Designers and User Interface

Evolve Advanced Track, Chapter 3

Overview

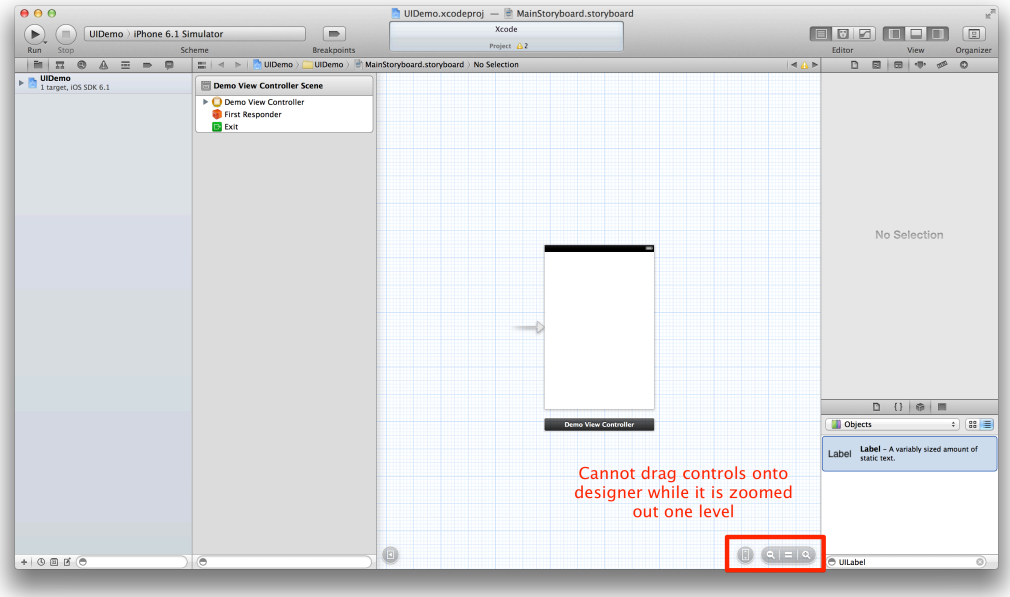
This chapter surveys a variety of different topics related to user interface development on both iOS and Android. It covers Xamarin.iOS integration with Interface Builder, showing how to work with the designer for both xibs and storyboards. It shows how to create views with Interface Builder and connect them to C# code. Then it delves into creating custom views entirely in code, as well as providing tips for common user interface scenarios, such as providing validation and handling device rotation. Next, it shifts over to Android, showing how use the Xamarin.Android designer to design user interfaces across multiple configurations. Finally, the chapter provides some additional tips on Android UI development.

Xcode's Interface Builder

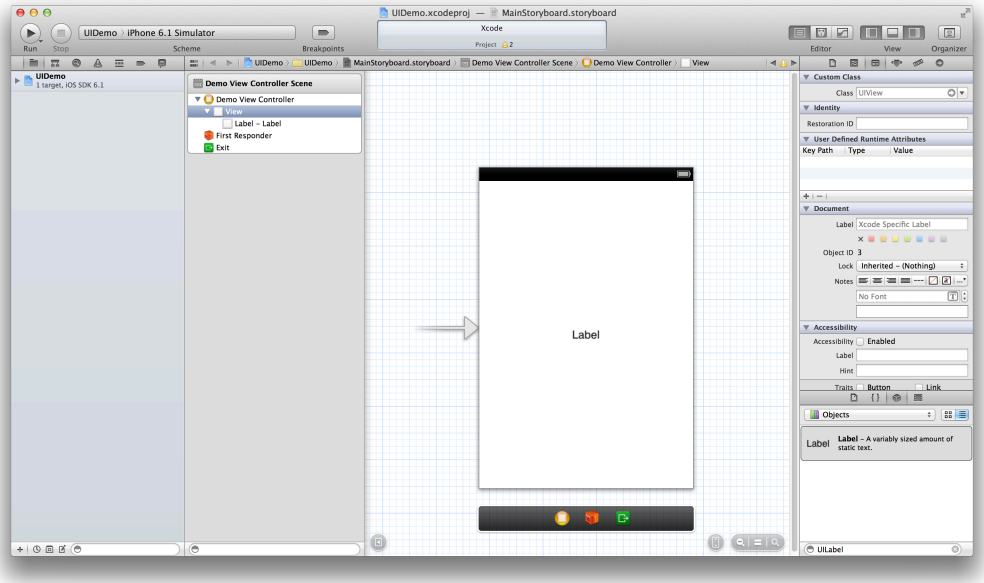
Tips

Zoom to Add Controls

Sometimes you can't drag controls unless you're at the right zoom level. For example in the screenshot below, the designer is zoomed out one level, preventing controls from being added:



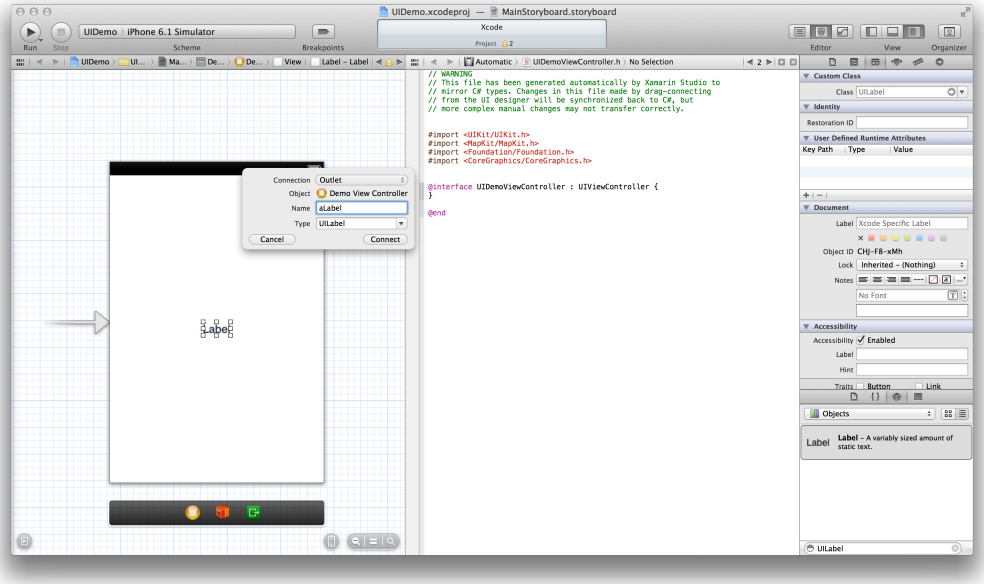
Zooming in allows controls to be added via drag and drop, as shown below where a label has been added to the view:



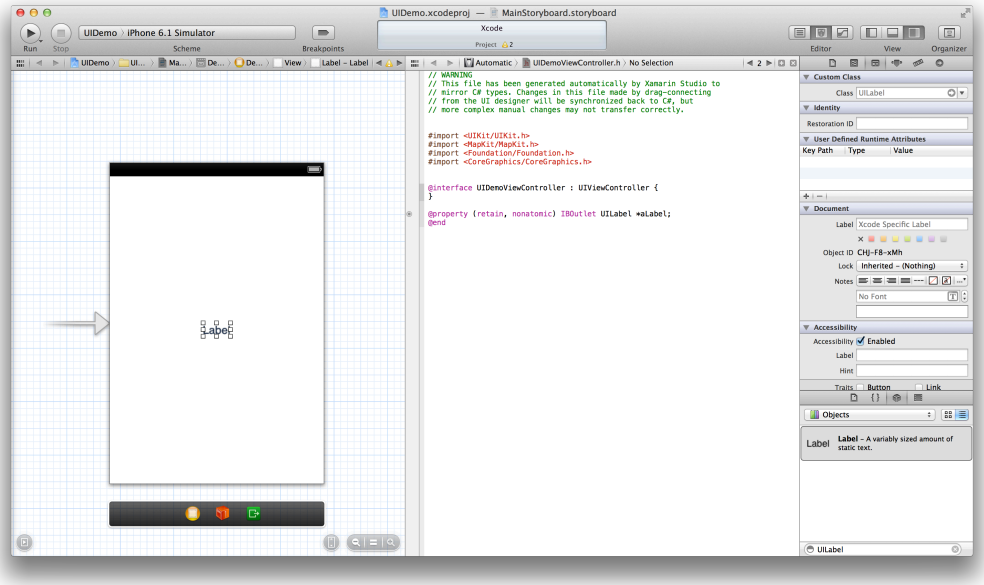
Cleaning up Failed Connections (Outlets and Actions)

Xamarin Studio is integrated with Xcode to watch for connections. Xamarin Studio generates properties for outlet connections and partial methods for action connections in a *.designer.cs* file. If for whatever reason, the code generation does not work properly, the connections need to be removed from Xcode.

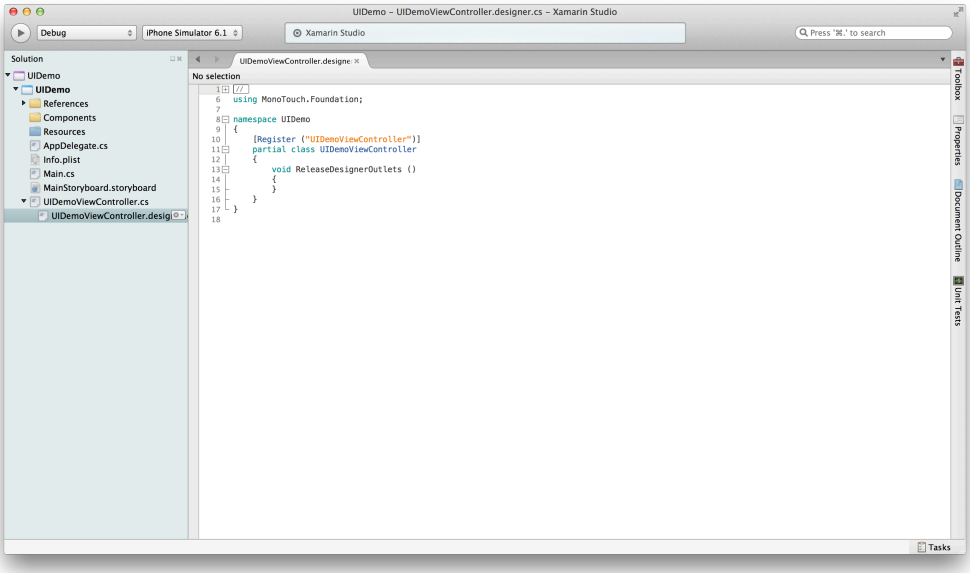
For example, consider the following storyboard where an outlet named `aLabel` is named in the outlet connection popup after Ctrl-dragging from the control in the designer to the header (.h) file:



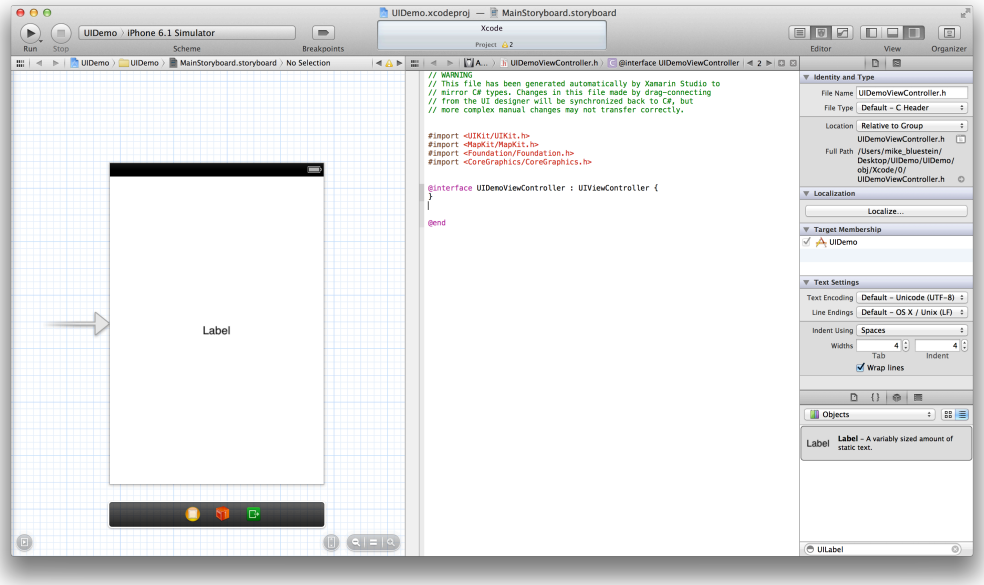
When the **Connect** button is pressed, the outlet is created:



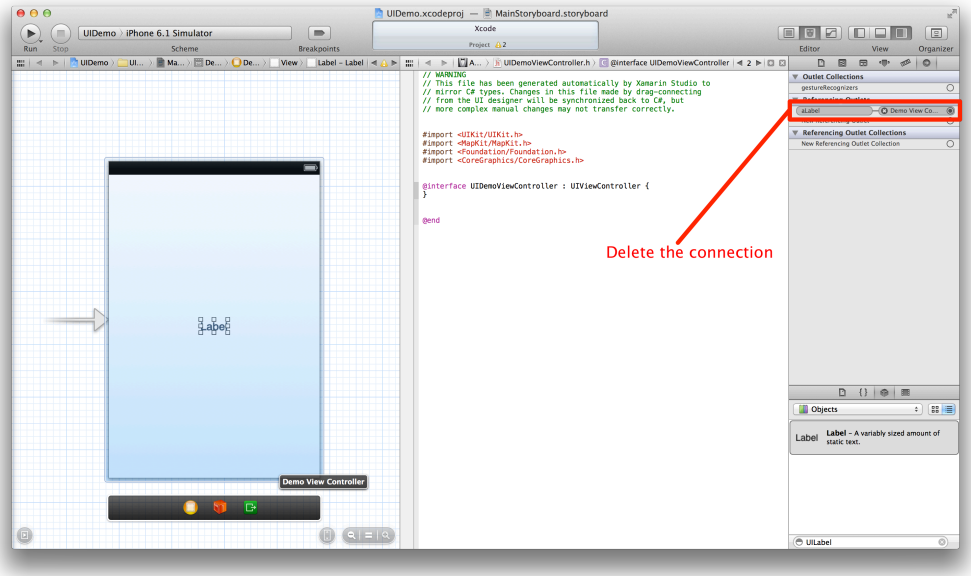
However, switching over to Xamarin Studio, we see that something went awry and the .designer.cs does not have a generated property for the label:



To clean up the outlet, first remove it from the header in Xcode:

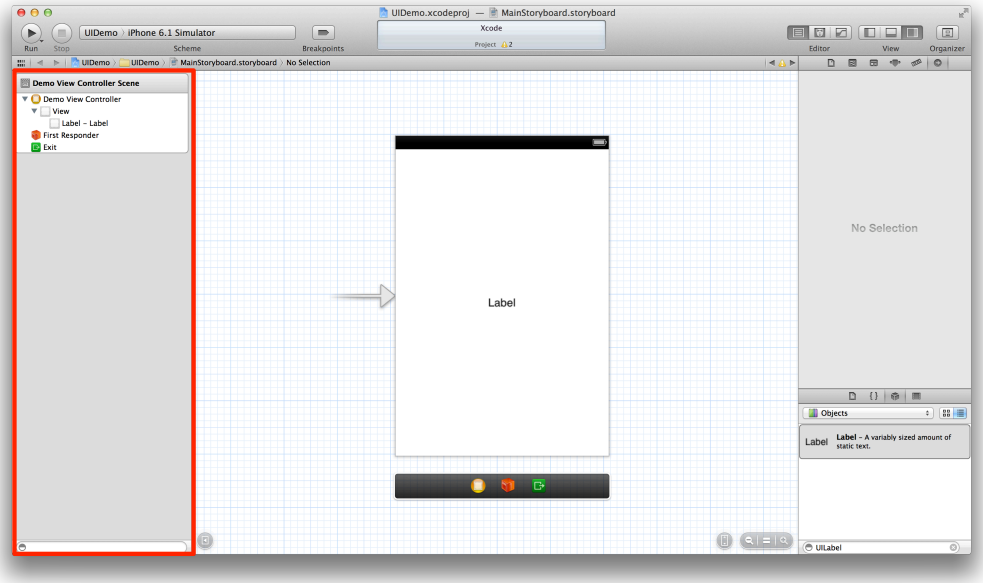


Then, with the label selected on the designer, open the **Connections Inspector** and delete the connection under the **Referencing Outlets** section:

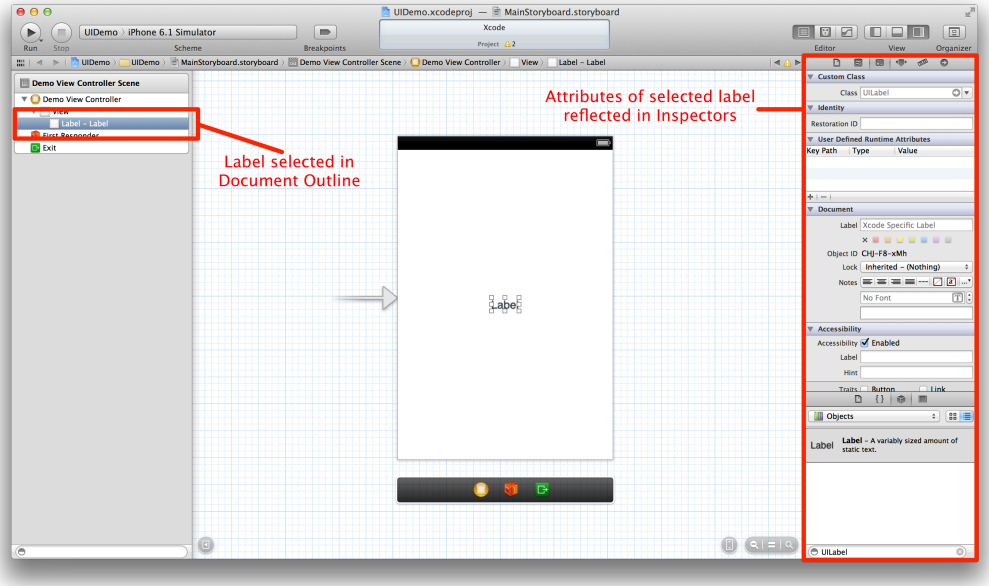


Locating Objects view the Tree

As user interfaces become more complex, it can sometimes be difficult to select controls on the designer. Interface Builder includes a **Document Outline**, which presents the view hierarchy in a tree as shown below:



By selecting a control in the **Document Outline**, it is automatically selected in the designer, and its attributes are reflected in the inspectors on the right side of Interface Builder:



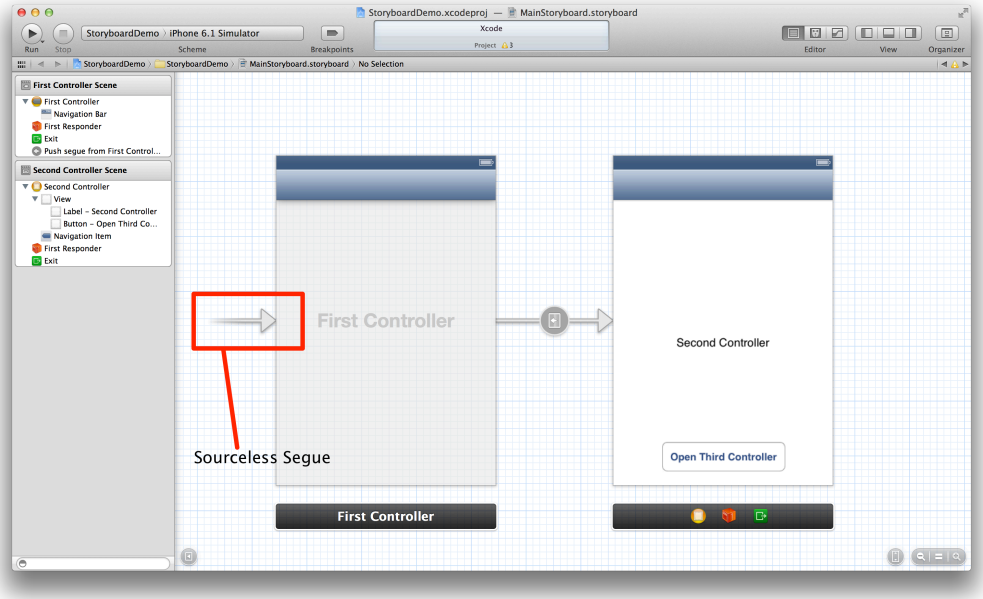
Storyboards

Storyboards offer a way to design user interfaces in Xcode. Unlike Xibs, where you create a separate Xib for each view controller and program the navigation between them manually, storyboards allow multiple controllers to be designed in a single file. A storyboard uses a design surface that offers WYSIWYG editing of the application user interface. Additionally, a storyboard offers developers a visual way

to interact with view controllers to control how an application transitions between them.

