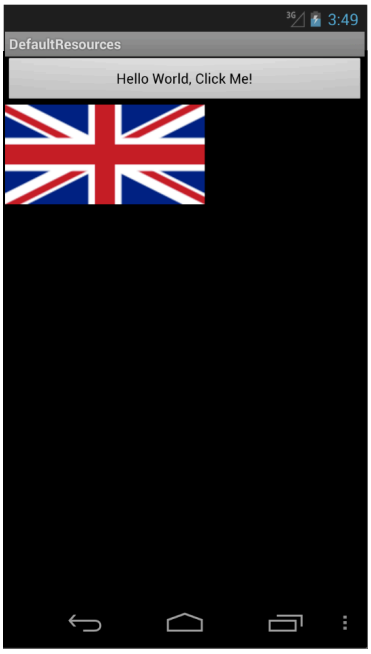


When the application gets compiled, the PNG files will be included in the application as a resource with a unique integer id. The Android tools will generate a unique id each time the project is compiled. This unique Id is stored as a constant in the class `Resource.Drawable`.

6. To see an example of how to use a resource declaratively, we'll display an image on the form when the activity is first displayed. Edit the layout file `Main.axml`, and add an `ImageView` to the layout:

```
<ImageView
    android:id="@+id/myImageView"
    android:src="”@drawable/flag_even”
    android:layout_width = "wrap_content"
    android:layout_height= "wrap_content"
/>
```

In this example, notice that the `src` attribute of the `ImageView` is set using the declarative syntax that was introduced earlier on in this chapter. When Android inflates this layout file as a View, it will try to load the drawable resource named `flag_even` and display it in the `ImageView`. The application, when run, should appear similar to the following screenshot:

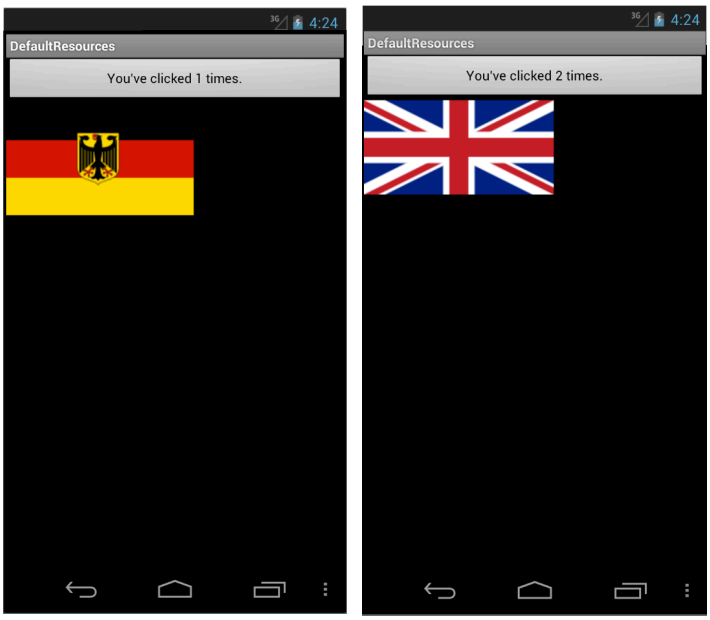


7. Next lets modify the application so that the image will change depending on the value of `count`. Edit the file `Activity1.cs` and update the `OnCreate` method so that it resembles the following code:

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

    SetContentView(Resource.Layout.Main);

    Button button = FindViewById<Button>(Resource.Id.myButton);
    ImageView image = FindViewById<ImageView>(Resource.Id.myImageView );

