Getting Started with Xamarin.Forms

An Introduction to Building Native User Interfaces for Cross-Platform Apps

WHITEPAPER

© 2017 Progress. All Rights Reserved.

Table of Contents

Introduction / 3

Getting to Know Xamarin.Forms / 4

- Xamarin.Forms Is More Than Controls / 4
- Creating a Basic Xamarin.Forms App / 5
- Ideal Use Cases for Xamarin.Forms / 8

Understanding Xamarin.Forms Layouts / 9

- StackLayout / 10
- Grid / 12
- AbsoluteLayout / 15
- RelativeLayout / 18

Using XAML in Xamarin.Forms / 22

- The Benefits of Using XAML / 22
- The Basics / 23
- Properties / 25
- Reacting to Events / 25

Accessing Static Data Using XAML / 26

- StaticExtension Class / 27
- Resource Dictionaries / 30
- Creating Objects From XAML / 32

Accessing Dynamic Data / 35

- Dynamic Resources / 35
- Data Binding / 37

© 2017 Progress. All Rights Reserved.

Introduction

Today's enterprises—especially industry leaders—rely on their mobile applications. These applications can be used for a variety of purposes, from in-field reporting to consumer engagement.

However, despite the widespread proliferation of mobile applications, mobile development remains a challenging prospect due to the number of popular operating systems powering these devices. Android, iOS, Universal Windows Platform (UWP)—they're all unique and require different skills to develop, deploy and maintain applications. At best, building a cross-platform mobile application is an exercise in tedium. At worst, it's a development nightmare. This is where Xamarin comes in. The Xamarin framework enables developers to build cross-platform mobile applications (with native performance) using a shared codebase—C#. However, that shared codebase only extends to the application logic side of things. Traditional Xamarin.iOS, Xamarin.Android, and Xamarin.UWP development still require developers to write the user interface from each other—and that is no small task.

To that end, Xamarin.Forms can help developers simplify this process and save significant development time. The Xamarin.Forms API abstracts the user interface of each platform—the individual operating system controls and navigation metaphors—into a common layer, which can be used to build applications for iOS, Android, and UWP with a both a shared application layer and user interface layer.

However, this only scratches the surface of Xamarin.Forms—it's so much more than a framework for building user interfaces across multiple mobile platforms. This ebook will provide an overview of both the cross-platform user interface elements as well as some other features that make Xamarin.Forms a full-fledged framework worth considering for future mobile app.

© 2017 Progress. All Rights Reserved.

Getting to Know Xamarin.Forms

The official Xamarin website defines Xamarin.Forms as "a cross-platform UI toolkit that allows developers to easily create native user interface layouts that can be shared across Android, iOS, and Windows Phone."

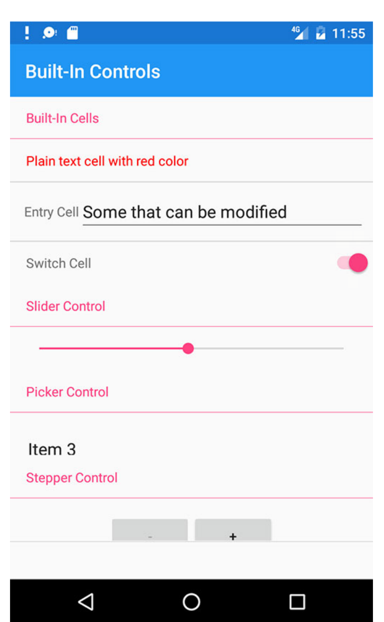
However, when looking at that definition, it's critical that developers don't simply focus on the term "UI" and think on-screen controls. Instead, they should focus on the "toolkit" aspect—Xamarin.Forms offers so much more in addition to user interface controls that work natively across platforms.

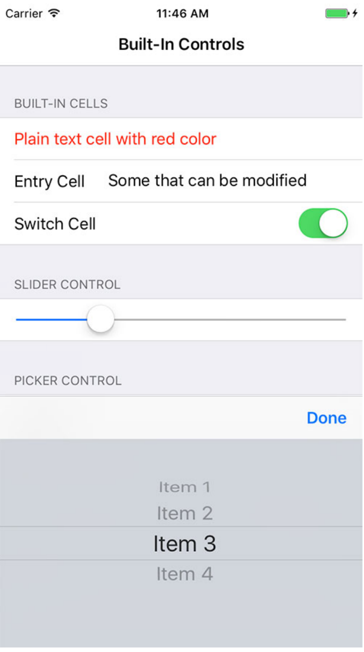
Xamarin.Forms will emit a 100% native iOS, Android or UWP app. In fact, the starting point of any Xamarin.Forms app is within one of those platform projects. However, that's as far as the platform-specific code needs to go. The rest of the code can all be written in one of the layers that are shared amongst all the applications.

Xamarin.Forms Is More Than Controls

Xamarin.Forms provides more than 20 cross-platform user interface controls, and each of these controls has an API specific to Xamarin.Forms that is emitted as its native iOS, Android or UWP counterpart. In other words, a Xamarin.Forms Label control will be emitted as an iOS UILabel.

Some of the built-in Xamarin.Forms controls, as natively rendered on iOS and Android, can be seen below.





© 2017 Progress. All Rights Reserved.

But Xamarin.Forms is so much more than just controls. Xamarin.Forms also provides:

- Several different page layouts, including a navigation page which controls a navigation stack of other pages, a tabbed page containing other pages accessed via tabs and a master detail page.
- Means to layout the controls within the pages via what are called Layouts, and there are several of those including Stack, Grid, Absolute, and Relative.
- A binding engine, so a property in a class can be "bound" to a property on a control—like the Text property on a Label. This alone greatly speeds up development time.
- Messaging Center, which is a messaging service. This enables various classes and components to communicate without knowing anything about each other.
- Numerous utilities, which facilitate access to the underlying platform projects to bring platform-specific functionality into the core or shared Xamarin.Forms project. The Dependency Service is one such utility. This enables the creation of functionality in a class within a platform project, and lets Xamarin.Forms find that class in the shared project.
- Effects, which are a means by which developers can create small platform-specific user interface tweaks to controls and have them applied in the shared project.
- Custom Renderers, which enable developers to take full control of how a control renders itself within Xamarin.Forms. This means developers can add whatever additional appearance or functionality you may need.

With the latest version of Xamarin.Forms, you can even directly add in controls that are only supported on one platform (such as Android floating action buttons) directly into XAML files.

Creating a Basic Xamarin.Forms App

Now that we've gone over the strengths of Xamarin.Forms, let's create a basic application. In this practice app, we will demonstrate several of the concepts discussed above—namely navigation pages, a grid layout, a couple of the built-in controls and data-binding. This app is intended to be a "Hello World" for each of those concepts and will demonstrate that Xamarin. Forms is more than just UI controls.

© 2017 Progress. All Rights Reserved.

We will be using XAML to create the user interface of this project.

As part of this app, we want to make sure the pages appear within a navigation hierarchy. So, new pages get added to the stack, and a back button press will remove them.

Within a new project, open App.xaml.cs and set the *MainPage* to be equal to a *NavigationPage*:

```
MainPage = new NavigationPage(new OverviewDemoPage());
```

OverviewDemoPage is the name of the class that will contain the main content of our app, and by virtue of being passed into the NavigationPage's constructor, it will serve as the root of the navigation hierarchy.

The content of OverviewDemoPage.xaml looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<ContentPage
        xmlns="http://xamarin.com/schemas/2014/forms"
        xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
        xmlns:local="clr-namespace:OverviewDemo"
        x:Class="OverviewDemo.OverviewDemoPage"
        Title="Forms Overview">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Label BindingContext="{x:Reference theEntry}" Text="{Binding Text}"
            VerticalOptions="Center" HorizontalOptions="Center"
            Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2" />
        <Entry x:Name="theEntry" Text="Hi" VerticalOptions="Center" HorizontalOptions="
FillAndExpand"
            Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2" />
        <Button Text="Button 1" Grid.Row="2" Grid.Column="0" Clicked="ButtonOne_
Clicked"/>
        <Button Text="Button 2" Grid.Row="2" Grid.Column="1" Clicked="ButtonTwo_
Clicked"/>

