XAMARIN COMMUNITY TOOLKIT

SUCCINCTLY

BY ALESSANDRO DEL SOLE

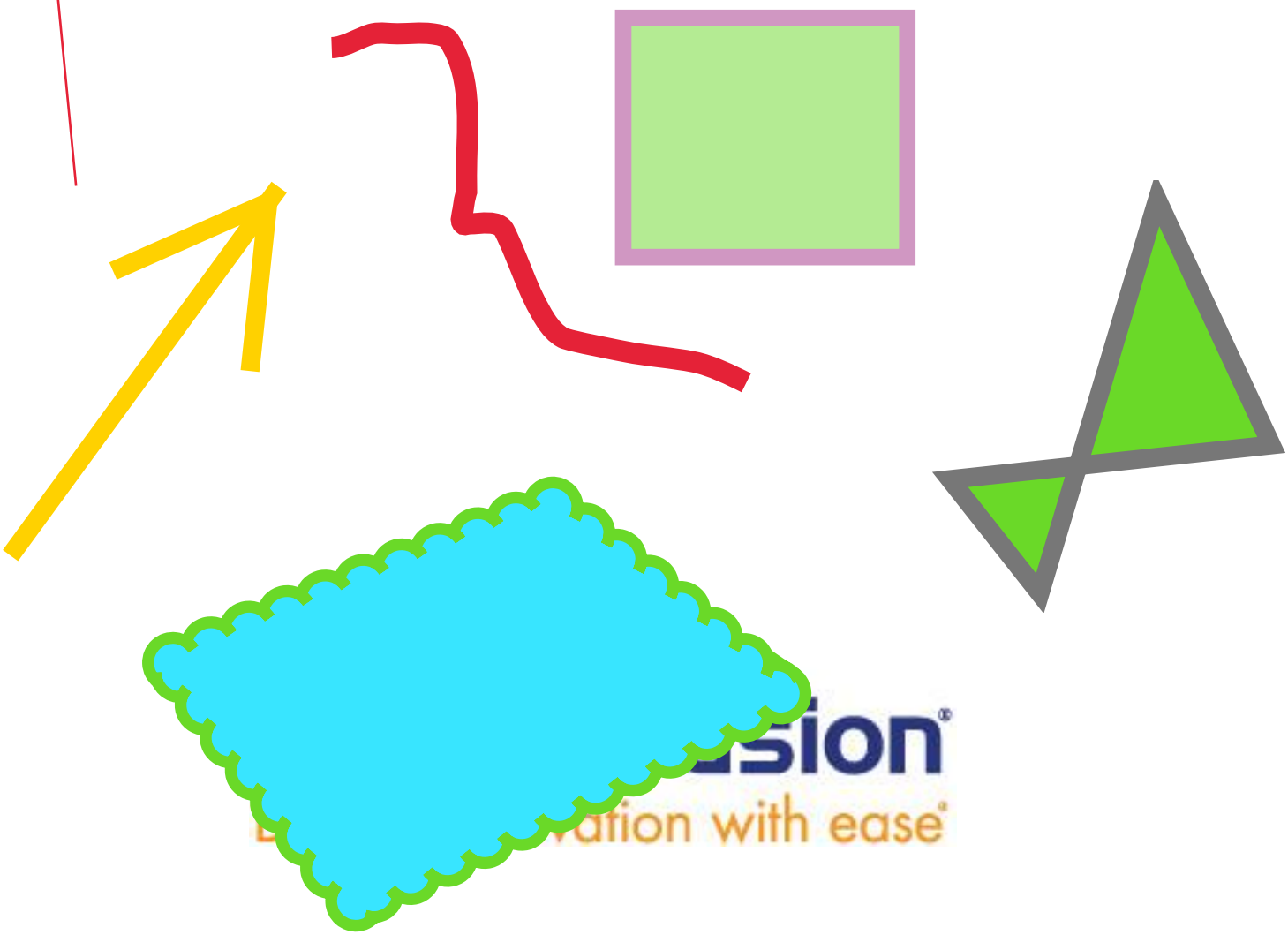
SUCCINCTLY EBOOK SERIES

Syncfusion

Xamarin Community Toolkit Succinctly

By
Alessandro Del Sole

Foreword by Daniel Jebaraj



Approve

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

ISBN: 978-1-64200-220-1

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any damages whatsoever whether direct, indirect, incidental, consequential, arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