Sourceless Segue

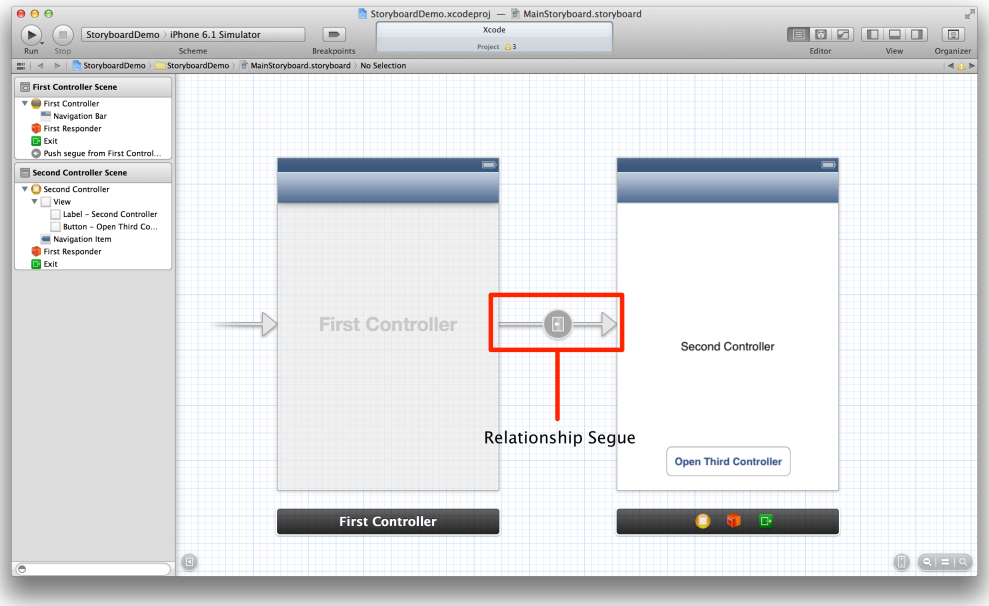
The following screenshot, from the StoryboardDemo sample project, shows a storyboard where a navigation controller has been set up as the root controller:



What makes it the root controller is the arrow pointing to it, known as a *sourceless segue*. Segues allow you to visually construct transitions between controllers.

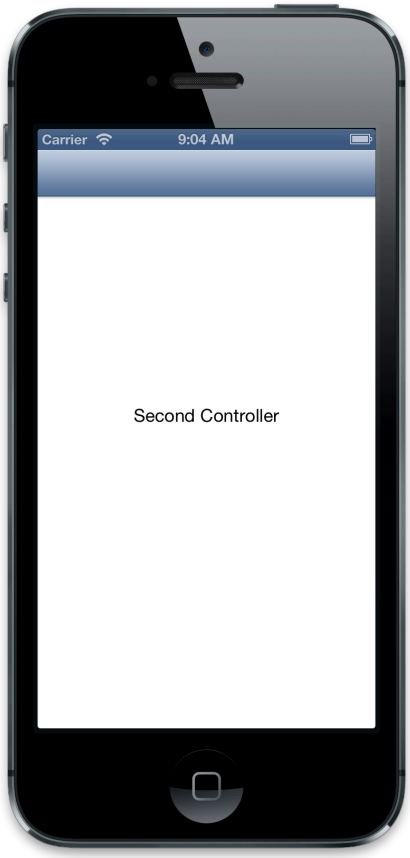
Relationship Segue

A segue is also used to set the navigation controller's root, not to be confused with the root controller of the application. The following screenshot shows a second arrow pointing from the navigation controller to a second controller:



This arrow is a *relationship segue*. It set the root view controller of the navigation control to be the second controller.

When the application runs, the navigation controller displayed, with the second controller visible within the navigation controller:



Adding Views to the View Hierarchy in Code

The `ViewDidLoad` method of `UIViewController` is the appropriate place to create any views programmatically and add them to the view hierarchy.

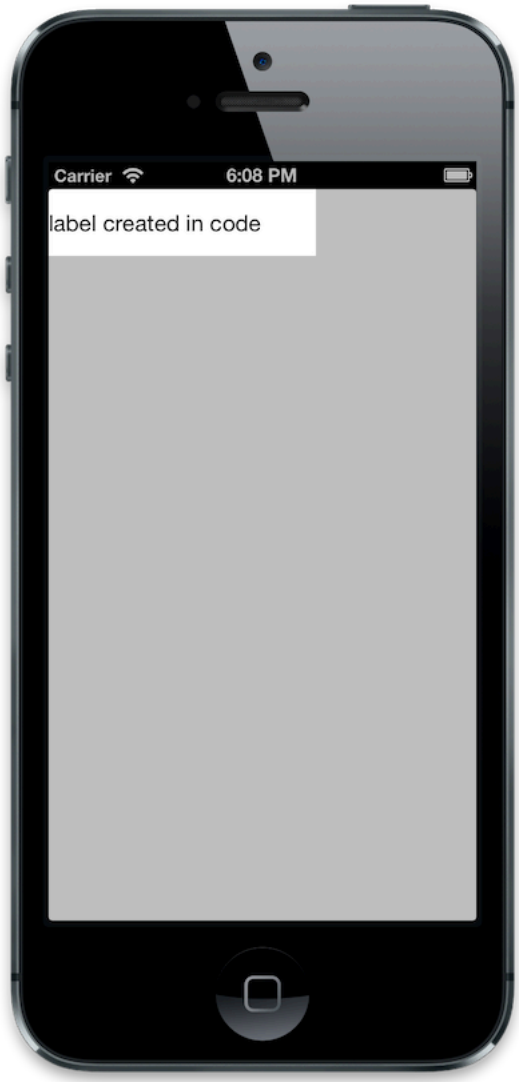
For example, the following code snippet creates a `UILabel` instance and adds it as a subview of the controller view:

```
UILabel label;

public override void ViewDidLoad ()
{
        base.ViewDidLoad ();

        label = new UILabel (new RectangleF(0,0,200,50));
        label.Text = "label created in code";
        View.AddSubview (label);
}
```

This results in the UI shown below:



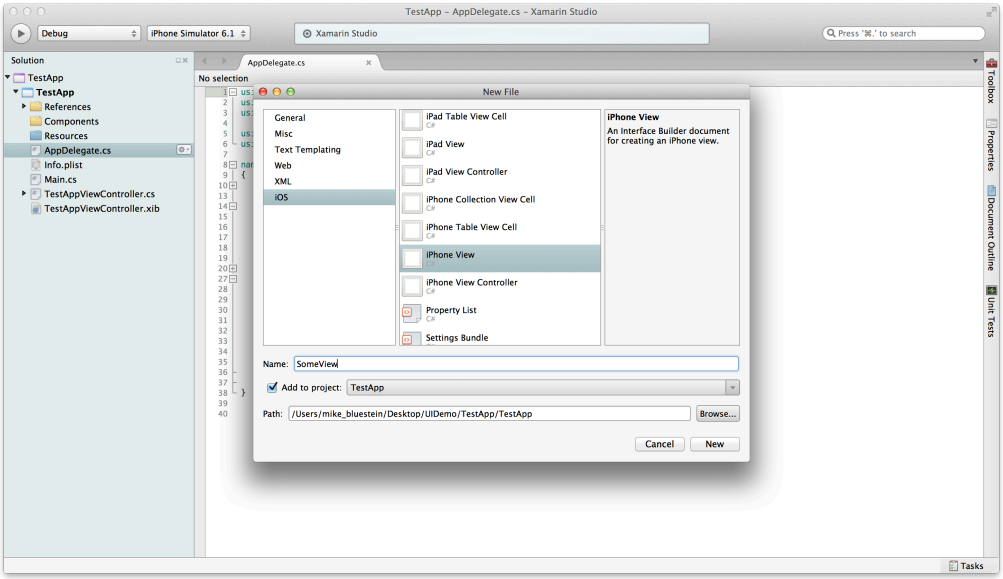
Creating Custom Views in a Xib

The *iOS View XIB template* in Xamarin Studio allows adding a stand-alone XIB file that can be attached to some backing class. This allows a reusable view to be designed in Interface Builder. However, to use such a view requires manually creating a backing view class.

Let's demonstrate creating a view in Interface Builder and connecting it to a backing C# class in Xamarin Studio.

Adding an iPhone View Xib

First, Create a new solution called `TestApp` using the **Single View Application** template, and add a new file called `SomeView` using **iPhone View** file template, as shown below:

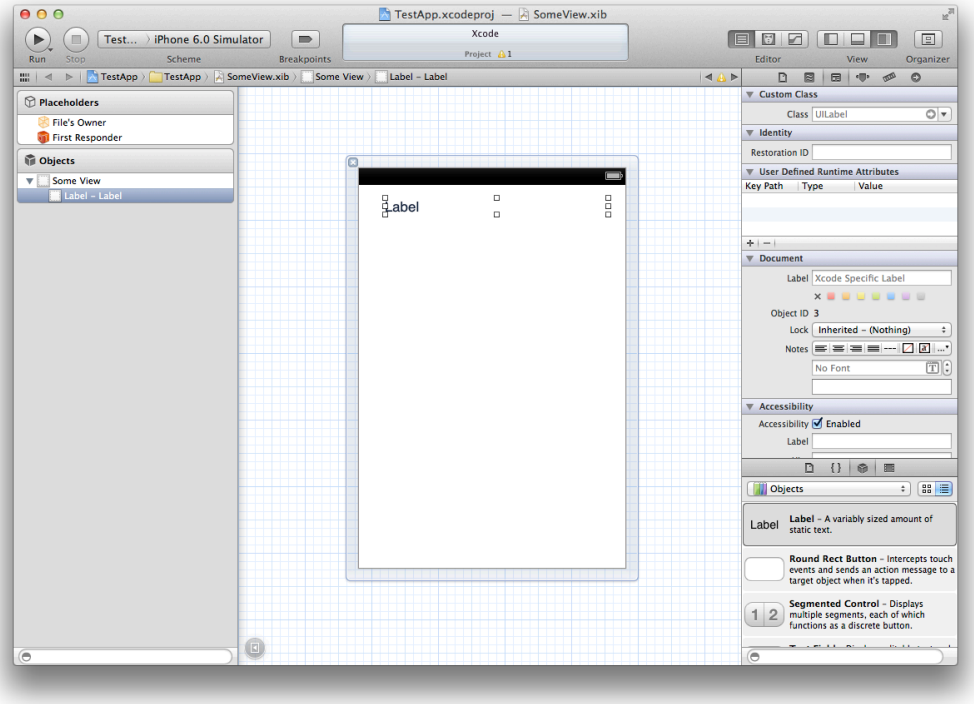


This creates a new file called `SomeView.xib`.

Creating the UI in Interface Builder

Double-click `SomeView.xib` to open it in Xcode's Interface Builder.