    </Grid>
</ContentPage>
```

© 2017 Progress. All Rights Reserved.

There are several things going on here.

First is the Grid. It is defined as two columns and three rows. The columns are defined each to take up an equal amount of space, while the bottom row is defined to take up exactly the amount of space it needs to house its contents, then the two rows on top of it will equally split the rest.

Each of the controls that follow position themselves in the *Grid* using *Grid.Row* or *Grid.Column*.

The *Entry* and the *Label* will take up the first and second row of the Grid respectively, but notice that they are also defined to span both columns of the Grid so that they can center themselves on screen.

A very interesting thing is the data binding setup between the *Label* and the *Entry* control. The *Label* has a property called *BindingContext*, which is the source of any data it is bound to. In this case, it is bound to the control named *theEntry*. Then the *Text* property on the Label is using that binding to get at a property in the source (*theEntry* control) by the name of *Text*.

So *Label.Text* is bound to whatever appears in *Entry.Text*, which is a helpful way to introduce automatic updating of controls without having to write a lot of event handlers (and it works with regular classes as well, not only controls).

Finally, the two buttons will occupy the bottom row of the grid and each have half of the screen. When clicked, they will invoke their respective event handlers, defined in *OverviewDemoPage.xaml.cs* as:

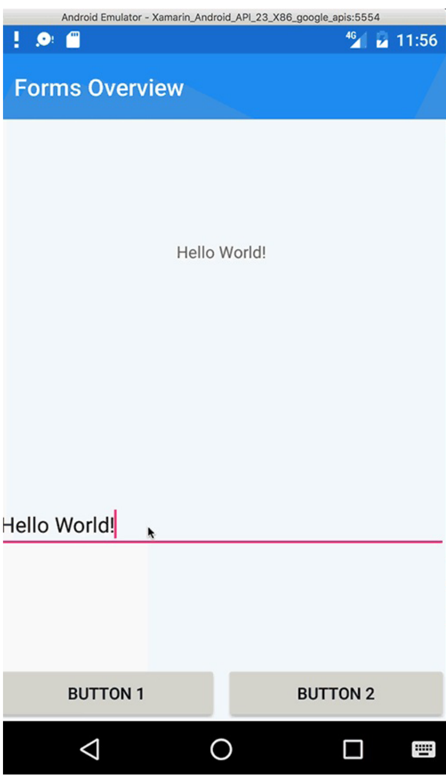
```
async void ButtonOne_Clicked(object sender, EventArgs e)
{
    await Navigation.PushAsync(new ChildPage(), true);
}

async void ButtonTwo_Clicked(object sender, EventArgs e)
{
    await App.Current.MainPage.DisplayAlert(“Alert”, “Button Two Clicked”, “OK”);
}
```

© 2017 Progress. All Rights Reserved.

The ButtonOne_Clicked function will push a new page onto the stack, while the ButtonTwo_ Clickedfunction will display an alert.

Obviously, this is a very simple app, but one that I think shows off the power of Xamarin.Forms with a minimal amount of code. Within this basic app, we utilize navigation hierarchy, grid layouts, and data binding.

This is the app running on Android. Notice the data-bound `Label` control and how it changes immediately with the Entry control. The navigation stack and alert prompt are being demonstrated by the *Buttons*.



Ideal Use Cases for Xamarin.Forms

As its name implies, Xamarin.Forms excels at collecting data, or when pages are laid out in forms. However, it can be used in many more scenarios.

With the easy access to the platform projects via the Dependency Service, Effects, Custom Renderers, or embedding native views directly into the XAML files, it is easier than ever to build platform-specific, feature-rich application with Xamarin.Forms. In that sense, the easier question to answer may not be when to use Xamarin.Forms, but when not to use it.

© 2017 Progress. All Rights Reserved.

Generally speaking, anytime developers need a highly customized app for a single platform, then the traditional Xamarin route is the way to go. For example, if developers need a highly customized user interface or functionality that is only provided by a specific operating system, it may not make sense to use Xamarin.Forms.

Along with that, apps that require complex animations or have complex, platform-specific requirements (like games) may not the best fit either. That said, there are libraries built for game development such as CocosSharp and UrhoSharp that are accessible from Xamarin.Forms—it's just not as optimal.

However, if developers want to produce an identical (or nearly identical) app that functions the same on all mobile platforms, Xamarin.Forms should be a prime candidate.

Understanding Xamarin.Forms Layouts

In the first chapter, we introduced Xamarin.Forms and discussed some of the core strengths and weaknesses associated with the toolkit. While Xamarin.Forms is certainly more than just UI controls, that should not undercut the fact that the framework does indeed provide a rich set of controls for building sophisticated user interfaces from a single code base.

In this chapter, we will discuss a fundamental building block that can be used to arrange these controls on-screen—Layouts. A Xamarin.Forms Layout is a subtype of the View class, and its purpose is to organize other Views placed within it (which will be referred to as controls throughout this ebook), in a specified manner.

The manner in which the Layout organizes those controls is determined by the Layout's type. There are four common layouts that are used regularly when building user interfaces with Xamarin.Forms. Each layout has a unique purpose with useful properties to arrange child controls on-screen. In order of increasing complexity, they are:

- StackLayout
- GridLayout
- AbsoluteLayout
- RelativeLayout

© 2017 Progress. All Rights Reserved.

StackLayout

The *StackLayout* is the simplest of all the layouts. It arranges child controls on either a vertical or a horizontal axis. Only a single control can occupy any row or column in the one-dimensional grid this layout provides.

Properties

The following properties are used by the *StackLayout* to help it arrange its controls:

- Orientation: Indicates which direction the *StackLayout* should layout the controls. Values can either be *Vertical* or *Horizontal*.
- Spacing: A *Double* to indicate how much space should be in-between each control in the *StackLayout*.

Tips

- Pay attention to each control's *LayoutOptions* within the *StackLayout*, as that will determine the position and size of the control.
- *StackLayouts* are best for simple layouts. Don't try to create complex layouts by nesting several *StackLayouts* together, one of the other layouts will probably be better suited.
- Do not host only a single control in a *StackLayout*.

Example

All of the examples that we'll go through today will have BoxViews in the layout. BoxViews are great for this because they can show the entire space allocated to a particular View with a color.

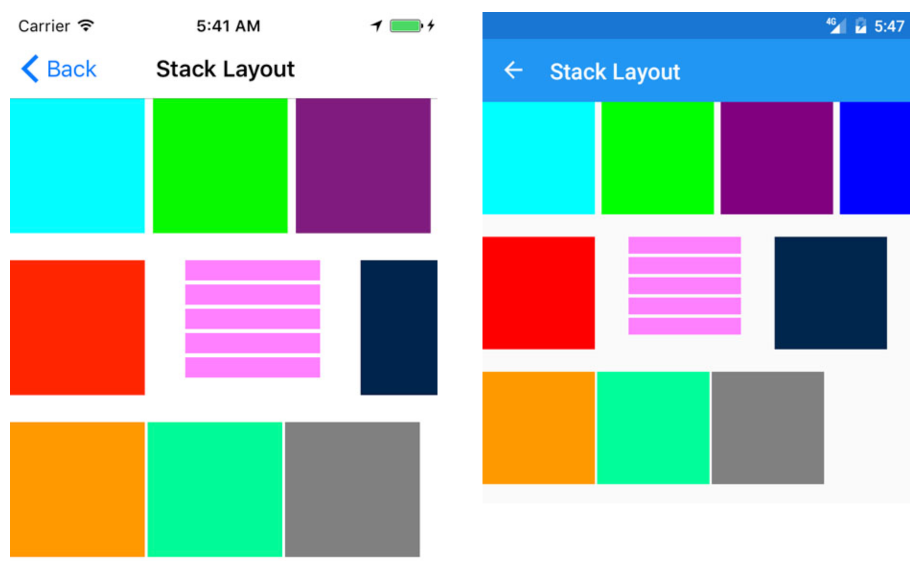
As was the case in the first chapter, all examples will be in XAML. However, everything you see can be created in C# as well if you prefer.

© 2017 Progress. All Rights Reserved.

This first example demonstrates a series of StackLayouts, using both the Orientation and Spacing properties.

```
<StackLayout Orientation="Vertical" Spacing="20">
  <StackLayout Orientation="Horizontal">
    <BoxView Color="Aqua" HeightRequest="100" WidthRequest="100" />
    <BoxView Color="Lime" WidthRequest="100" />
    <BoxView Color="Purple" WidthRequest="100" />
    <BoxView Color="Blue" WidthRequest="100" />
  </StackLayout>
  <StackLayout Orientation="Horizontal" Spacing="30">
    <BoxView Color="Red" HeightRequest="100" WidthRequest="100"/>
    <StackLayout Orientation="Vertical" Spacing="3">
      <BoxView Color="#FF80FF" HeightRequest="15" WidthRequest="100" />
      <BoxView Color="#FF80FF" HeightRequest="15"  />
      <BoxView Color="#FF80FF" HeightRequest="15"  />
      <BoxView Color="#FF80FF" HeightRequest="15"  />
      <BoxView Color="#FF80FF" HeightRequest="15"  />
    </StackLayout>
    <BoxView Color="#00264d" HeightRequest="100" WidthRequest="100"/>
  </StackLayout>
  <StackLayout Orientation="Horizontal" Spacing="2">
    <BoxView Color="#FF9900" HeightRequest="100" WidthRequest="100" />
    <BoxView Color="#00FD99" HeightRequest="100" WidthRequest="100"/>
    <BoxView Color="Gray" HeightRequest="100" WidthRequest="100"/>
  </StackLayout>
</StackLayout>
```

That code produces the following screen:



© 2017 Progress. All Rights Reserved.

Notice that there is an overall vertical layout with a spacing of 20 between the rows. Inside, there are three horizontal layouts, each with various spacing. Finally, the middle horizontal *StackLayout* has a vertical *StackLayout* nested within it.

Grid

Whereas a *StackLayout* arranges controls in one dimension, a *Grid* arranges its controls in a two-dimensional grid pattern.

A defining characteristic of a *Grid* is that its rows and columns can have their heights and widths set to varying values. They could contain an absolute value, be proportionally assigned or assigned by the height and width of the control they contain.

Properties

- ColumnSpacing: A *Double* that specifies the amount of space between the columns in the *Grid*.
- RowSpacing: A *Double* that specifies the amount of space between the rows.
- ColumnDefinitions: A collection of *ColumnDefinition* objects, each *ColumnDefinition* has one property—*Width*. See the following section for information on how to specify.
- RowDefinitions: A collection of *RowDefinition* objects, each *RowDefinition* has one property, *Height*. See the following section for information on how to specify.

Specifying Height and Width

Below is a quick example of defining three rows and specifying three types of heights on those rows.