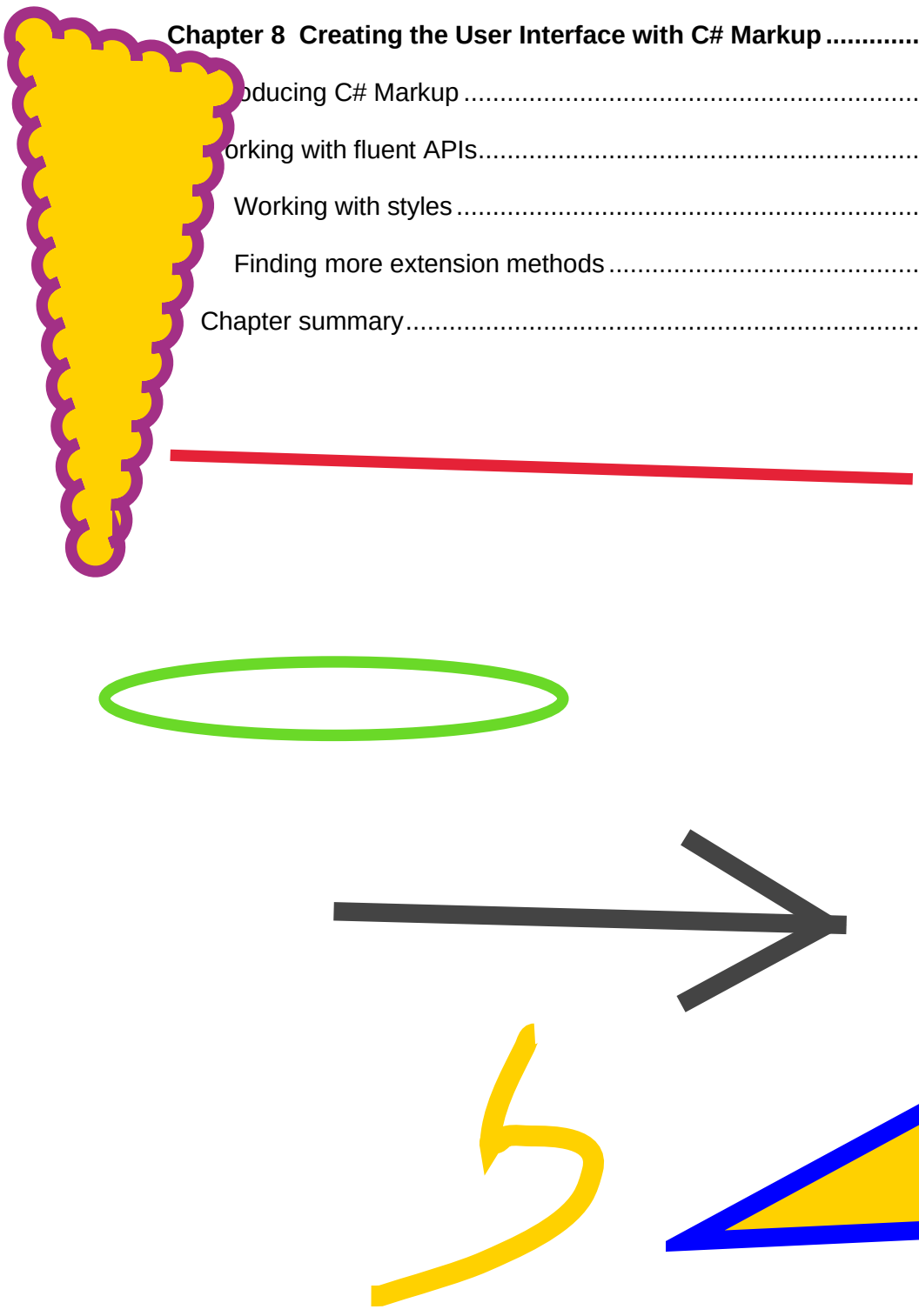
Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