At this point controls can be added to the xib. For example, drag a `UILabel` onto the design surface as shown below:



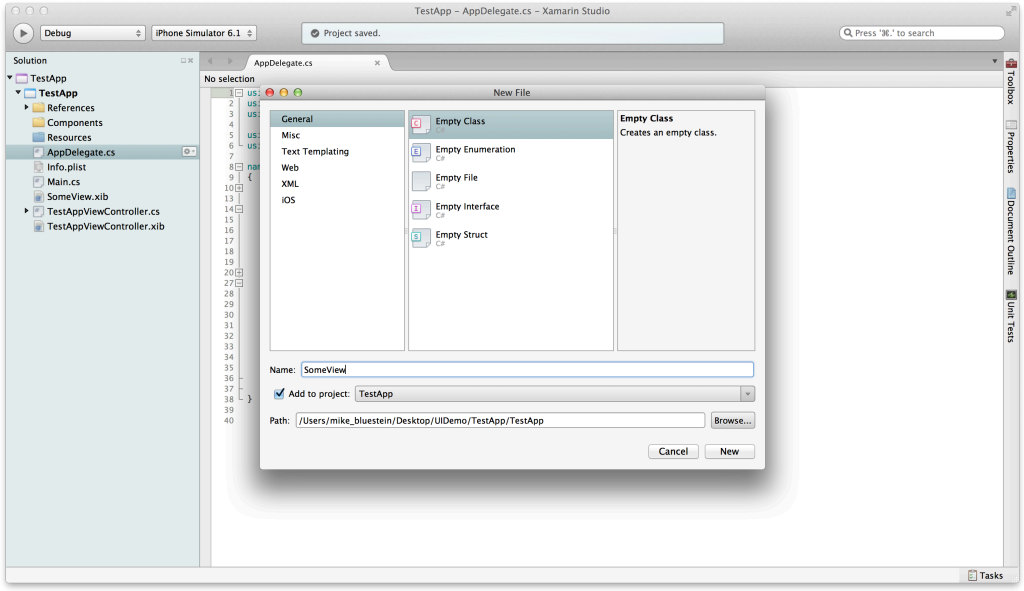
However, since the xib has not been connected to any backing class, we cannot yet add outlets to controls such as the `UILabel` above. This is different than the ViewController template, whose xib file comes already connected to a backing

class that Xamarin Studio creates. In that case, the class is a `UIViewController`. In the case of the view template we are working with, we'll want to connect the xib to a `UIView` class that we must create. The result will be a reusable view that can be included in any view hierarchy as with any other view. The only difference will be that the view's UI will be designed entirely in Interface Builder.

Connecting a Class to the Xib

Let's create a class in Xamarin Studio that we will use for this view.

In Xamarin Studio add a new C# class named `SomeView`, as shown below:

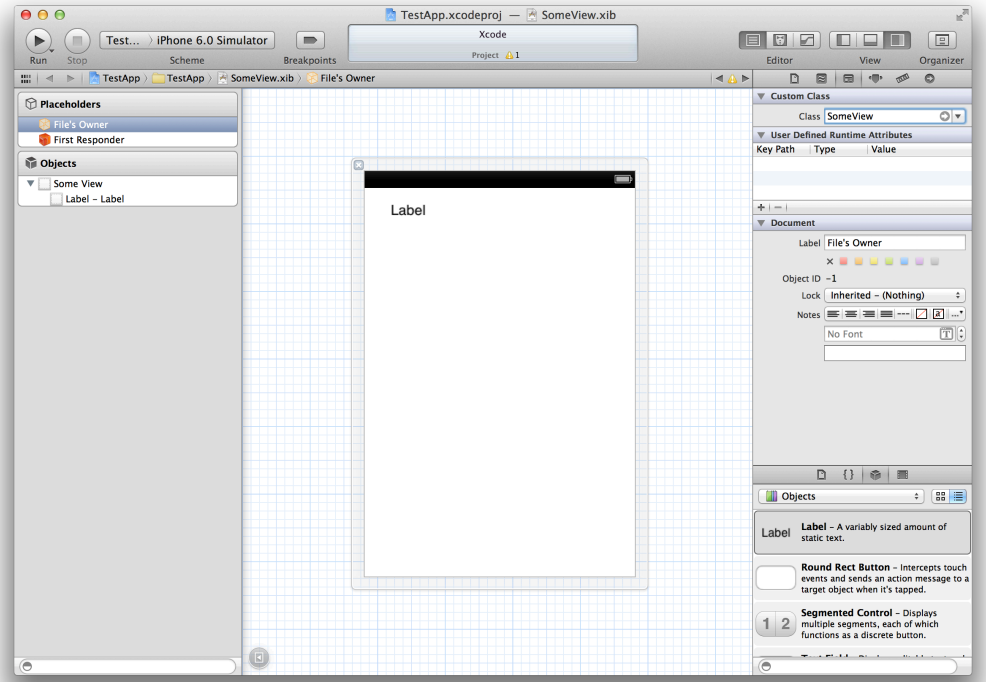


In the code for the class add the following:

```
using System;
using MonoTouch.UIKit;
using MonoTouch.Foundation;

namespace TestApp
{
        [Register("SomeView")]
        public partial class SomeView : UIView
        {
        }
}
```

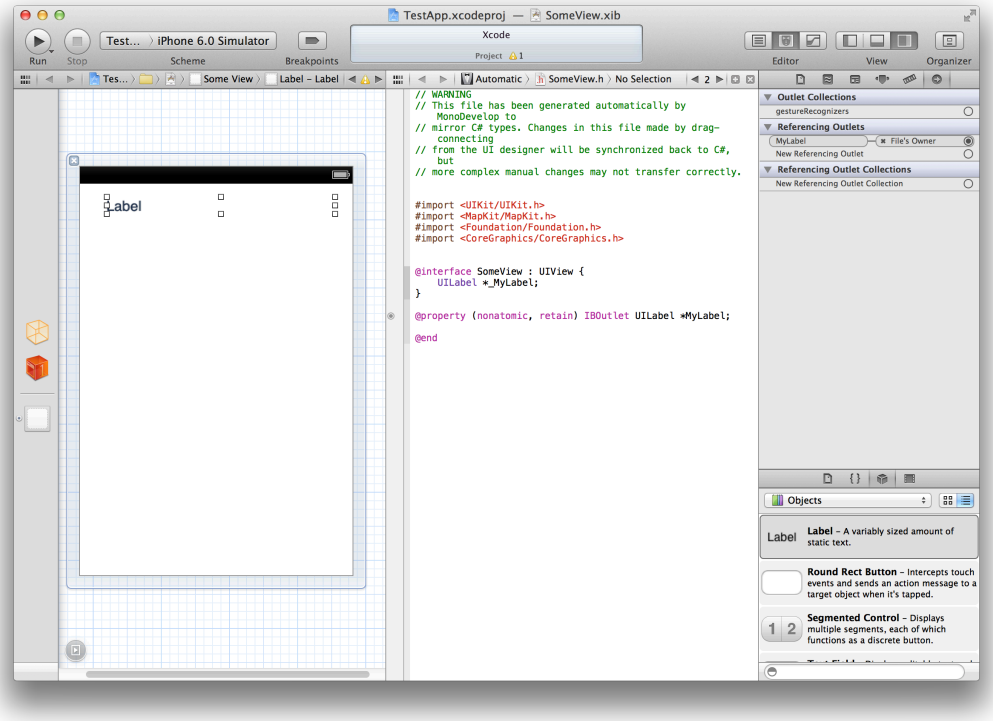
With this class in place we can now connect it in Interface Builder. First select the **File's Owner** under `Placeholder` and set the **Class** to `SomeView` under the **Identity Inspector**:



Next, select the root view, named **Some View** under **Objects** and again set the Class to `SomeView`.

Adding an Outlet

Now we can connect outlets to the controls in the view, such as the label we added earlier, and a partial class will be generated with a property representing the outlet. Create an outlet called `MyLabel` by Ctrl-Dragging into the `SomeView.h` header file. This connects the label to the **File's Owner**, whose backing class is the `SomeView` class we created earlier. This is why we set the File's Owner's **Class** to `SomeView`. Interface Builder, with the **Connections Inspector** selected, should look like the following screenshot after making the connection:



For more information on creating outlets with Interface Builder see Hello, iPhone.

Now that we have everything connected in Interface Builder, we can return to Xamarin Studio to add code to create the View.

Loading the View in Code

In the `SomeView` class add the following namespaces and constructors to load the view from the XIB file we just created:

```
using System;
using MonoTouch.UIKit;
using MonoTouch.Foundation;
using MonoTouch.ObjCRuntime;
using System.Drawing;

namespace TestApp
{
	[Register("SomeView")]
	public partial class SomeView : UIView
	{
		public SomeView(IntPtr h): base(h)
		{
		}

		public SomeView ()
		{
			var arr = NSBundle.MainBundle.LoadNib("SomeView",
this, null);
			var v = Runtime.GetNSObject(arr.ValueAt(0)) as
UIView;
```

```
                        AddSubview(v);

                        MyLabel.Text = "hello from the SomeView class";
                }
        }
}
```

Here we also loaded the view from the XIB and changed the `Text` property of the label we had created an outlet connection for.

The `SomeView` class in now ready to use. In the `TestViewController` class, create an instance of `SomeView` and add it to the view hierarchy, as shown below:

```
SomeView v;

...
public override void ViewDidLoad ()
{
        base.ViewDidLoad ();

        v = new SomeView (){Frame = View.Frame};
        View.AddSubview (v);
}
```

Run the application now and the view from the xib is loaded and displayed:

Creating Custom Views in Code

The example above showed how to create a `UIView` subclass connected to a view designed in Interface Builder. Views can also be created entirely in code by inheriting from `UIView`.

For example the following class creates a custom view containing a label:

```
[Register("CustomView")]
public class CustomView : UIView
{
	UILabel label;

	public CustomView ()
	{
		InitView ();
	}
```

```
        public CustomView (IntPtr p) : base (p)
        {
                InitView ();
        }

        void InitView ()
        {
                BackgroundColor = UIColor.White;

                label = new UILabel (new RectangleF (31, 15, 269, 21)){
                        Text = "some default text"
                };
        }

        public override void Draw (RectangleF rect)
        {
                base.Draw (rect);
                AddSubview (label);
        }
}
```

The default constructor allows the view to be created in code like another other view and added to the view hierarchy as shown below:

```
cv = new CustomView (){Frame = UIScreen.MainScreen.Bounds};
View.AddSubview (cv);
```

RegisterAttribute

Also, a constructor that takes an `IntPtr` is included along with a `Register` attribute. The `Register` attribute makes the class available to the Objective-C runtime, so it can be set in Interface Builder. When Interface Builder unfreezes the class, the additional constructor will be called.

However the view is added to the view hierarchy, either in code or with Interface Builder, the same UI is achieved, as shown:

LayoutSubviews

For fine-grained control of how a view's subviews are laid out, you can override `LayoutSubviews` in your `UIView` subclass. `LayoutSubviews` will be called on the next run loop pass following a call to `SetNeedsLayout`, which will happen implicitly when a view's size changes.