```
<Grid.RowDefinitions>
  <RowDefinition Height="Auto" />
  <RowDefinition Height="*" />
  <RowDefinition Height="100" />
</Grid.RowDefinitions>
```

© 2017 Progress. All Rights Reserved.

There are three possible values that can go into the *Height* or *Width* properties.

- Auto: The word "Auto" indicates the row or column should size itself to fit its contents.

- (Asterisk): An asterisk indicates the row or column should take up the remaining space on screen, proportionally. Several of these asterisk indicators can appear within the same Grid, meaning that each row or column will be proportionally sized. A number can also be used to modify the asterisk, so *1.5** will get 1.5 times the space as *.

- Absolute: This is the specific value the height or width for the row or column and is expressed as a number.

Attached Properties

Before proceeding, a word must be said about attached properties. Attached properties are a means by which a property that belongs to one object is assigned a value from the XAML definition of another object. In other words, the properties below are all Grid properties, but their value is being set from within the child controls in the *Grid*.

In the example:

```
<BoxView Grid.Row="1" Grid.Column="2" Color="Red" />
```

Grid.Row and Grid.Column are attached properties.

The attached properties for the grid are:

- Column: Zero based index of the column the control resides in.
- Row: Zero based index of the row the control resides in.
- ColumnSpan: The number of columns the control crosses.
- RowSpan: The number of rows the control crosses.

Tips

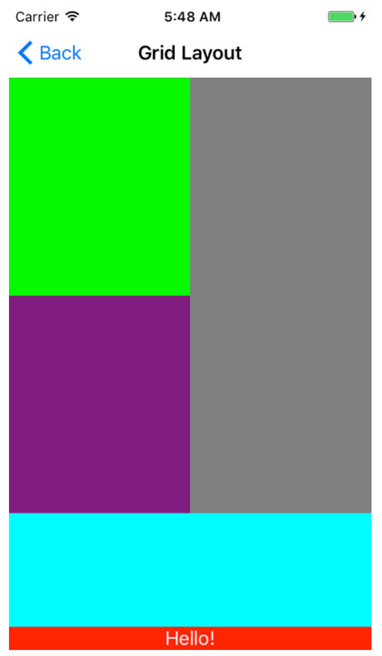
- Do not use a grid to layout controls when a StackLayout will suffice.
- Use the * based sizing of rows and columns over Auto when possible.
- Grids are great for layering controls over the top of one another.

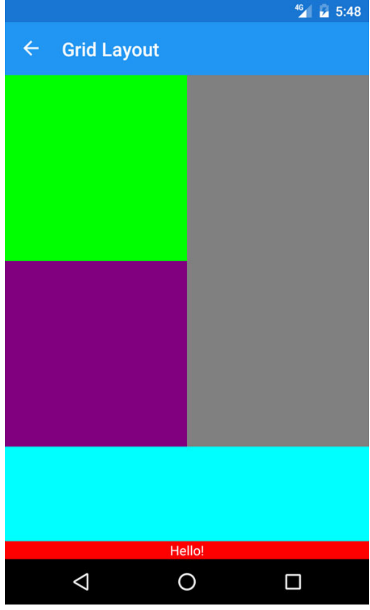
© 2017 Progress. All Rights Reserved.

Example

```
<Grid RowSpacing="0" ColumnSpacing="0">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
    <RowDefinition Height="100" />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
  <BoxView Color="Lime" Grid.Column="0" Grid.Row="0" />
  <BoxView Color="Purple" Grid.Column="0" Grid.Row="1" />
  <BoxView Color="Aqua" Grid.Column="0" Grid.Row="2"
    Grid.ColumnSpan="2" />
  <Label Text="Hello!" Grid.Column="0" Grid.Row="3" Grid.ColumnSpan="2"
    HorizontalOptions="FillAndExpand" HorizontalTextAlignment="Center"
    BackgroundColor="Red" TextColor="White" />
  <BoxView Color="Gray" Grid.Column="1" Grid.Row="0" Grid.RowSpan="2" />
</Grid>
```

The above code defines a *Grid* with two columns and four rows. The gray *BoxView* on the right side spans multiple rows. The red and aqua rows at the bottom span multiple columns. The bottom, red-colored row is sized so its height fits only the *Label* placed within it. The aqua-colored row above that is sized to be exactly 100 units. The other two rows then split the remaining space.





© 2017 Progress. All Rights Reserved.

AbsoluteLayout

As its name suggests, an *AbsoluteLayout* arranges controls on screen exactly as the X, Y, height and width are specified, with the top left corner of the layout serving as the X=0 and Y=0 point. In addition to the exact specifications, with the *AbsoluteLayout*, you can also proportionally specify the control's position, height and width based on the overall layout size. In other words, you can specify that a control should appear 20% across and 50% down.

Attached Properties

Attached properties are again used so the control can specify where it is positioned within an *AbsoluteLayout*.

- LayoutBounds: A comma-delimited string of numbers. The numbers may represent an absolute or proportional value (specified as between 0.0 and 1.0). Each position of the string represents the following:

  - X position of the control.
  - Y position.
  - Width.
  - Height.

- LayoutFlags: LayoutFlags specify which, if any, of the bounds of the control should be proportionally allocated. This property can be set to the following values:

  - None: All values in *LayoutBounds* are interpreted as absolute.
  - All: All values are interpreted as proportional.
  - XProportional: The X value is interpreted as proportional the rest as absolute.
  - YProportional: The Y value is proportional, the rest absolute.
  - WidthProportional: The width is proportional, the rest absolute.
  - HeightProportional: The height is proportional, the rest absolute.
  - PositionProportional: X and Y are proportional, height and width absolute.
  - SizeProportional: Height and width are proportional, X and Y are absolute.

Tips

- You can mix both static and proportional *LayoutBounds* values for a single control.

- *LayoutFlags* can be combined via a comma separated list in XAML.

- Be aware that it is possible to position controls off the screen when using proportional values with an *AbsoluteLayout*, you need to take the size of the control into account.

© 2017 Progress. All Rights Reserved.

- *AbsoluteLayout* is a great layout when overlays across the entire screen need to be presented.

- *AbsoluteLayout* also is worth consideration when animation of controls across the screen are required.

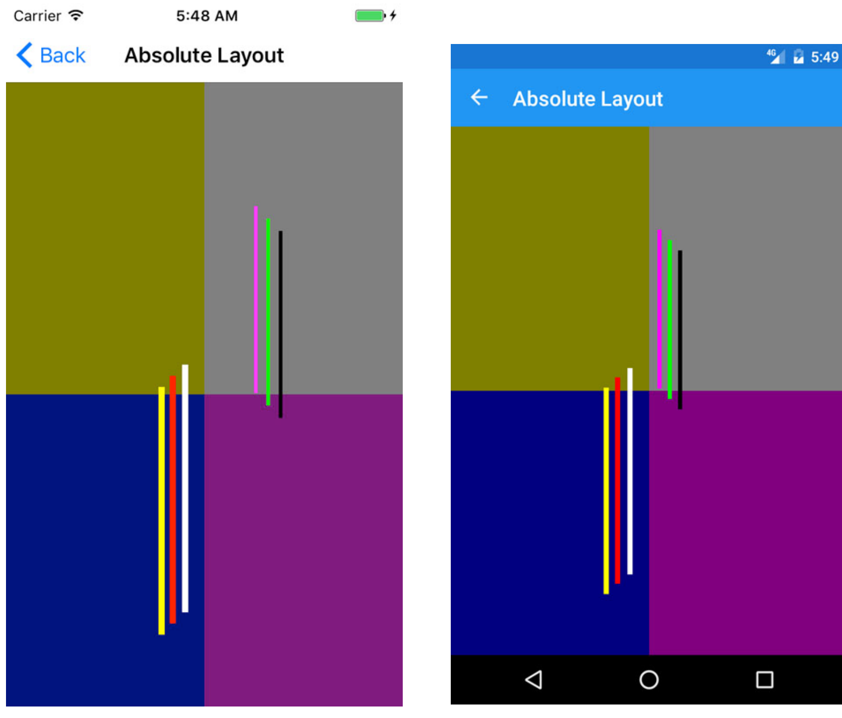
- Some thought should be given as to how the screen will look in both portrait and landscape mode when mixing absolute and proportional positioning/sizing.

Example

The following example demonstrates an *AbsoluteLayout* with controls that have *LayoutFlags* of All, *PositionProportional* and *SizeProportional*.