IKEA

Table of Contents



The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, CEO
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Alessandro Del Sole is a Xamarin Certified Mobile Developer and has been a Microsoft MVP since 2008. Awarded MVP of the Year in 2009, 2010, 2011, 2012, and 2014, he is internationally considered a Visual Studio expert and a .NET authority. Alessandro has authored many printed books and e-books on programming with Visual Studio, including *Xamarin.Forms Succinctly*, *Visual Basic 2015 Unleashed*, and *Visual Studio 2019 Succinctly*. He has written tons of technical articles about .NET, Visual Studio, and other Microsoft technologies in Italian and English for many developer portals, including MSDN Magazine and the Visual Studio Developer Center from Microsoft. He has also produced a number of instructional videos in both English and Italian. Alessandro works as a senior software engineer for Fresenius Medical Care, focusing on building mobile apps with Xamarin in the healthcare market. You can follow him on Twitter at @progalex.

Chapter 1 Introducing the Xamarin Community Toolkit

When working on several Xamarin.Forms projects, developers often tend to replicate some elements used in one project into another one. This is very common with value converters, custom views, and behaviors. In addition, sometimes developers need to build views that are common in modern mobile app design, and that the Xamarin.Forms code base does not include. In order to simplify reusing elements across projects, Microsoft offers a new library called Xamarin Community Toolkit. This chapter introduces the library and explains some conventions that I will be using across the book.

What to expect from this book

After reading this book, you will be able to use all the new elements offered by the Xamarin Community Toolkit, and this will improve your productivity and skills on Xamarin.Forms. However, this book assumes you already have good knowledge of Xamarin.Forms as a development platform and of Microsoft Visual Studio 2019 as the development tool you use to build apps with Xamarin.Forms. The main reason for this is to keep this book succinct and provide you with productivity tools in a faster way, so it's not possible to explain how Xamarin.Forms works.

If you are a beginner, I recommend you read my free book *Xamarin.Forms Succinctly* first, since it will give you all the necessary knowledge you need. In addition, and for exactly the same reasons, it will not be possible to describe the architecture and the implementation of such elements, except where strictly necessary. If you need to understand more about architecture and implementation, you can always look at the official Microsoft documentation and investigate the source code yourself. You will discover that the backing code is not complex at all, and that it is very well commented.

Note: At this writing, the latest stable version of the Xamarin Community Toolkit is 1.2.0, which also adds support for F#. Keep in mind that this library continuously evolves, which means that new elements might be added over time (after the release of this book). This is another reason to bookmark the documentation for further updates.

Xamarin Community Toolkit: Solving common problems

The Xamarin Community Toolkit is an open-source collection of reusable elements for mobile development with Xamarin.Forms and works with all the supported operating systems (OS). It includes views, value converters, behaviors, animations, color themes, and helper classes that you will be able to reuse across projects, and that will solve common problems without the need to share your code manually or reinvent the wheel.

Some of the views included in the Xamarin Community Toolkit bridge the gap between modern mobile app designs and the Xamarin.Forms code base. Technically speaking, it is a .NET Standard library that you can quickly add to your projects via NuGet, and for which a GitHub repository is available. This repository is very important, not only because of the availability of the source code of the library, but also for the availability of sample code and for the product roadmap.

Using the Xamarin Community Toolkit

The Xamarin Community Toolkit is available to the general public as a NuGet package. At this writing, the latest version available is 1.2.0. When you work with Xamarin.Forms, you can install the library to all the projects in the solution. Figure 1 shows how the library appears in the NuGet Package Manager user interface of Visual Studio 2019.