For example, say the view's height and top are reduced at runtime by equal amounts in a gesture recognizer as shown below:

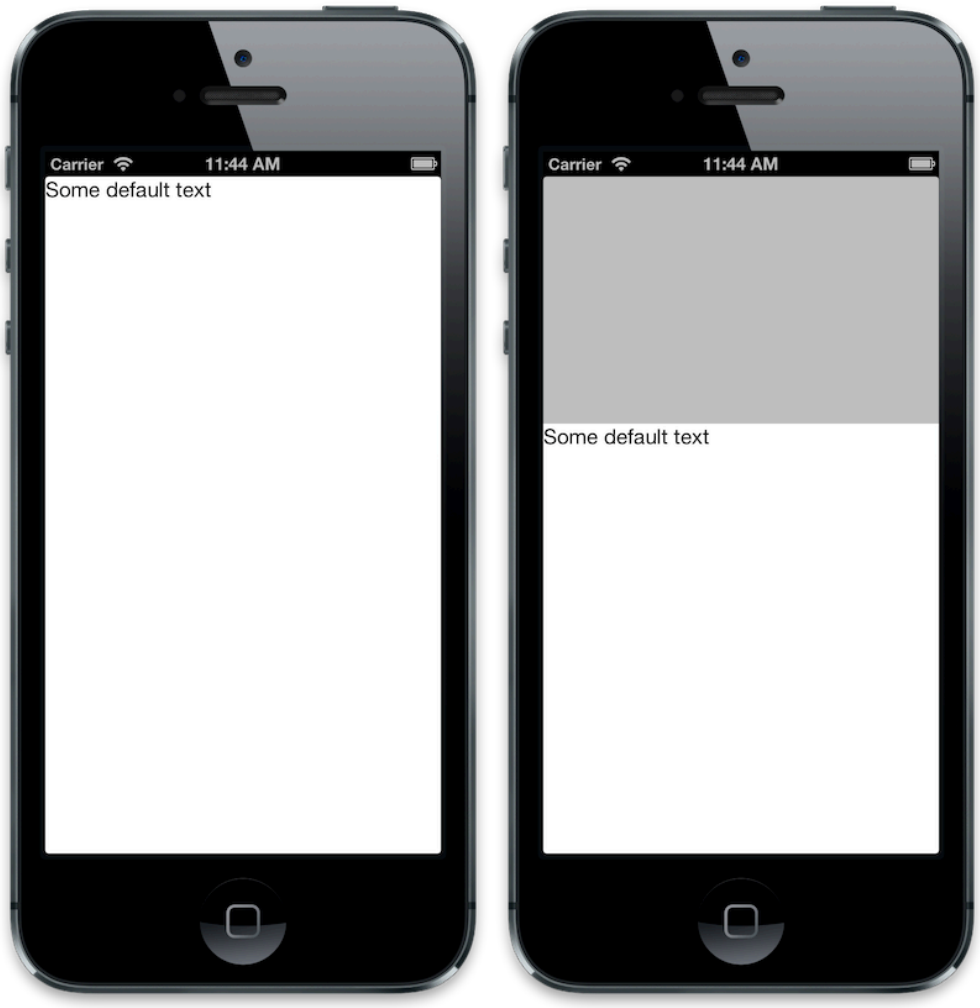
```
cv.AddGestureRecognizer (new UITapGestureRecognizer((g) => {
        cv.Frame = new RectangleF (
                0,
                cv.Frame.Top + 50,
                cv.Frame.Width,
                cv.Frame.Height - 50);
}));
```

If you wanted to programmatically adjust the label's position, you could implement such code in `LayoutSubviews`:

```
public override void LayoutSubviews ()
{
        base.LayoutSubviews ();

        // for programmatic, fine-grained layout of views
        label.Frame = new RectangleF (0, Bounds.Top, 269, 21);
}
```

In this case the code keep the label at the top of the view:

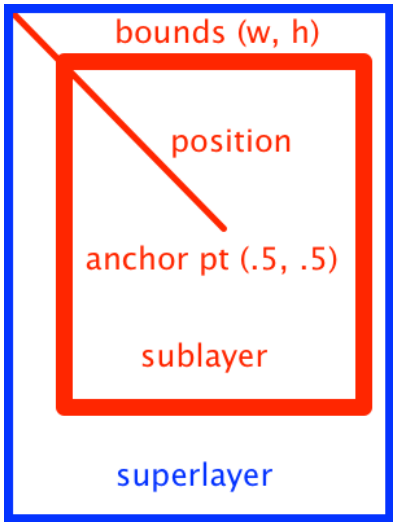


Note this is a simple case to demonstrate the API. The built-in auto-layout (springs & struts) or constraints layout mechanisms should be used whenever possible.

Bounds vs. Frame

The `UIView` class had a `Frame` property, which is used when constructing a view in code to size and position the view. The `Frame` sets the size as a rectangle and the position as a point, calculated within the superview's coordinate system.

When creating a view externally, you would work with the `Frame` property. However, if creating a view by subclassing it, you would work with the `Bounds`, which represents the size of the view within its own coordinate system, as illustrated by the following figure, which shows the geometry of a view's backing layer:



The position defines the location of the layer as measured against an anchor point, a normalized point in layer coordinates, where 0,0 is the upper left and 1,1 the lower right. The bounds defines the width and height.

The reason you would use the bounds when working in a `UIView` subclass, as opposed to the frame, is the bounds is calculated before any transformations are applied. In fact, the frame is actually derived from the bounds and the center point. The rule of thumb is, use the frame when creating a view, and use the bounds when working within a view.

iOS UI Tips

Providing UI Validation

One of the easiest ways to show validation errors in iOS is to highlight the border of the text field. This can be done both after the user finishes entering text, and subsequently changes focus to another control, or via an event on another control, such as a button being tapped.

EditingDidEnd Event

Consider the following code the handles the EditingDidEnd event of a UITextField:

```
// to validate after the user has entered text and moved away from that
// text field, use EditingDidEnd
this.FirstText.EditingDidEnd += (object sender, EventArgs e) => {

        // perform a simple "required" validation
        if ( ((UITextField)sender).Text.Length <= 0 ) {
```

```
                // need to update on the main thread to change the border
color
                InvokeOnMainThread ( () => {
                        this.FirstText.BackgroundColor = UIColor.Yellow;
                        this.FirstText.Layer.BorderColor =
UIColor.Red.CGColor;
                        this.FirstText.Layer.BorderWidth = 3;
                        this.FirstText.Layer.CornerRadius = 5;
                } );
        }
};
```

The `EditingDidEnd` event is raised when a user moves focus from a control, either from selecting the next control, or from clicking the **Done** button. The border properties on the text field can be accessed via the underlying Layer object.

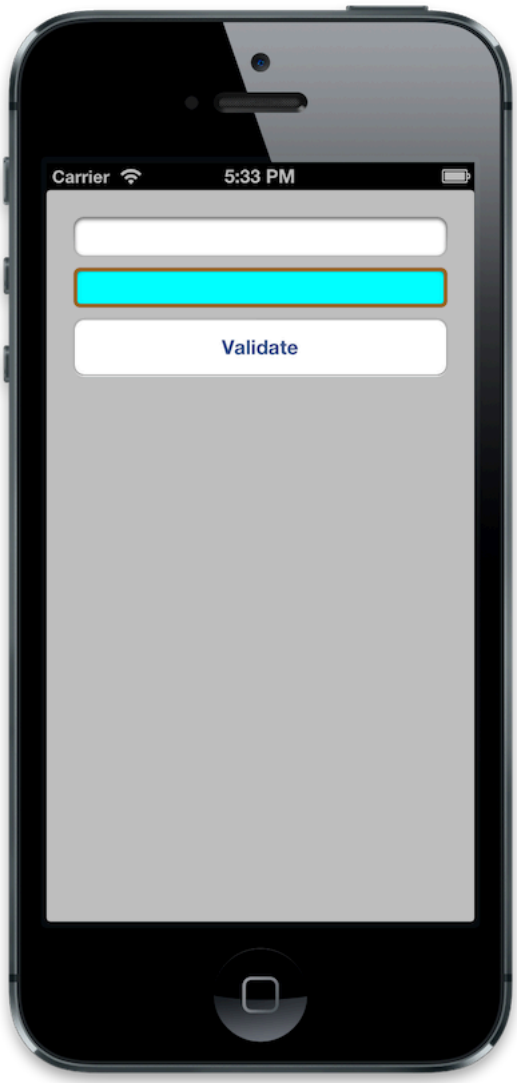
This results in the following:

Validation using a Button

Similarly, handling a button being tapped can also do validation, as shown in the following code:

```
// we can also validate on the touch of a button
this.ValidateButton.TouchUpInside += (object sender, EventArgs e) => {
        if ( this.SecondText.Text.Length <= 0 ) {
                        InvokeOnMainThread ( () => {
                        this.SecondText.BackgroundColor = UIColor.Cyan;
                        this.SecondText.Layer.BorderColor =
UIColor.Brown.CGColor;
                        this.SecondText.Layer.BorderWidth = 3;
                        this.SecondText.Layer.CornerRadius = 5;
                } );
        }
};
```

After the user releases the button, the `UITextField` is highlighted as shown:



FirstResponder and hiding the keyboard

Explain how to shrink a view so it scrolls above the keyboard, and hide the keyboard when done.

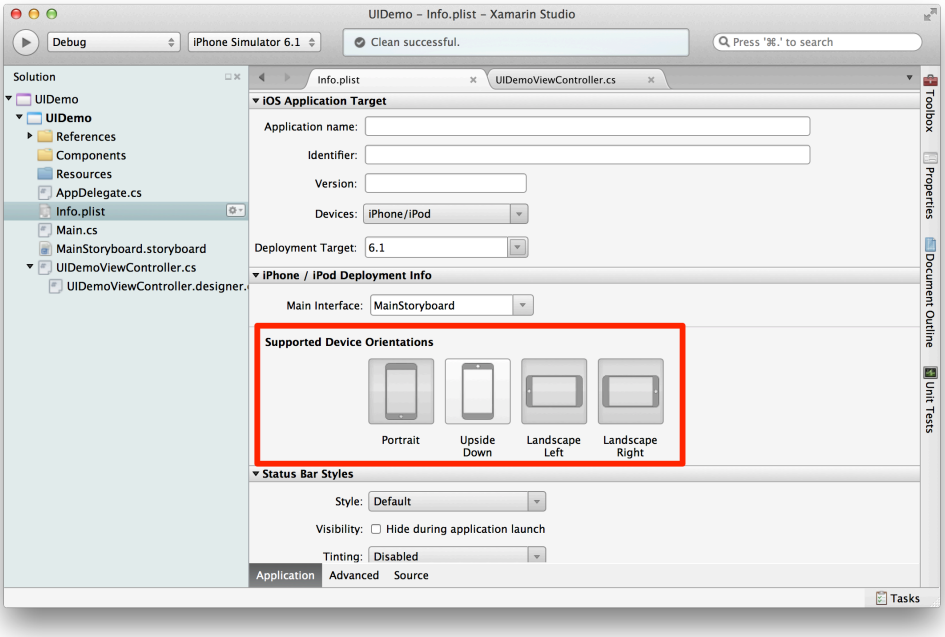
Device Rotation

In iOS 6 applications support multiple orientations by default. iPad supports all orientations by default, whereas iPhone defaults to supporting every orientation except Portrait Upside Down.