```
<AbsoluteLayout>
  <BoxView Color="Olive" AbsoluteLayout.LayoutFlags="All"
    AbsoluteLayout.LayoutBounds="0,0,.5,.5" />
  <BoxView Color="Gray" AbsoluteLayout.LayoutFlags="All"
    AbsoluteLayout.LayoutBounds="1,0,.5,.5" />
  <BoxView Color="Navy" AbsoluteLayout.LayoutFlags="All"
    AbsoluteLayout.LayoutBounds="0,1,.5,.5" />
  <BoxView Color="Purple" AbsoluteLayout.LayoutFlags="All"
    AbsoluteLayout.LayoutBounds="1,1,.5,.5" />
  <BoxView Color="Fuchsia" AbsoluteLayout.LayoutFlags="SizeProportional"
    AbsoluteLayout.LayoutBounds="200,100,.01,.3" />
  <BoxView Color="Lime" AbsoluteLayout.LayoutFlags="SizeProportional"
    AbsoluteLayout.LayoutBounds="210,110,.01,.3" />
  <BoxView Color="Black" AbsoluteLayout.LayoutFlags="SizeProportional"
    AbsoluteLayout.LayoutBounds="220,120,.01,.3" />
  <BoxView Color="White" AbsoluteLayout.LayoutFlags="PositionProportional"
    AbsoluteLayout.LayoutBounds=".45,.75,5,200" />
  <BoxView Color="Red" AbsoluteLayout.LayoutFlags="PositionProportional"
    AbsoluteLayout.LayoutBounds=".42,.78,5,200" />
  <BoxView Color="Yellow" AbsoluteLayout.LayoutFlags="PositionProportional"
    AbsoluteLayout.LayoutBounds=".39,.81,5,200" />
</AbsoluteLayout>
```

© 2017 Progress. All Rights Reserved.



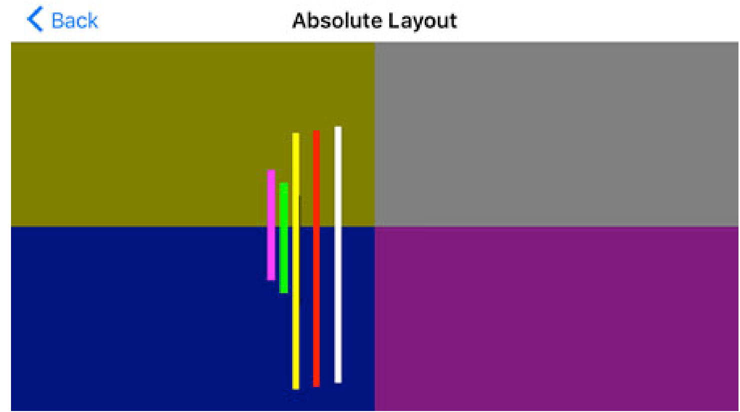
The four large *BoxViews* that form the background of the screen use proportional sizing and positioning, and are set to take up one quarter of the screen each.

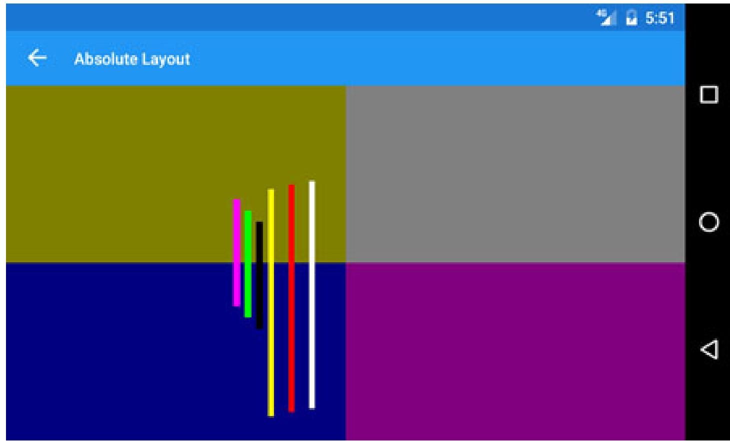
The thin *BoxViews* in the gray quadrant are all set to be *SizeProportional*, meaning their X and Y are absolute and will not change, but their height and width are dependent upon the orientation and size of the device.

The thin *BoxViews* in the blue quadrant are all *PositionProportional*. Their height and width are absolute, but their X and Y are dependent upon the orientation and size of the device.

© 2017 Progress. All Rights Reserved.

The screenshots below demonstrate how the *BoxViews* change when the device is rotated.





RelativeLayout

The final layout that needs to be covered is the *RelativeLayout*. The *RelativeLayout* positions its child controls relatively to one another and to their parent, based on constraints to their position and size properties.

Attached Properties

Attached properties again dictate how the controls are arranged. Each of the following properties constrain their respective position or size value to be dependent upon another View in the layout:

- XConstraint: The X position.
- YConstraint: The Y position.
- WidthConstraint: The width of the control.
- HeightConstraint: The height of the control.

© 2017 Progress. All Rights Reserved.

ConstraintExpression

The values of the above properties are set to a *ConstraintExpression*. The *ConstraintExpression* is used to relate the size or position of one control to another control within the *RelativeLayout*. There are several properties which comprise a *ConstraintExpression*:

- Type: Indicates whether the constraint is relative to the control's parent (*RelativeToParent*) or to another control (*RelativeToView*).
- ElementName: If *Type* is set to *RelativeToView* this is the control's name the constraint is relative to.
- Property: The *property* name the constraint is relative to.
- Factor: A multiplier to apply to the *Property* value.
- Constant: A constant to add to the *Property* value.

An example of a *ConstraintExpression* constraining the X position of one control relative to another control (always 20 units to the left of it) looks like the following:

```
RelativeLayout.XConstraint=”{ConstraintExpression Type=RelativeToView, ElementName=aquaBox,
  Property=X, Factor=1, Constant=-20}”
```

Tips

- Computing the size and positions of all the controls within a *RelativeLayout* can be slow. Thought should be given whether another Layout can be used in lieu of this.

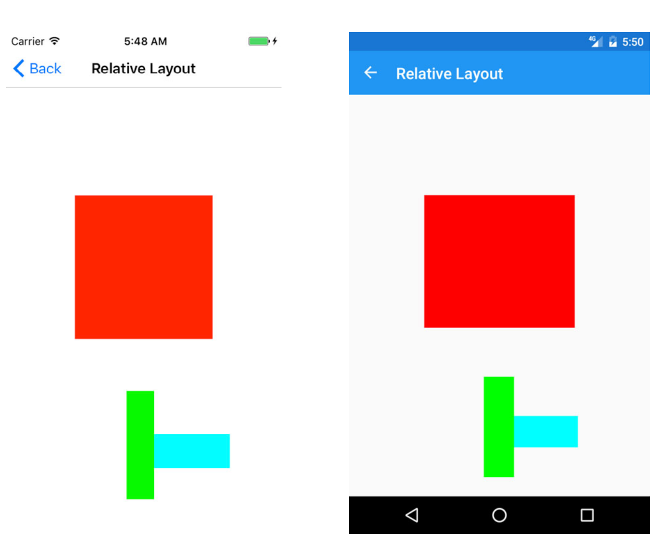
- The height and width of a control can also be specified through the control's *HeightRequest* and *WidthRequest* properties instead of the constraint properties.

Example

This last example shows a *RelativeLayout* in something of an abstract art piece. It contains *BoxViews* that are relative to both other *BoxViews* and to their parent container.

© 2017 Progress. All Rights Reserved.

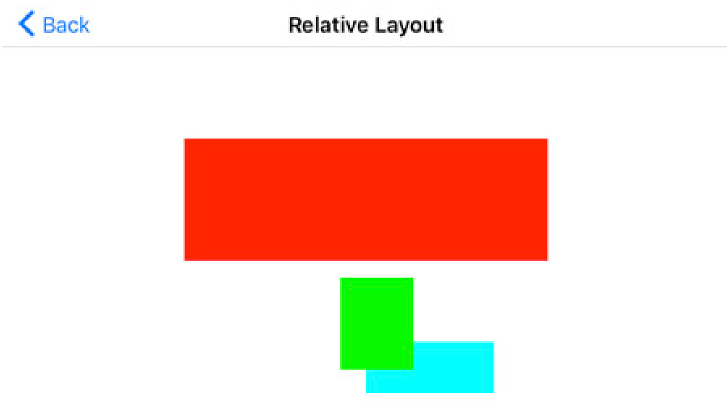
```
<RelativeLayout>
  <BoxView Color="Aqua" x:Name="aquaBox"
    RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent,
                               Property=Width, Factor=0.5}"
    RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent,
                               Property=Height, Factor=.8}"
    WidthRequest="100" />

  <BoxView Color="Lime" x:Name="limeBox"
    RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToView, ElementName=aquaBox,
                               Property=X, Factor=1, Constant=-20}"
    RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToView, ElementName=aquaBox,
                               Property=Y, Factor=1, Constant=-50}"
    RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent,
                                 Property=Width, Factor=.1}"
    RelativeLayout.HeightConstraint="{ConstraintExpression Type=RelativeToParent,
                                 Property=Height, Factor=.25}"/>