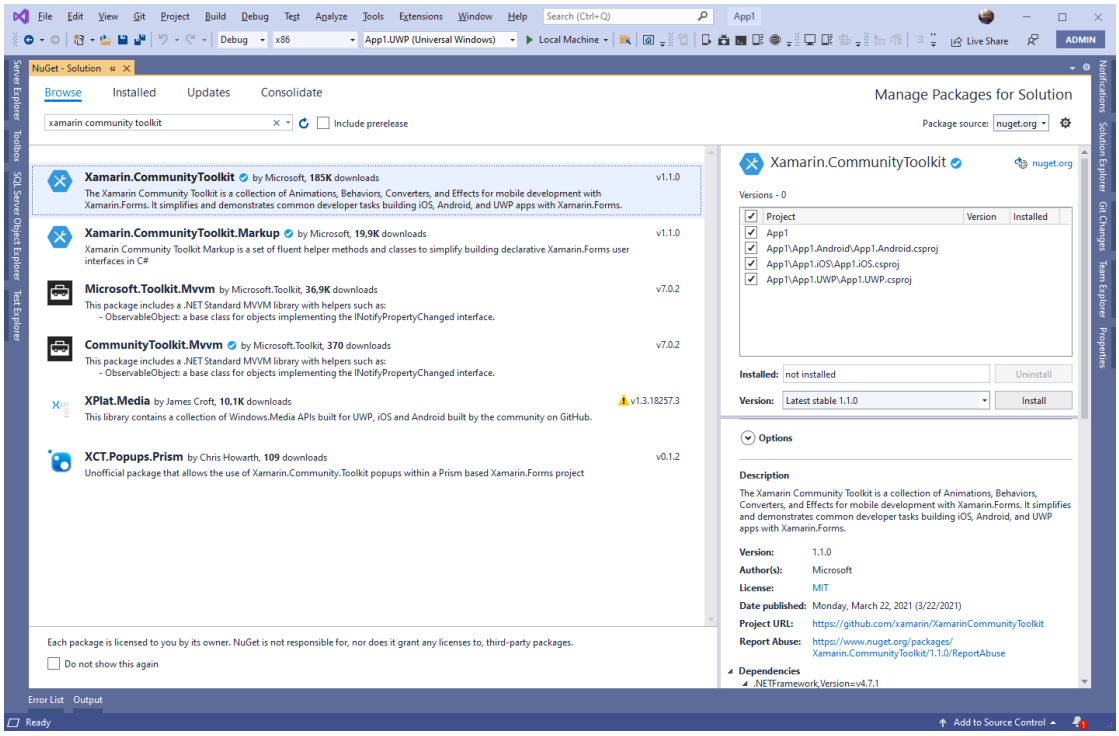


Figure 1: Installing the Xamarin Community Toolkit

There is also another NuGet package called Xamarin.Community.Toolkit.Markup, which is actually necessary only if you want to work with C# Markup Extensions. These are discussed in Chapter 8. Once you have installed the library, you will be able to use all the objects discussed in this book. However, it is not necessary to create a new project now. In the next paragraph, I will explain how to use the official sample app.

Tip: Although some of the elements provided by the Xamarin Community Toolkit will work with previous versions of Xamarin.Forms, some of them require Xamarin.Forms 5.0. My recommendation is that you ensure your projects are using Xamarin.Forms version 5.0 or later to use all the features discussed in the book.

Setting up the development environment

It is possible to combine efficiency, productivity, and learning purposes by using the official sample app created by Microsoft to demonstrate how the Xamarin Community Toolkit works. It's very well organized, so all the discussions and examples in the book will be based on the official sample. In addition, this is always updated as new releases are available, so you can more easily stay up to date. Having that said, first open the GitHub repository for the project. Click the **Code** button, as shown in Figure 2.