    button.Click += delegate
    {
        if (count % 2 == 0) {
            // Even values of count, so display flag_even
            image.SetImageResource(Resource.Drawable.flag_even);
        }
        else {
            image.SetImageResource(Resource.Drawable.flag_odd);
        }
        button.Text= GetString(Resource.String.button_clicked_text,
count++);
    };
}
```

The new code does two things. First a reference to the `ImageView` is obtained. Then when the button is clicked, the `ImageView` will be updated to display one flag for odd values, and another flag for even values.

8. Finally, if we run the application and click on the Button we should see the image change with every click:



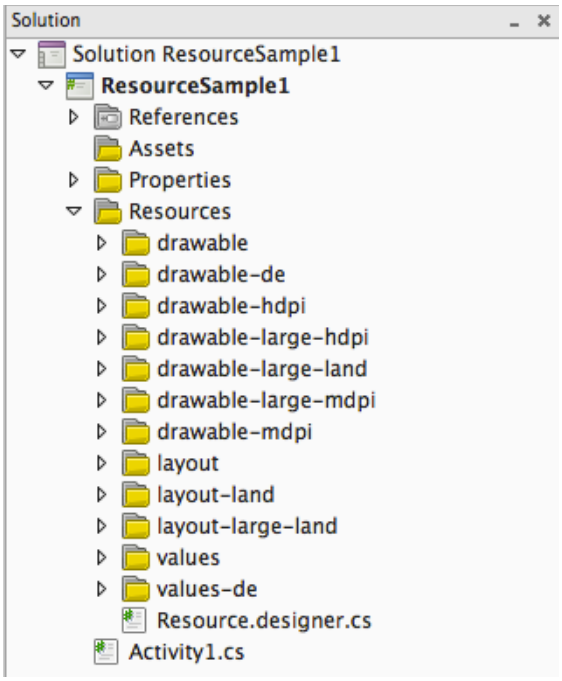
At this point we have an Android application that demonstrates how to add resources to an application. We learned how to use those applications declaratively, and how the application can use different resources while it is running. The next section will introduce the concept of *Alternate Resources*.

Alternate Resources

Alternate resources are those resources that target a specific device or run-time configuration, such as the current language, particular screen size, or pixel density. If Android can match a resource that is more specific to a particular device or configuration than the default resource is, then that resource will be used instead. If it does not find an alternate resource that matches the current configuration, then the default resources will be loaded.

Alternate resources are organized just like default resources, as a subdirectory inside the Resources folder. The name of the alternate resource subdirectory is in the form `<ResourceType>-<Qualifier>`.

Qualifier is a term that identifies a specific device configuration. There may be more than one qualifier in a name, with each qualifier separated by a dash. For example, the screenshot below shows a simple project that uses qualifiers and has alternate resources for various configurations such as locale, screen density, screen side, and orientation:

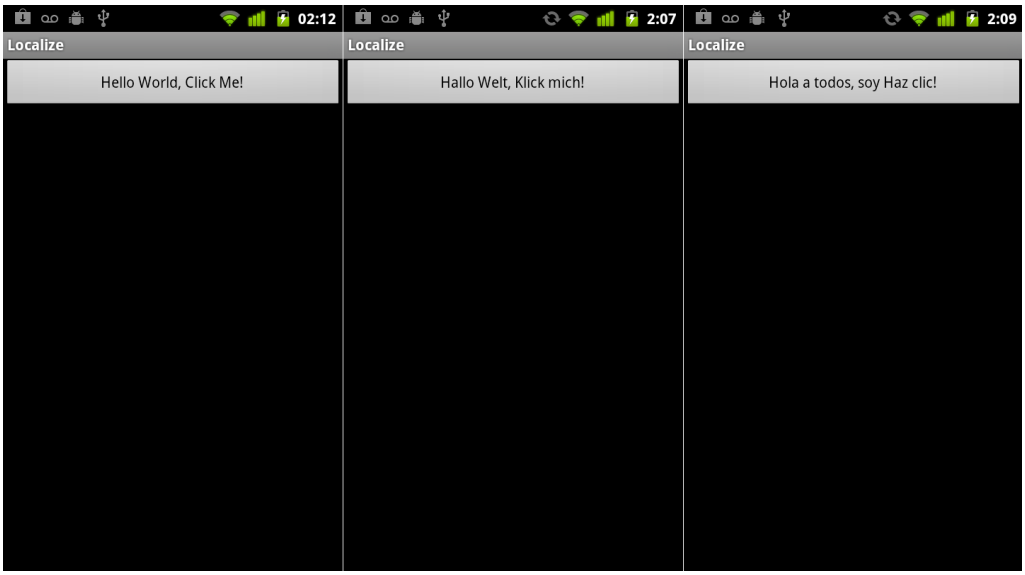


A detailed discussion of Alternate Resources is out of scope in the context of this curriculum, however for a more comprehensive discussion on them, as well as a discussion on how they're chosen by the OS and a complete list of quantifiers, please see the Alternate Resources section of Resources in Android guide.

Application Localization and String Resources

Application localization is the act of providing alternate resources to target a specific region or locale. For example, an application might be given localized language strings for various countries, or might change colors or layout to match particular cultures. Android will load and use the resources appropriate for the device's locale at runtime without any changes to the source code.

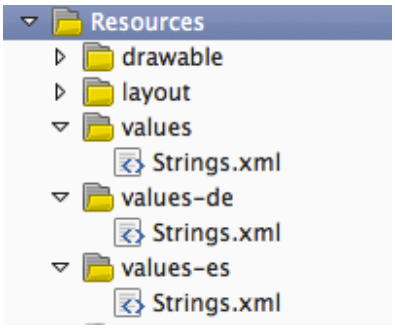
For example, the image below shows the same application running in three different device locales, but the text displayed in each button is specific to the locale that each device is set to:



In this example, the contents of a layout file, `Main.axml`, look something like this:

```
<?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/myButton"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
    />
</LinearLayout>
```

The resource ID `@string/hello` tells Android to load the string for the button from the resource file `Strings.xml`. Android will look for this file in the resource folder that is appropriate for this locale. For example, a French locale that folder will be `/Resources/values-fr` and for a German locale it will be `/Resources/values-de`. All other locales will use the strings in the default folder `/Resources/values`:



As with Alternative Resources, Localization is out of scope for this chapter. For more information, see the Application Localization and String Resources section of the Resources in Android guide on Xamarin's web site.

Summary

This chapter covered how to use Android resources in an Android application. It introduced default resources and through an application that demonstrated how to add resources to a project use those resources declaratively and programmatically. Finally, alternate resources were briefly introduced with a short discussion on how they maybe used for tasks such as localization.