  <BoxView Color="Red" x:Name="redBox"
    RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=.25}"
    RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=.25}"
    RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=.5}"
    RelativeLayout.HeightConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=.33}"/>
</RelativeLayout>
```

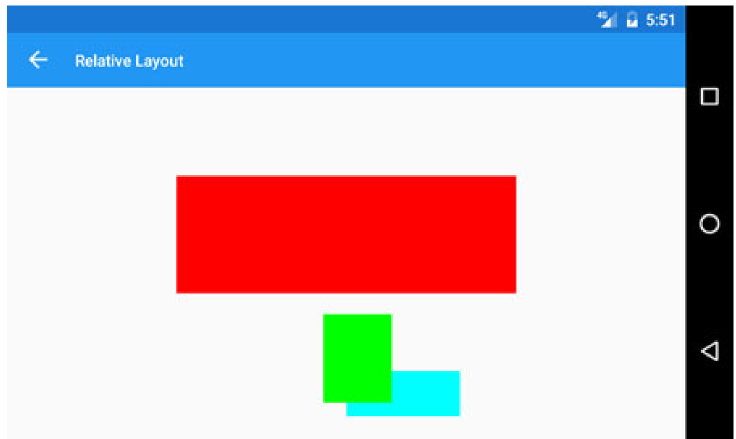


© 2017 Progress. All Rights Reserved.

Here, the aqua *BoxView* is specifically requesting its width to be 100, otherwise it is receiving its X and Y position relative to its parent. The lime *BoxView* is setting its X and Y relative to the aqua *BoxView*. However, it is setting its size relative to its parent. Finally, the red *BoxView* is setting its size and position all relative to its parent.

When the device is rotated, you can see how the layout changes the position and sizes of the elements. Both landscape and portrait views must be considered when designing with a *RelativeLayout*.





Using those four types of layouts, developers can build extremely detailed screens or keep the screens as simple as possible. The power lies in coding the user interface once, then having the layouts arrange the *Views* on-screen to result in an app that has the same fundamental look on each platform.

© 2017 Progress. All Rights Reserved.

Using XAML in Xamarin.Forms

Now that we've gone over the basics of Xamarin.Forms and talked about some common Layouts used to build user interfaces, it's time to introduce XAML—a declarative language that is used for creating user interfaces for Xamarin.Forms.

For many developers, using a markup language such as XAML to build sophisticated user interfaces may seem counterintuitive. Other, more powerful programming languages such as C# may seem better suited for developing complex mobile applications. This is a common reaction.

However, XAML has some unique advantages when it comes to creating powerful user interfaces for Xamarin.Forms. This chapter will explore some of these popular misconceptions and illustrate the strengths of XAML compared to other programming languages like C#.

The Benefits of Using XAML

First and foremost, the biggest benefit to laying out a user interface in XAML is legibility. It becomes very apparent, very quickly, how the visual tree of controls lays out within a page when XAML is used.

With XAML, the code is more readable overall as well. For example, when a control is databound to a property in a view model, the XAML syntax is much cleaner than the corresponding C# syntax.

For example, this is what a XAML databinding looks like:

```
<Label Text="{Binding FirstName}" />
```

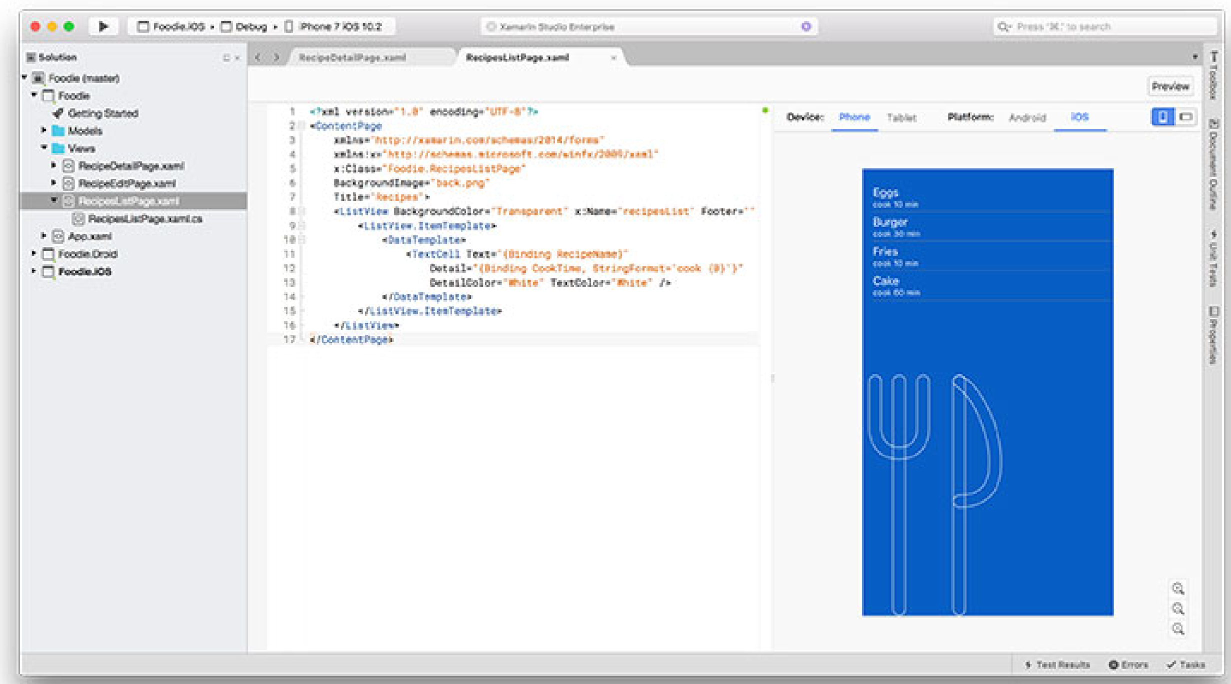
Compared to the same databinding in C#:

```
Label firstNameLabel = new Label();
firstNameLabel.SetBinding(Label.Text, "FirstName");
```

© 2017 Progress. All Rights Reserved.

Combine that syntax over many controls within a page (not to mention placing the controls in their proper layouts and setting their properties), and the XAML version quickly becomes much more readable.

Finally, Xamarin has introduced a XAML previewer in both Xamarin Studio and Visual Studio. This tool enables developers to view their user interface as they build it from within the IDE. The long build, deploy and debug cycle only to view user interface tweaks is gone—and that in itself is a good reason to switch to XAML!

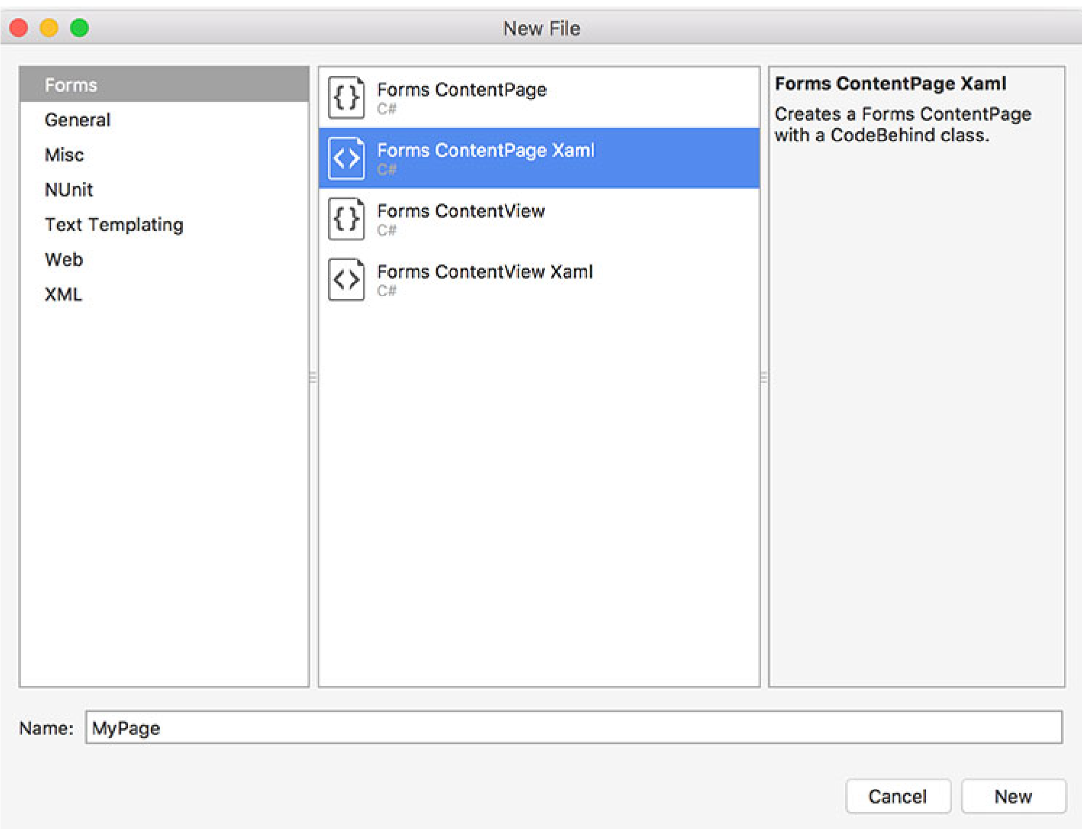


Kitchenware by Xinh Studio via The Noun Project with modifications

The Basics

To get started, developers will need a Xamarin.Forms page that is meant to be built with XAML, and you can find that under the **New File** dialog -> **Forms ContentPage XAML** option.

© 2017 Progress. All Rights Reserved.



Once added, the template will appear as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="Foodie.MyPage">

  <ContentPage.Content>
  </ContentPage.Content>

</ContentPage>
```

Ignoring, for the moment, the first line and the *xmlns* attributes, the opening *<ContentPage>* element lets the XAML parser know that it has come across a *ContentPage* and it should instantiate one. Other UI controls are then placed within the *<ContentPage.Content>* tags, which the XAML parser will then instantiate to build the UI.