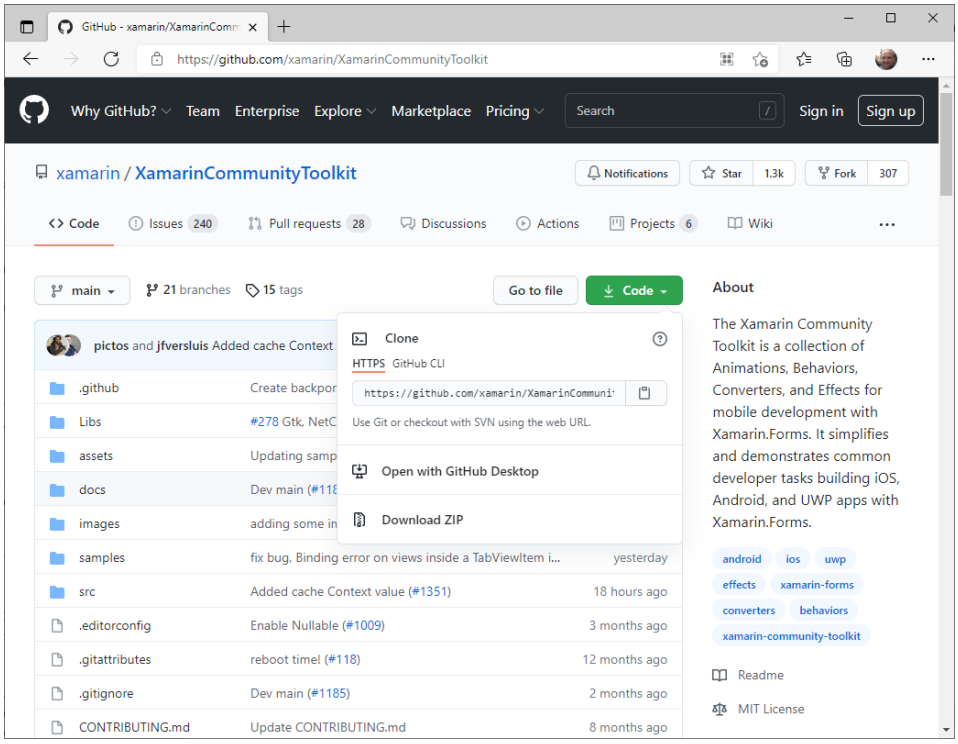


Figure 2: The official repository for the Xamarin Community Toolkit

At this point, you can decide how to get the source code between cloning the repository in Visual Studio or downloading the full ZIP archive. The repository contains the full source code of the library, plus the source code for a complete sample application.

If you want to explore the source code for the Xamarin Community Toolkit, you can open the Xamarin.CommunityToolkit.sln solution file in Visual Studio. For the purposes of this book, the solution you need to open is called **XCT.Sample.sln**, which is in the **samples** folder of the repository. Figure 3 shows how the solution appears in Solution Explorer.

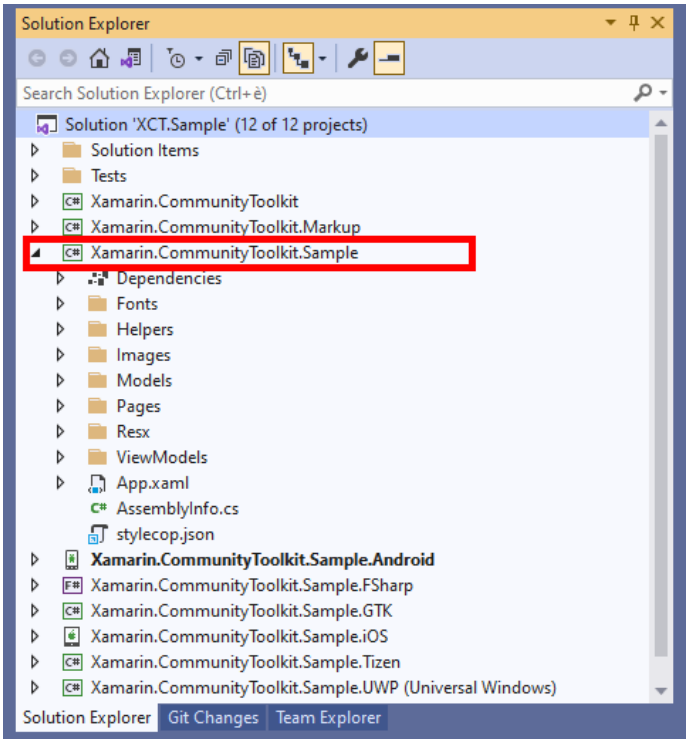


Figure 3: The official sample project in Visual Studio 2019

The project of interest throughout this book is the shared project called **Xamarin.CommunityToolkit.Sample,** highlighted in red in Figure 3. This contains the sample pages, ViewModels, assets, and resources required to build the sample app, which provides access to examples of all the objects exposed by the toolkit.

> ***Tip: The reason why I told you to download the source code for the full repository instead of only the source code for the sample app is that the Xamarin Community Toolkit in the sample project is not downloaded from NuGet; it is referenced via the Visual Studio projects in the repository. If you only downloaded the sample app source code, you would have had to install the NuGet package manually and make some changes that would result in a waste of time.***

Choose a platform project as the startup project, select an appropriate device (either physical or emulated), and then press F5 to start the sample application. I will use the Android platform project and an Android emulator, but everything will work similarly on a different device.