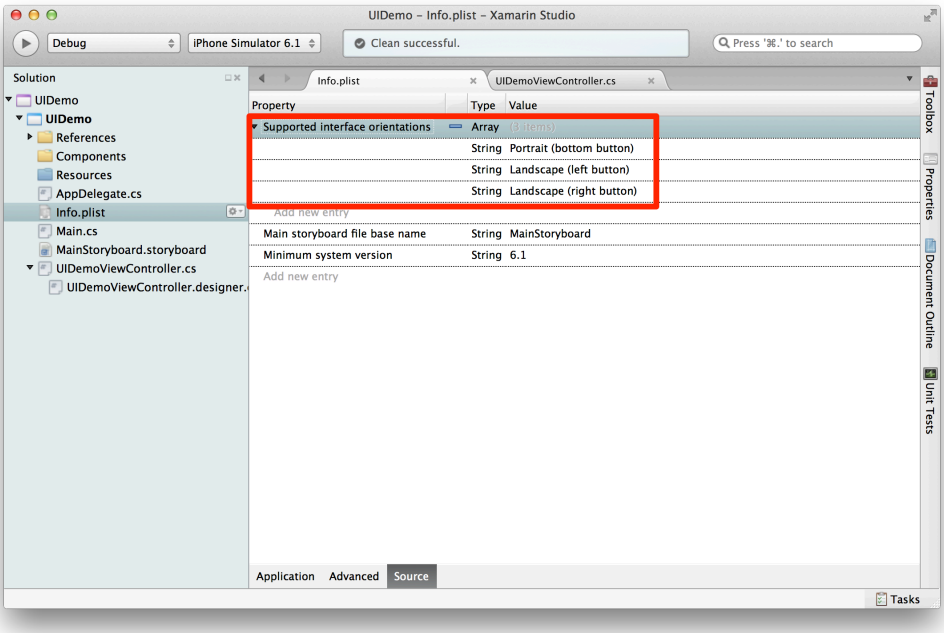
To change the defaults either modify the Info.plist or implement `GetSupportedInterfaceOrientatons`.

Info.plist Supported Interface Orientations

Xamarin Studio allows the supported orientations to be set under the **Supported Device Orientations** section, as shown below:



Each selected orientation appears with a dark gray color. Also, clicking on the **Source** tab shows the actual Info.plist keys and values:



GetSupportedInterfaceOrientations

An application's supported orientations can also be set in code by overriding the `GetSupportedInterfaceOrientations` method of the view controller to return a `UIInterfaceOrientationMask` bit mask.

For example, the following code sets the supported orientations to `LandscapeRight` and `Portrait`:

```
public override UIInterfaceOrientationMask
GetSupportedInterfaceOrientations ()
{
        return UIInterfaceOrientationMask.LandscapeRight |
UIInterfaceOrientationMask.Portrait;
}
```

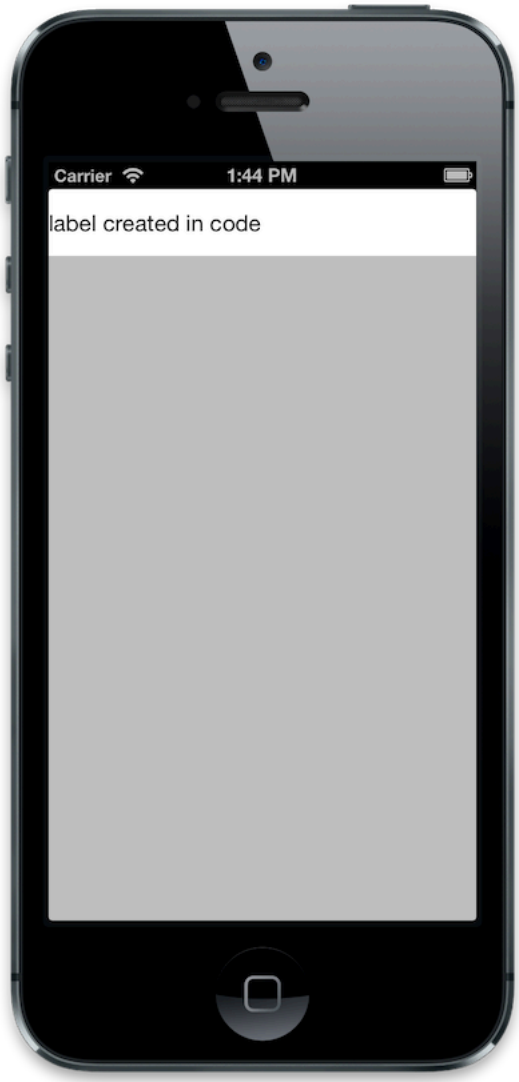
Auto-Resizing

iOS provides auto-resizing masks that can be applied to any view to affect it layout. This is useful for handling simple view layout needs in response to orientation changes.

For example, the following code adds a label that spans the width of the screen:

```
label = new UILabel (new RectangleF (0, 0, View.Frame.Width, 50));
label.Text = "label created in code";
View.AddSubview (label);
```

When the application runs, everything looks fine in portrait orientation:



However, when rotated to landscape, the label does not resize to fill the wider width:

To correct this, the label's `AutoresizingMask` property can be set to `UIViewAutoresizing.FlexibleWidth`:

```
label = new UILabel (new RectangleF (0, 0, View.Frame.Width, 50));
label.Text = "label created in code";
label.AutoresizingMask = UIViewAutoresizing.FlexibleWidth;
View.AddSubview (label);
```

Then when the application runs, the label stretches when in landscape:

Xamarin Android Designer

Designing Views for Multiple Configurations

Android supports a variety of different combinations of devices, orientations, versions, language, themes, etc. To design user interfaces visually for these combinations, the Xamarin.Android designer supports resource qualifiers that can be locked to specific values. Changes introduced for a locked set of qualifiers only take affect when those values are met. For example, a control could be added to a layout when it is in portrait orientation, but not when in landscape.

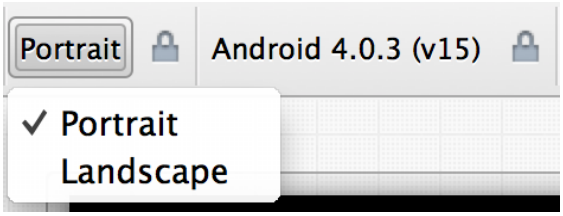
Such differences can lead to conflicts across configurations. If for example some code depended on the control that was only available in portrait, the relevant code could throw an exception due the control not being present in landscape. To help resolve conflicts, the designer includes visual support to alert you in such situations.

Locking Resource Qualifiers

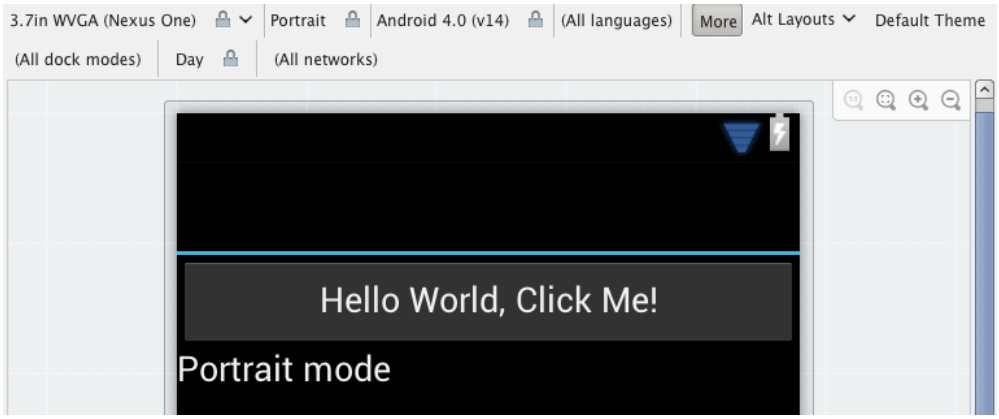
Let's look at a case using the designer to lock resource qualifiers and handle conflicts.

In this example, available in the DesignerApp project, the layout is changed independently for landscape and portrait. In landscape a `TextView` control's text is set to a different value than in portrait, while in portrait the text color of the `TextView` is changed, causing a conflict.

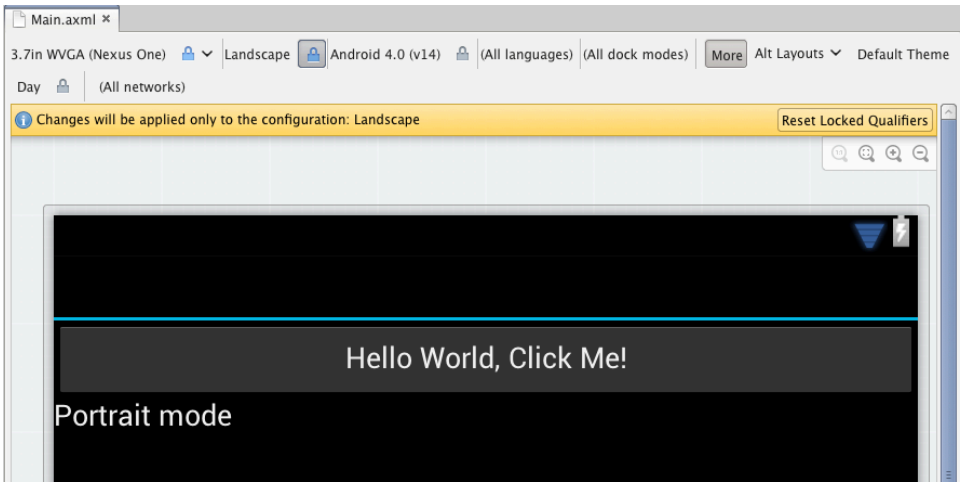
To change the design of the layout depending on a qualifier, or set of qualifiers, the qualifier must be locked. To lock a qualifier, click the **lock** button next to the corresponding qualifier selector. If the selector is bound to several qualifiers, a drop-down menu will display all the qualifiers. The qualifier that is to be locked can be selected from this list, as shown below:



Once a qualifier is locked, changes made in the designer will be specific to that qualifier. For example, consider this layout:

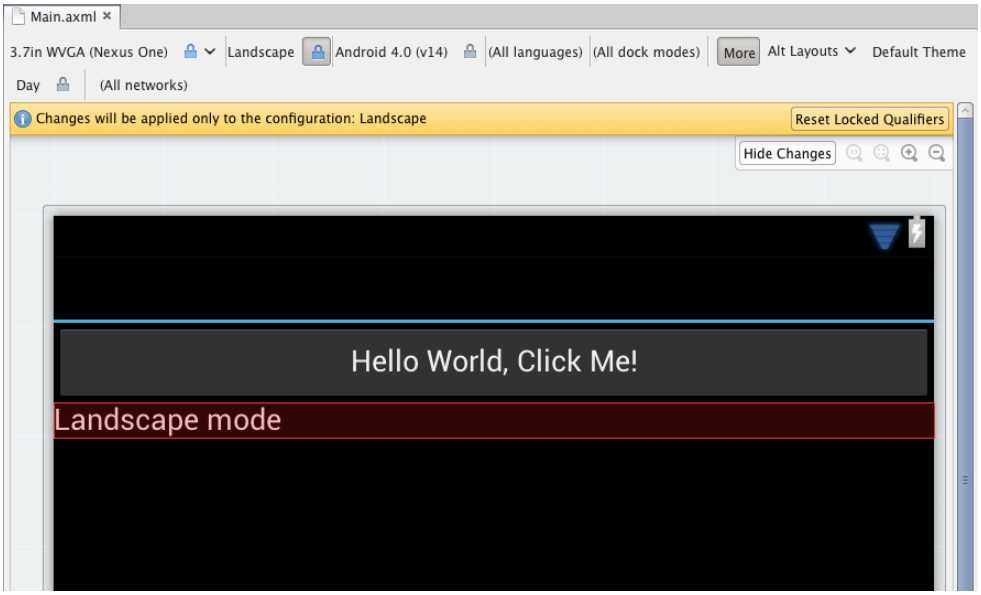


To change the label from "Portrait mode" to "Landscape mode" when the device changes to landscape, select the **Landscape** configuration and click the **lock** button next to it. A message will appear indicating that changes will only be applied for the selected qualifiers, as shown below:



Double-clicking the label and changing the text causes a red frame to appear, indicating that the control has changes that are specific to the current view—landscape in this case. Also, the **Hide Changes** button at the top-right allows

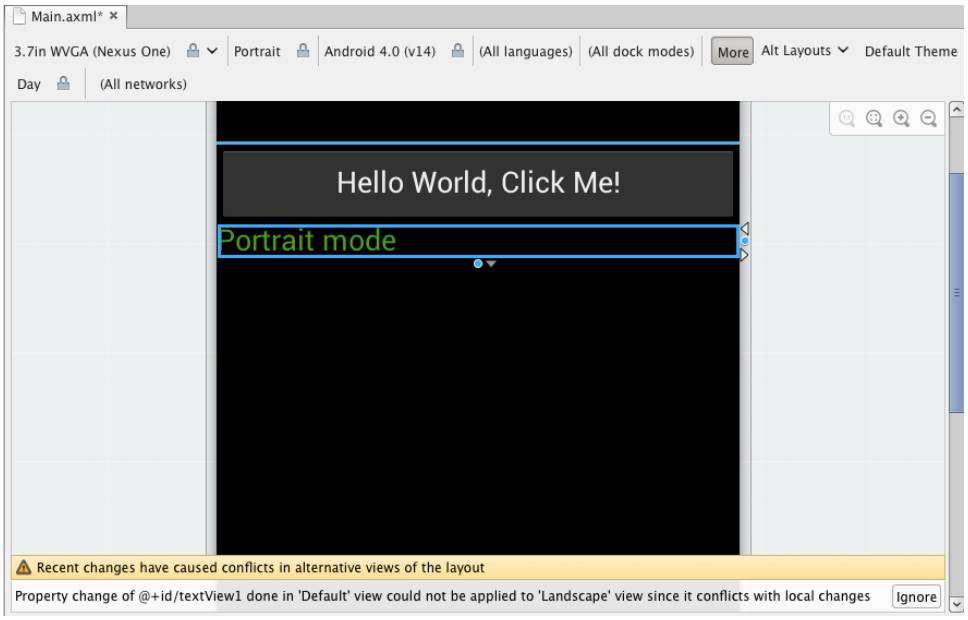
hiding/showing the local change markers. The following screenshot shows the label changed for landscape:



Handling Property Conflicts

The designer automatically propagates changes made in the default layout to all other layout views. However, if you modify a control for which there is a specific version in another view, the designer detects the conflict and reports it.

For example, if you change the color of the label in the default layout to green, a conflict warning is shown at the bottom because there is a local change for landscape that conflicts with this change, as shown:



Clicking the **Ignore** button removes the warning, which is fine in this case since the conflict is only due to a color change. If the conflict existed due to a control

being present in one orientation, but not the other, you would need to make sure the code handles the control not being present for all orientations.

Android UI Tips

Handling Rotation with Layouts

By including files in folders that follow naming conventions, Android automatically loads the appropriate files when the orientation changes. This includes support for:

- *Layout Resources* - Specifying which layout files are inflated for each orientation.
- *Drawable Resources* – Specifying which drawables are loaded for each orientation.

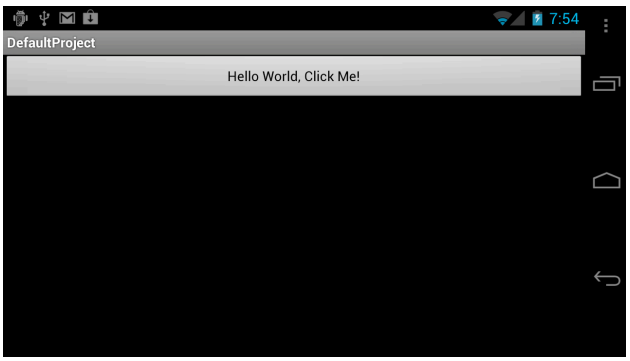
Layout Resources

By default, Android XML (AXML) files included in the **Resources/layout** folder are used for rendering views for an Activity. This folder's resources are used for both portrait and landscape orientation if no additional layout resources are provided specifically for landscape.

Consider the project created by the default project template. The template creates a single Main.axml file in the **Resources/layout** folder. When the Activity's `OnCreate` method is called, it inflates the view defined in Main.axml, which declares a button as shown in the XML below:

```
<?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>
```

If the device is rotated to landscape orientation, the Activity's `OnCreate` method is called again and the same Main.axml file is inflated, as shown in the screenshot below:



Orientation-Specific Layouts

In addition to the layout folder (which defaults to portrait and can also be explicitly named *layout-port* by including a folder named `layout-land)`, an application can define the views it needs when in landscape without any code changes.

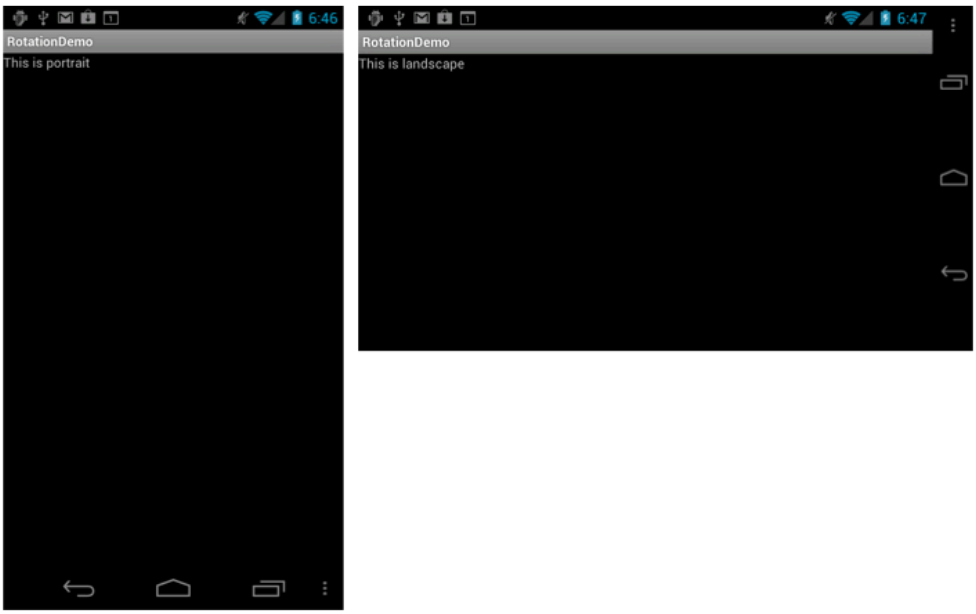
Say the Main.axml file contained the following XML (the code for the remainder of this section is available in the RotationDemo project):

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent">
      <TextView
              android:text="This is portrait"
              android:layout_height="wrap_content"
              android:layout_width="fill_parent" />
</RelativeLayout>
```

If a folder named layout-land that contains an additional Main.axml file is added to the project, inflating the layout when in landscape will now result in Android loading the newly added Main.axml. Consider the landscape version of the Main.axml file that contains the following code (for simplicity, this XML is similar to the default portrait version of the code, but uses a different string in the `TextView`):

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="fill_parent"
              android:layout_height="fill_parent">
      <TextView
              android:text="This is landscape"
              android:layout_height="wrap_content"
              android:layout_width="fill_parent" />
</RelativeLayout>
```

Running this code and rotating the device from portrait to landscape demonstrates the new XML loading, as shown below:



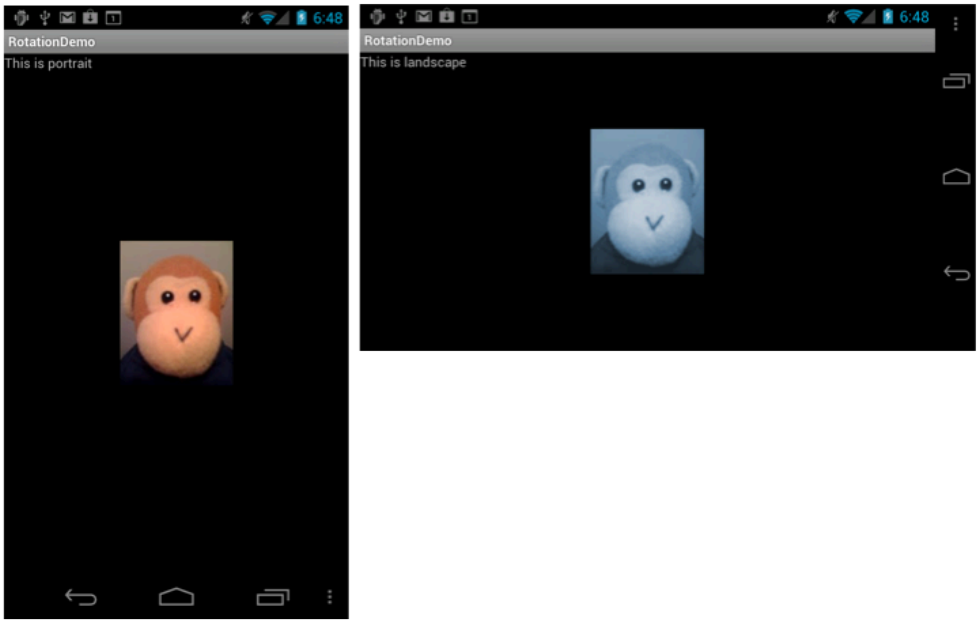
Drawable Resources

During rotation, Android treats drawable resources similarly to layout resources. In this case, the system gets the drawables from the **Resources/drawable** and **Resources/drawable-land** folders, respectively.