The XAML parser is responsible for going through the XAML file, finding elements that need to be instantiated, setting their properties, placing controls in their proper place in the overall layout, perform any databinding, and so on. It is a pretty impressive piece of technology.

© 2017 Progress. All Rights Reserved.

The first thing to consider are the properties on various elements, and how the XAML parser understands them.

Properties

Properties come in two flavors in XAML. The first is known as **Attribute Properties**, which are the easiest and most intuitive to understand. Attribute properties are declared within the tag of the UI component they are a part of. The attribute name will always be the same as the control's property name being referencing, but the value of the attribute will always be a string, regardless of the property's underlying datatype. See below for an example with the *TextColor* property.

```
<Label Text="Burger and Fries" TextColor="Red" />
```

You always specify a string because XAML must adhere to the rules of XML, and the XAML parser is smart of enough to be able to do various type conversions on the fly.

The second type of property encountered in XAML is called the Element Property. These are used when the value of the property is too complex to be expressed within a simple string. Element properties have a tendency to hide in plain sight. In the example above when the blank ContentPage was created, the <ContentPage.Content> element is an Element Property. That property is used to hold the rest of the content of the page, which of course is too complex to be expressed as a string!

Element properties will always be expressed as XML elements. And they will always be of the format {ClassName}.{PropertyName}. You cannot reference a property only by its name alone.

Reacting to Events

Occasionally, it is necessary to react to events that a control raises. And even though a control has been defined in a XAML file, it can still be accessed in the C# code behind file.

You can respond to an event in one of two ways. The first is simply to reference the event's name as an Attribute Property, and then specify a function name. Then in the code-behind file, create a function of the same name—making sure the method signature matches what is expected from the event.

© 2017 Progress. All Rights Reserved.

The XAML:

```
<Button Text="Edit" TextColor="White"
  BackgroundColor="#065ec4" Clicked="recipeClicked"></Button>
```

The code-behind:

```
void recipeClicked(object sender, System.EventArgs e)
{
   // Handle the click event
}
```

The next way to handle events raised from controls defined in XAML is to use another Attribute Property: *x:Name*. This provides access to the entire control in the code-behind file, not just the event. With access to the entire control, the *Clicked* event can be handled as such:

```
recipeButton.Clicked += (sender, args) =>
{
   // do something
}
```

And, of course, developers can access any properties of that control as well.

Accessing Static Data Using XAML

Typical XAML is littered with what appear to be hardcoded string constants all over the place. So then, why is it worth the time to break with that pattern to learn how to access static data rather than just type the value in?

Hardcoding constants in multiple files isn't the best of practices. When the value of the constant changes at design-time and that constant is defined in one spot, rather than referenced all over the place, the whole coding process becomes much easier. XAML also becomes more legible and the scoping of constants can become very fine-grained too.

Additionally, referencing constants from XAML rather than hardcoding them into the control's definition generates some interesting use cases of data binding. Since XAML is a language used in the user interface layer, accessing static data defined as a constant from elsewhere is a great way to start building up a consistent look, or style, for the app.

© 2017 Progress. All Rights Reserved.

StaticExtension Class

One of the easiest ways of accessing static data from a XAML file is with the *StaticExtension* class. The static extension class is used to access a static member, be it a constant, static property or field, or an enumeration of an object. It then returns the value of that static member.

For example, the *RecipeNameLabel* field of the following class:

```
public static class RecipeUIConstants
{
   public static string RecipeNameLabel = "Recipe Name";
}
```

Would be accessed in XAML as the following:

```
<Label Text="{x:Static local:RecipeUIConstants.RecipeNameLabel}" />
```

(Note that the local: XML namespace would be set to reference the CLR namespace of the RecipeUIConstants class at the very top of the XAML file.)

The usage then is to set the XAML property to the *StaticExtenstion* class. The *StaticExtensionclass* takes the format of *{x:Static* followed by the path to get at the static member. In this case the XML namespace of local followed by the class name and field name.

The *StaticExtension* class is not limited to only returning strings however, and it can be used to build up a more interesting UI as in the following example:

```
public static class RecipeUIConstants
{
   public static string RecipeNameLabel = "Recipe Name";
   public static string CookTimeLabel = "Cook Time";
   public static string IngredientsLabel = "Ingredients";
   public static string DirectionsLabel = "Directions";
   public static string NumberOfServingsLabel = "Number of Servings";

   public static Thickness PickerMargin = new Thickness(15, 0);

   public static AllServingOptions NumberOfServingsOptions = new AllServingOptions
   {
```

© 2017 Progress. All Rights Reserved.

```
    new ServingOption { Description = "Individual", Servings = 1 },
    new ServingOption { Description = "Family Sized", Servings = 4 },
    new ServingOption { Description = "Buffet", Servings = 12 }
  };
```

```
}

public class AllServingOptions : List<ServingOption>
{
  public AllServingOptions(params ServingOption[] args)
  {
    this.AddRange(args);
  }

  public AllServingOptions() { }
}

public class ServingOption
{
  public string Description { get; set; }
  public int Servings { get; set; }
}
```

The *RecipeUIConstants* class has been expanded to include new fields—including one with a type of *Thickness* and another with a type of *AllServingOptions* : List<*ServingOption*>.

These can all be accessed in the following way to build up a data entry page:

```
<?xml version="1.0" encoding="utf-8"?>
<ContentPage
  xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:local="clr-namespace:Xaml2"
  x:Class="Xaml2.Xaml2Page"
  Title="XAML Demo">
  <TableView Intent="Form">
    <TableView.Root>
      <TableSection Title="Enter Data">
        <EntryCell Label="{x:Static local:RecipeUIConstants.RecipeNameLabel}" />
        <EntryCell Label="{x:Static local:RecipeUIConstants.CookTimeLabel}" />
        <EntryCell Label="{x:Static local:RecipeUIConstants.IngredientsLabel}" />
        <EntryCell Label="{x:Static local:RecipeUIConstants.DirectionsLabel}" />
        <ViewCell>
          <StackLayout Orientation="Horizontal" Margin="{x:Static local:RecipeUIConstants.
PickerMargin}">
```

© 2017 Progress. All Rights Reserved.

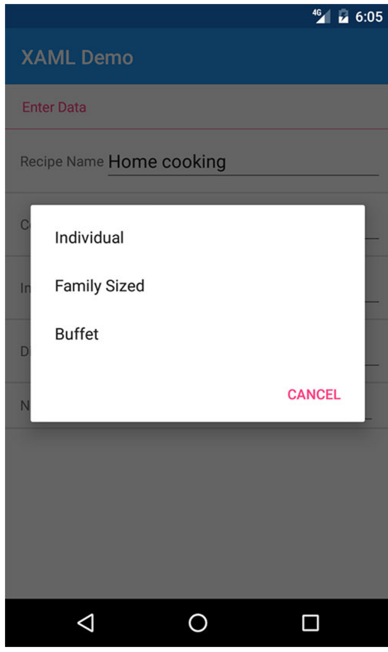
```
                <Label Text="{x:Static local:RecipeUIConstants.NumberOfServingsLabel}"
VerticalOptions="Center" />
                <Picker VerticalOptions="Center" HorizontalOptions="EndAndExpand"
                  ItemsSource="{x:Static local:RecipeUIConstants.NumberOfServingsOptions}"
                  ItemDisplayBinding="{Binding Description}" />
              </StackLayout>
            </ViewCell>
          </TableSection>
        </TableView.Root>
    </TableView>
</ContentPage>
```

The Labels of the *EntryCells* are all set to the various strings, but something interesting is going on in the *ViewCell* that contains the *Picker* control.

First off, notice the *StackLayout*'s *Margin* property is set to a value corresponding to a Thickness, thus demonstrating that the *StaticExtension* class can do more than return string values.

Secondly, the *Picker* is bound to an object that inherits from *List<ServingOption>* property. The ItemsSource can be set to a statically declared collection.

The running app with the picker shown looks like this:



That's great, but this ebook is about XAML, and those constants were declared in C#. Fortunately, developers can do the same thing even when declaring the constants in XAML.

© 2017 Progress. All Rights Reserved.

Resource Dictionaries

Besides being stored in C# files, static values can be stored in XAML, inside what is known as a *ResourceDictionary*.

A *ResourceDictionary* is a *Dictionary<string, object>* class and is accessed through the Resources property of any Xamarin.Forms object that inherits from *VisualElement*.

That means the *Resources* property is available on most controls in Xamarin.Forms. And what's more—a child control can access anything in its parent's *Resources ResourceDictionary*.

For example, every control on a page will have access to objects in the ResourceDictionary declared in the *ContentPage.Resources* property. Defining a *ResourceDictionary* will look like the following:

```
<ContentPage.Resources>
  <ResourceDictionary>
    <x:String x:Key="RecipeNameLabel">Recipe Name</x:String>
    <x:String x:Key="CookTimeLabel">Cook Time</x:String>
    <x:String x:Key="IngredientsLabel">Ingredients</x:String>
    <x:String x:Key="DirectionsLabel">Directions</x:String>
    <x:String x:Key="NumberOfServingsLabel">Number of Servings</x:String>
    <Thickness x:Key="PickerMargin">15,0</Thickness>
  </ResourceDictionary>
</ContentPage.Resources>
```

Each static value is declared via its data type followed by a key, which it will serve as its key in the Dictionary. Finally, the value of the constant is given.