When the app is running, you will see a welcome page that you can scroll through to see a list of cards. Each card represents a shortcut to examples about a specific collection of features. When you click a card, you access a sublist of cards, and each card provides examples about a specific feature. If you look at Figure 4, starting from left to right, you can see part of the list of cards, then the list of subcards for a given feature, behaviors in the figure, and the example page about a specific feature (one specific behavior, in this case). When topics and features are discussed, keep in mind this way of accessing examples about features, as this will be assumed later on.

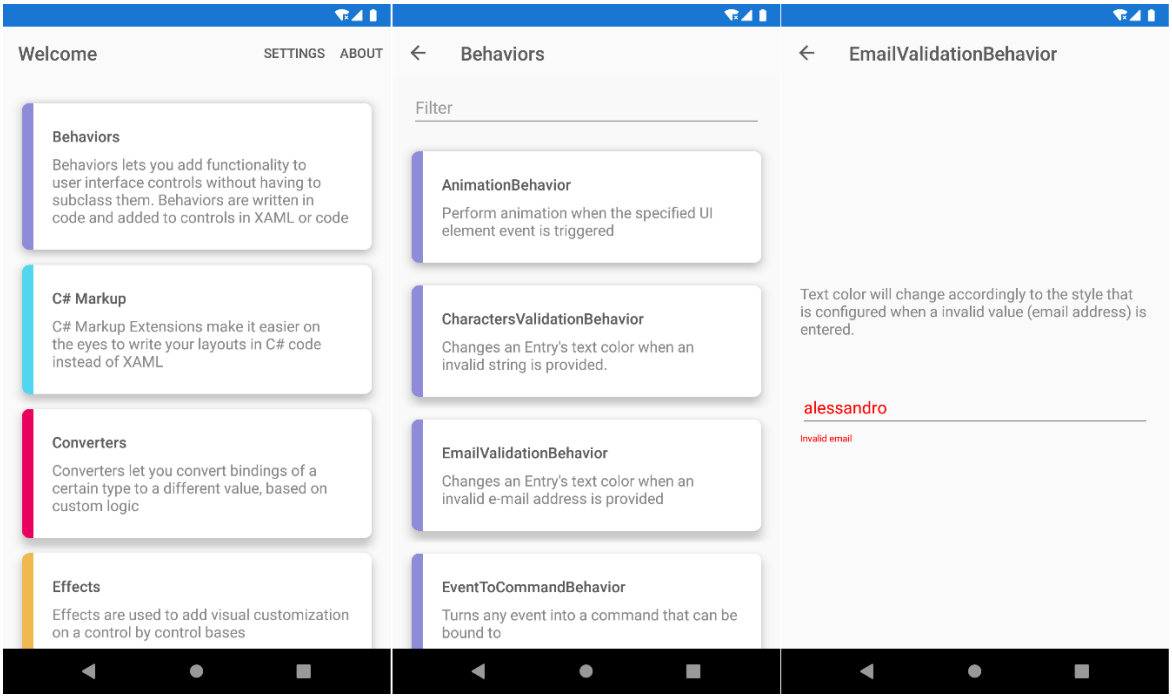


Figure 4: The official sample app running

In addition to discovering and running examples, it is a good idea to define some conventions that will be used across the book, and that will simplify your reading.

Conventions used in this book

In the next chapter, you will start looking at all the available elements in the Xamarin Community Toolkit, so it is useful to provide you with some indications that are shared across the book, and that will not be repeated every time. Because each element is presented with an example in the official sample app, I will always point you to the page in the app, and I will always tell you where to find the XAML page files in the project.