For example, say the project includes an image named Monkey.png in the **Resources/drawable** folder, where the drawable is referenced from an `ImageView` in XML like this:

```
<ImageView
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:src="@drawable/monkey"
        android:layout_centerVertical="true"
        android:layout_centerHorizontal="true" />
```

Let's further assume that a different version of Monkey.png is included under **Resources/drawable-land**. Just like with the layout files, when the device is rotated, the drawable changes for the given orientation, as shown below:



Performance

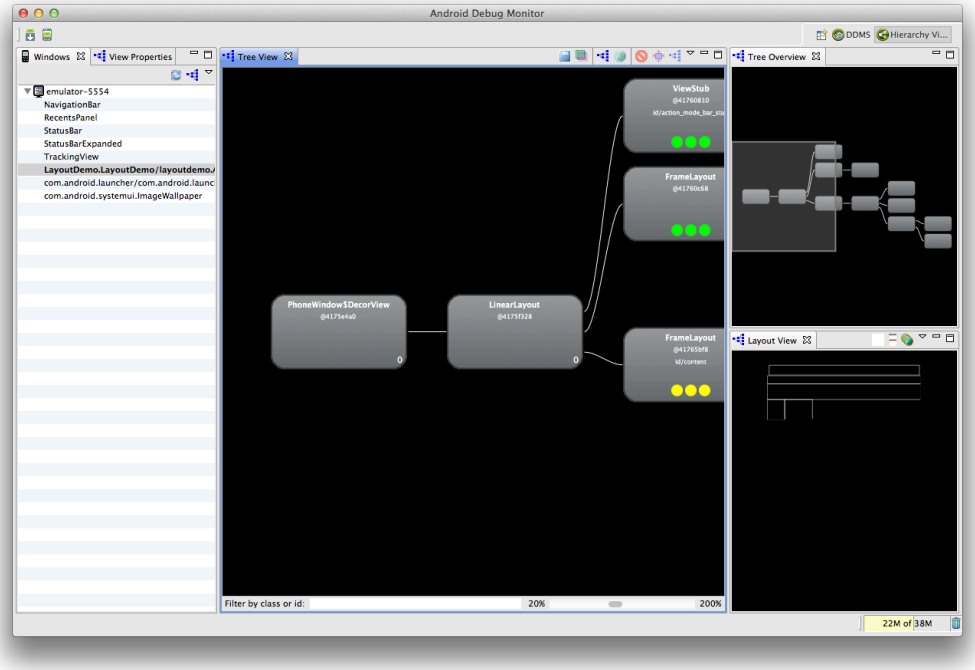
Use RelativeLayout Instead of LinearLayout

`RelativeLayout` displays child views relative to siblings or relative to the parent. The performance benefit of `RelativeLayout` vs. `LinearLayout` arises from the fact that `RelativeLayout` maintains a flatter hierarchy.

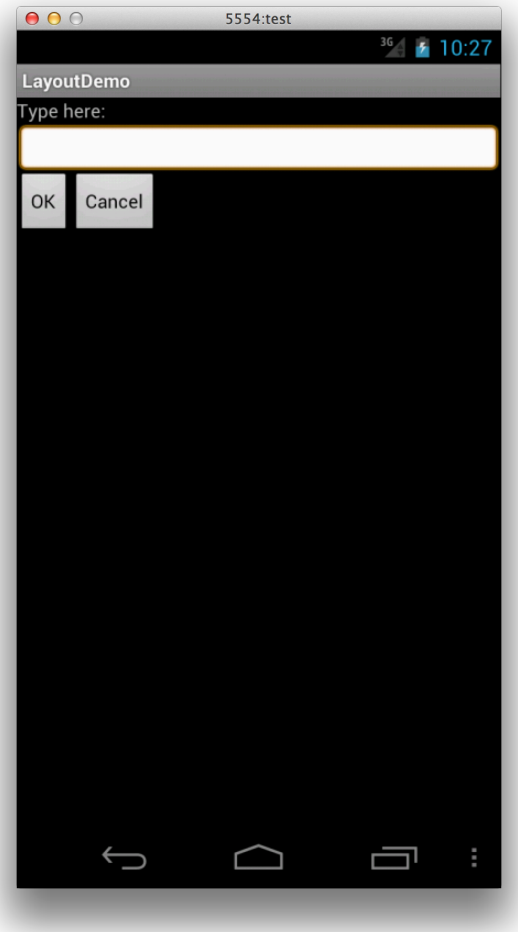
Therefore if nesting several `LinearLayout` instances, replacing them with a single `RelativeLayout` is more efficient. This can be visualized using the *Hierarchy Viewer*.

Using the Hierarchy Viewer

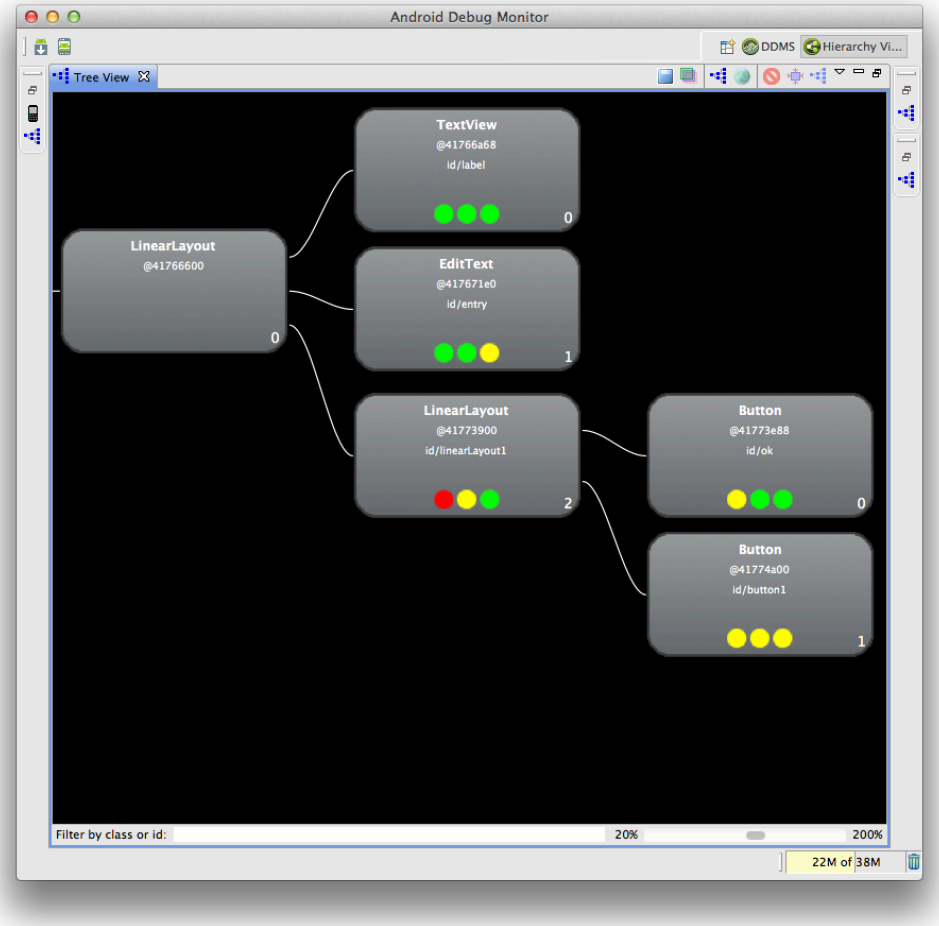
The Hierarchy Viewer, included with the *Android Device Monitor* tool, shown below, provides a tree-like view showing the view hierarchy of a running application.



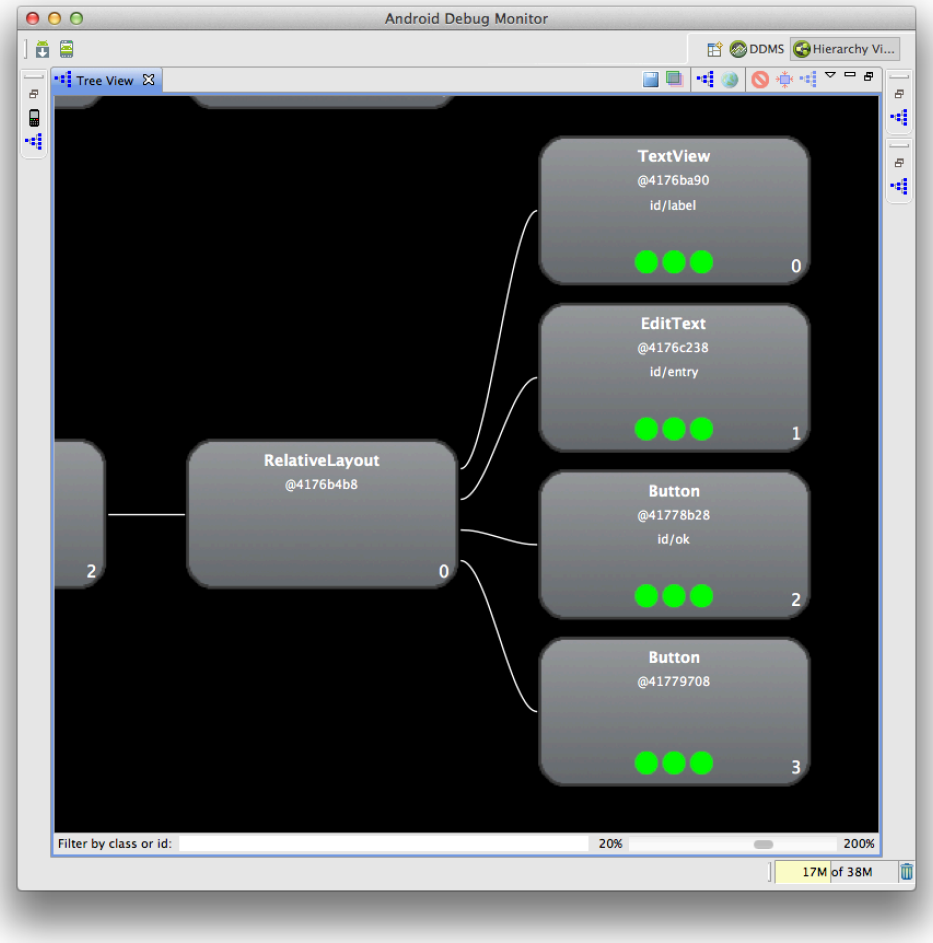
For example say you wanted to generate an application UI like the following (included in the LayoutDemo project):



The following screenshot shows a nested layout created using `LinearLayout` used to achieve this UI:



Now consider the following layout shown in the hierarchy Viewer, which creates the same UI, only this time using a `RelativeLayout`:



As you can see, the `RelativeLayout` creates flatter layout hierarchy, which improves performance because Android can inflate it faster at runtime.

Summary

This chapter covered user interface tools and techniques fro both iOS and Android. For iOS, it examined a variety of the features available via the tooling integration with Xcode and Interface Builder. Then it discussed how to create views using Interface Builder as well as in code, along with useful tip for handling device rotation and performing validation. Next, it covered how to use the Xamarin.Android designer to provide different views for different orientations. Finally, it discussed how to use layout resources to deal with rotation.