To access values defined in a ResourceDictionary you need to use the StaticResource markup extension.

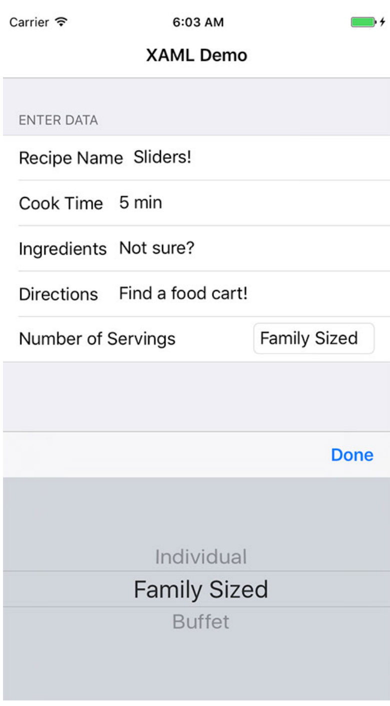
```
<EntryCell Label="{StaticResource RecipeNameLabel}"/>
<EntryCell Label="{StaticResource CookTimeLabel}" />
<EntryCell Label="{StaticResource IngredientsLabel}" />
<EntryCell Label="{StaticResource DirectionsLabel}" />
<ViewCell>
  <StackLayout Orientation="Horizontal" Margin="{StaticResource PickerMargin}">
    <Label Text="{StaticResource NumberOfServingsLabel}" VerticalOptions="Center" />
    <Picker VerticalOptions="Center" HorizontalOptions="EndAndExpand"
      ItemsSource="{x:Static local:RecipeUIConstants.NumberOfServingsOptions}"
      ItemDisplayBinding="{Binding Description}" />
  </StackLayout>
</ViewCell>
```

© 2017 Progress. All Rights Reserved.

The *StaticResource* is invoked within curly braces followed by the name of the key in the *ResourceDictionary* you wish to retrieve the value from. You then set the *{StaticResource KeyName}* to the property you wish to receive its value. The XAML is also a bit more legible, a side-benefit of using the *ResourceDictionary*.

Because values from *ResourceDictionary's* have a scope, it is easy to define global level constants in the App class and more finely grained constants only where needed at the Page level or even further down, such as at the individual *Layout* level.

Thus *ResourceDictionary* then not only provides developers with the ability to define static values in XAML, leading to more legible code, but also gives more control of where the values apply to.



Everything still looks the same in the app. However, you may have noticed that there was one *StaticExtension* class holdover in the *Picker.ItemSource* property:

```
<Picker VerticalOptions="Center" HorizontalOptions="EndAndExpand"
  ItemsSource="{x:Static local:RecipeUIConstants.NumberOfServingsOptions}"
  ItemDisplayBinding="{Binding Description}" />
```

That's because creating complex objects is a bit more difficult—but it can be done, and the next section will illustrate how.

© 2017 Progress. All Rights Reserved.

Creating Objects From XAML

It is entirely possible to create complex objects within XAML. Both parameterless and parametered constructors can be invoked.

Parameterless constructors are easy, all developers needs to do is reference the class they wish to instantiate, and the object will be created. For example:

```
<local:ServingOption />
```

Will create a *ServingOption* class—although its properties will not be set to any values. However, we can also invoke constructors with parameters in XAML, and that's with using the *<x:Arguments>* keyword.

If we modify the *ServingOption* class to look like the code below:

```
public class ServingOption
{
   public ServingOption(string description, int servings)
   {
      Description = description;
      Servings = servings;
   }

   public string Description { get; set; }
   public int Servings { get; set; }
}
```

Then the following syntax would be used to create that class in XAML:

```
<local:ServingOption x:Key="Buffet">
   <x:Arguments>
      <x:String>Buffet</x:String>
      <x:Int32>12</x:Int32>
   </x:Arguments>
</local:ServingOption>
```

© 2017 Progress. All Rights Reserved.

The values for the constructor get passed in within the **<x:Arguments>** tags. Now the entire page can be created within XAML—no object instantiation needed from anywhere else in the code:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
        xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
        xmlns:local="clr-namespace:Xaml2"
        xmlns:generic="clr-namespace:System.Collections;assembly=mscorlib"
        x:Class="Xaml2.ConstructorPage">

  <ContentPage.Resources>
    <ResourceDictionary>
      <x:String x:Key="RecipeNameLabel">Recipe Name</x:String>
      <x:String x:Key="CookTimeLabel">Cook Time</x:String>
      <x:String x:Key="IngredientsLabel">Ingredients</x:String>
      <x:String x:Key="DirectionsLabel">Directions</x:String>
      <x:String x:Key="NumberOfServingsLabel">Number of Servings</x:String>
      <Thickness x:Key="PickerMargin">15,0</Thickness>

      <local:AllServingOptions x:Key="servingOptions">
        <x:Arguments>
          <x:Array x:Key="args" Type="{x:Type local:ServingOption}">

            <local:ServingOption x:Key="Individual">
              <x:Arguments>
                <x:String>Individual</x:String>
                <x:Int32>1</x:Int32>
              </x:Arguments>
            </local:ServingOption>

            <local:ServingOption x:Key="FamilySized">
              <x:Arguments>
                <x:String>Family Sized</x:String>
                <x:Int32>4</x:Int32>
              </x:Arguments>
            </local:ServingOption>

            <local:ServingOption x:Key="Buffet">
              <x:Arguments>
                <x:String>Buffet</x:String>
                <x:Int32>12</x:Int32>
              </x:Arguments>
            </local:ServingOption>

          </x:Array>
        </x:Arguments>
      </local:AllServingOptions>

    </ResourceDictionary>
  </ContentPage.Resources>
```

© 2017 Progress. All Rights Reserved.

```
  <TableView Intent="Form">
    <TableView.Root>
      <TableSection Title="Enter Data">
        <EntryCell Label="{StaticResource RecipeNameLabel}"/>
        <EntryCell Label="{StaticResource CookTimeLabel}" />
        <EntryCell Label="{StaticResource IngredientsLabel}" />
        <EntryCell Label="{StaticResource DirectionsLabel}" />
        <ViewCell>
          <StackLayout Orientation="Horizontal" Margin="{StaticResource PickerMargin}">
            <Label Text="{StaticResource NumberOfServingsLabel}" VerticalOptions="Center" />
            <Picker VerticalOptions="Center" HorizontalOptions="EndAndExpand"
              ItemsSource="{StaticResource servingOptions}"
              ItemDisplayBinding="{Binding Description}" />
          </StackLayout>
        </ViewCell>
      </TableSection>
    </TableView.Root>
  </TableView>
</ContentPage>
```

The interesting thing to look at here is what is happening within the servingOptions key. It is first creating an AllServingOptions object, using the constructor that accepts an array of parameters. Then it is creating individual ServingOption objects to be sent into that array.

All the variable initialization does is make the page long, and one would most likely put the servingOptions key into the App class, but the point is that you can pass static data to properties in XAML.

To conclude, accessing static data via XAML can be done in three different ways:

1. Simply hardcode the value in. However, the downsides to this are apparent as soon as keeping the code consistent or refactoring takes place and should be avoided.

2. Use the StaticExtension class. This class accesses statically defined fields, constants, and enumerations from with XAML.

3. Use the ResourceDictionary. You can place and access the same type of data from a ResourceDictionary that you can from a StaticExtension, except all of the values are initialized from XAML.

© 2017 Progress. All Rights Reserved.

Accessing Dynamic Data

In the previous chapter, we discussed how to use XAML to access static date. However, that's only one part of the equation—most applications also need to be able to access dynamic data. Data is always changing, so the application must also be able to support data that changes at runtime as well.

In this chapter, we will discuss how controls defined in XAML aren't required to have static properties that are set at design time and never change. XAML controls can update their values dynamically when the data attached to them changes.

Dynamic Resources

As previously discussed, one way to store and subsequently retrieve static data is via the *ResourceDictionary* property on a class that inherits from *VisualElement*—and generally whatever is read from the *ResourceDictionary* does not change.

The code to access a static value in a *ResourceDictionary* looks like the following:

```
<ContentPage.Resources>
  <ResourceDictionary>
    <x:String x:Key="RecipeNameLabel">Recipe Name</x:String>
  </ResourceDictionary>
</ContentPage.Resources>

<!-- Some other code here -->

<EntryCell Label="{StaticResource RecipeNameLabel}" />
```

However, should the need ever arise to change that *EntryCell*'s label, *DynamicResources* can be used. A *DynamicResource* allows the value of the *ResourceDictionary* object to be changed at runtime, and that updated value will be reflected in whatever property that references the object.

Setting up a *ResourceDictionary* to use a *DynamicResource* is done the same way as *StaticResources*—no changes are needed. The change comes in on how the object is referenced and, as you may have guessed by now, it's referenced by the *DynamicResource* keyword.

© 2017 Progress. All Rights Reserved.

In the example from above, the *EntryCell* control would access the *RecipeNameLabel* object in the *ResourceDictionary* as follows:

```
<EntryCell Label="{DynamicResource RecipeNameLabel}" />
```

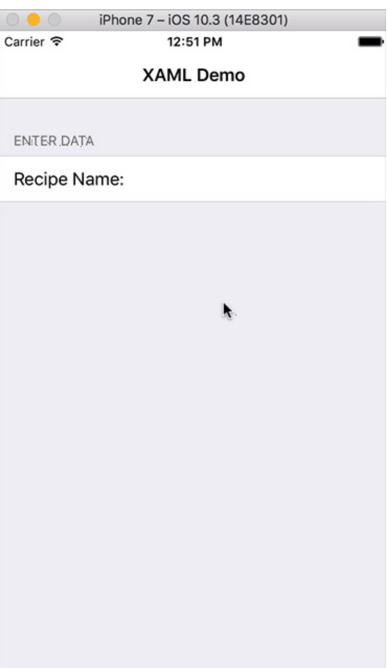
The difference between *DynamicResource* and *StaticResource* is that the XAML compiler will only access anything defined as a *StaticResource* once, pop in the value and then leave it be. Whereas with *DynamicResource* a link is maintained between that dictionary item and the property it's set to.

The natural question, then, is how does the *RecipeNameLabel* entry in the *ResourceDictionary* get changed? That will have to happen in the XAML's code-behind file.

In response to some external event, the *Resources* dictionary is accessed, and the dictionary item is updated—exactly in the same way that any dictionary item is updated.

In the example below, a button that indicates the recipe is "the world's greatest" is clicked.

```
worldsGreatest.Clicked += (sender, e) => {
   Resources["RecipeNameLabel"] = "The world's greatest recipe name:";
};
```



The *ResourceDictionary* is located in the *Resources* property.

DynamicResources are not the only way to update data. Xamarin.Forms comes with a powerful data binding engine that allows properties (and events) of controls to be bound to (special) properties and actions of various classes.

© 2017 Progress. All Rights Reserved.

Data Binding

Updating a control's property value through a *DynamicResource* works, but it's not robust enough to use on a large scale. One of the key benefits of XAML is that all of the code for the user interface can be expressed in the XAML markup and a lot of logic can stay out of the code-behind file. An excellent way to keep true to that benefit is to use data binding.

Data binding allows a control to be bound to a backing model (or something that purely represents data) class and have both various properties of the control and the model be updated whenever either changes—without having to handle any events.

Imagine there is a class that models recipe data. Naturally it would have properties for the recipe's name, ingredients, directions and so on. That class would be used to both capture data from the user interface and then save the recipe to a data store of some type.

In the case of the recipe data entry, data binding saves the developer from writing a bunch of boilerplate code that sets the recipe's properties to the various properties of the controls that display them—and then more code to move the new values from the controls back into the recipe object. This becomes especially tedious when that same model class is used throughout the app. Thus, databinding frees the developer to focus on implementing core app logic instead of boilerplate code, like getting the recipe to save to the data store.

Setting up the XAML code to make use of data binding is simple, all developers need to do is set whatever property they want to bind equal to the *Binding* keyword and then the model's property that should be bound to.

A simple form that displays and updates recipes would look like the following:

```
<TableView Intent="Form">
  <TableView.Root>
    <TableSection Title="Enter Data">
      <EntryCell Label="Recipe Name" Text="{Binding RecipeName}" />
      <EntryCell Label="Ingredients" Text="{Binding Ingredients}" />
      <EntryCell Label="Directions" Text="{Binding Directions}" />
    </TableSection>
  </TableView.Root>
</TableView>
```

© 2017 Progress. All Rights Reserved.

However, there is some extra work that needs to be done in the model class that enables the two-way communication between it and the view. Specifically, it must implement the *INotifyPropertyChanged* interface.

It's this interface that allows the controls on the UI to know that something has changed, and vice versa.

Only one event must be implemented as part of *INotifyPropertyChanged*:

```
public event PropertyChangedEventHandler PropertyChanged;
```

By invoking that event every time a property changes value, the user interface will automatically get updated. And by extension, the model will get updated every time the user interface changes.

The recipe class, when fully implemented, will look like the following:

```
public class Recipe : INotifyPropertyChanged
{
  string _recipeName;
  public string RecipeName
  {
    get => _recipeName;
    set
    {
      if (string.Equals(_recipeName, value))
        return;

      _recipeName = value;

      PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(nameof(RecipeName)));
    }
  }

  string _ingredients;
  public string Ingredients
  {
    get => _ingredients;
    set
    {
      if (string.Equals(_ingredients, value))
        return;
      _ingredients = value;

      PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(nameof(Ingredients)));
    }
  }
```

© 2017 Progress. All Rights Reserved.

```
   string _directions;
   public string Directions
   {
      get => _directions;
      set
      {
         if (string.Equals(_directions, value))
            return;

         _directions = value;

         PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(nameof(Directions)));
      }
   }

   public event PropertyChangedEventHandler PropertyChanged;
}
```

A couple things to note. First is that if the passed in value of the property is the same as what is already there, nothing happens. You don't want to have the binding fire to set something that is already set.

The second is that developers must ensure something is subscribed to the *PropertyChanged* event before invoking it to avoid any null exceptions.

Then there is a final piece of the puzzle to get data binding to work: associating the model to the *Page* that hosts the controls. That's done through the *BindingContext* property.

You can set the *BindingContext* either in the code-behind file, or in XAML by using the constructor syntax you learned about in the last article to create the model. Here's an example of setting the *BindingContext* on the *ContentPage* using the XAML constructor syntax.

```
<ContentPage.BindingContext>
   <local:Recipe />
</ContentPage.BindingContext>
```

© 2017 Progress. All Rights Reserved.

Here's an example of a button on a Xamarin.Forms page simulating a database update. Pretend the button click event is actually a database service call coming back and updating the model – you can see how the control's in the UI get updated instantly.

```
dbSimulation.Clicked += (s, e) =>
{
  // The recipe object would be returned from the data store - this is only for demo
  var bc = BindingContext as Recipe;

  bc.RecipeName = "new recipe";
  bc.Ingredients = "new ingredients";
  bc.Directions = "new directions";
};
```

It should be noted that anything in Xamarin. Forms that inherits from *BindableObject* has a *BindingContext* property. Controls are one thing that inherit from *BindableObject*, so it is possible to have different controls bound to different data sources all on the same page. The *BindingContext* of a parent control will be applied to all of its child controls, so setting it at the page level (as done above at the *ContentPage*) will ensure all the controls beneath it have the same *BindingContext*.

One concern developers may have is messing up their models with *INotifyPropertyChanged*. Fortunately, there is a design pattern around that, and it's called MVVM, or Model-View-*ViewModel*, where the *ViewModel* implements *INotifyPropertyChanged*. That will be covered in a future publication.

© 2017 Progress. All Rights Reserved.

Telerik UI for Xamarin

Now that you are well on your way to being a rockstar Xamarin.Forms developer, you may soon realize a little gap in your arsenal – professional Xamarin apps invariably benefit from polished performant UI controls. You would rather not sweat on pixel-perfect UI and want to ship your apps faster. Take a good look at Telerik UI for Xamarin – there are showcase apps in app Stores for you to play around with and they are super easy to integrate in your apps. You'll be covered for some of popular yet complex UI scenarios – like Graphs/Charts, ListView, SideDrawer, Calendar, DataForm and more. You get truly native, truly cross-platform rich UI controls - all wrapped up for your Xamarin.Forms apps. Download a free trial today and get started.

Download a free trial

About the Autor

Matthew Soucoup

Microsoft and Xamarin MVP

Matthew Soucoup is a Microsoft and Xamarin MVP, a Pluralsight author, and a principal at Code Mill Technologies. Matt loves mobile development. For his job, he creates elegant cross-platform apps. Then for fun, Matt shares his passion and insight on mobile and cloud development by blogging, writing articles, and presenting at conferences such as Xamarin Evolve, CodeMash, That Conference, Indy. Code(), and MKEdotNET.

© 2017 Progress. All Rights Reserved.

About Progress

Progress (NASDAQ: PRGS) offers the leading platform for developing and deploying mission-critical business applications. Progress empowers enterprises and ISVs to build and deliver cognitive-first applications that harness big data to derive business insights and competitive advantage. Progress offers leading technologies for easily building powerful user interfaces across any type of device, a reliable, scalable and secure backend platform to deploy modern applications, leading data connectivity to all sources, and award-winning predictive analytics that brings the power of machine learning to any organization. Over 1,700 independent software vendors, 80,000 enterprise customers, and 2 million developers rely on Progress to power their applications. Learn about Progress at www.progress.com or +1-800-477-6473.

Worldwide Headquarters

Progress, 14 Oak Park, Bedford, MA 01730 USA
Tel: +1 781 280-4000 Fax: +1 781 280-4095
On the Web at: www.progress.com
Find us on facebook.com/progresssw twitter.com/progresssw youtube.com/progresssw
For regional international office locations and contact information, please go to www.progress.com/worldwide

© 2017 Progress. All Rights Reserved.

Progress and Telerik by Progress are trademarks or registered trademarks of Progress Software Corporation and/or one of its subsidiaries or affiliates in the U.S. and/or other countries. Any other trademarks contained herein are the property of their respective owners.

© 2017 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

Rev 2017/08 | 170808-0023