By *project*, unless where expressly specified, I mean the shared Xamarin.CommunityToolkit.Sample project in the solution. The XAML page files are located in the **Pages** folder of the shared project. Each page populates its own data context with a dedicated **`viewmodel`** class. ViewModel .cs files are located within specific subfolders of the **ViewModels** folder of the project. They are declared and assigned directly in the XAML of the page, and they are identified via either the **`vm:`** or the **`viewmodel:`** XML namespaces. ViewModel names and page names are based on the name of the object currently discussed; for example, for the **`ImageResourceConverter`** class, the sample page is called ImageResourceConverterPage.xaml, and the ViewModel is called ImageResourceConverterViewModel.cs. Pages that use views, converters, behaviors, or any other feature provided by the Xamarin Community Toolkit will always include the following namespace declaration.

```
xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
```

In this way, a view like the **MediaElement** will be accessed as follows.

```
<xct:MediaElement />
```

Now that you know what the **xct** identifier refers to, it will not be repeated in the next chapters, as well as all the other indications, for the sake of simplicity.

Chapter summary

The Xamarin Community Toolkit is an open-source project backed by Microsoft that provides common, reusable elements for Xamarin.Forms. In this chapter, you got an introduction to the library, how to get the sample source code that will be used in the book, and how to set up the development environment. Next, you got some indications about conventions used in the book that will make your reading better and clearer. It is now time to start working with all the goodies that the Xamarin Community Toolkit has to offer, and certainly the more natural way to do so is by looking at new views.

Chapter 2 Working with Views

The Xamarin Community Toolkit includes several views that are of common use in mobile apps but that were not available in the Xamarin.Forms code base, filling the gap with native development. Views also include new layouts, which extend the possibilities of arranging the user interface. This chapter describes layouts and views offered by the library, using the official sample project as the starting point, but with many considerations about practical use.

New layouts

Three new layouts are available in the Xamarin Community Toolkit: **`DockLayout`**, **`StateLayout`**, and **`UniformGrid`**. Actually, as you'll discover shortly, the **`StateLayout`** is rather a collection of attached properties, but it is classified as a layout. This section describes both, and the sample pages are located under the Pages\Views folder of the official sample project.

Docking contents with DockLayout

`DockLayout` is a new layout that allows for docking child visual elements in all the four directions (top, bottom, left, right). If you run the sample app and look at Figure 5, you can see how several visual elements (**`Button`** views, to be precise) are docked in different positions.

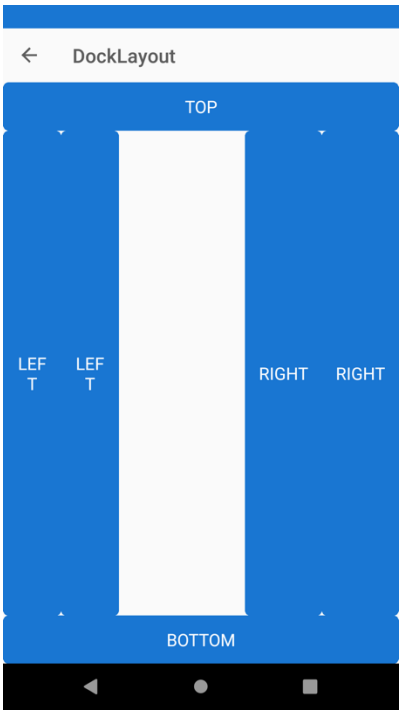


Figure 5: Docking contents on four directions

The XAML code for this example is the following.

```
<xct:DockLayout
       LastChildFill="False">
       <Button xct:DockLayout.Dock="Top" Text="Top" HeightRequest="50"/>
       <Button xct:DockLayout.Dock="Bottom" Text="Bottom"
               HeightRequest="50"/>
       <Button xct:DockLayout.Dock="Left" Text="Left" WidthRequest="60"/>
       <Button xct:DockLayout.Dock="Left" Text="Left" WidthRequest="60"/>
       <Button xct:DockLayout.Dock="Right" Text="Right" WidthRequest="80"/>
       <Button xct:DockLayout.Dock="Right" Text="Right" WidthRequest="80"/>
</xct:DockLayout>
```

These are the relevant points:

- You assign the dock direction to each view via the **`DockLayout.Dock`** attached property, and possible values are **`Top`**, **`Left`**, **`Bottom`**, and **`Right`**.
- The **`LastChildFill`** property allows you to decide whether the last visual element in the collection should fill the remaining space in the center.
- If multiple views are docked in the same direction, they will be docked to one another (see Figure 5 about the last two right-aligned buttons).

`DockLayout` is extremely useful and adds a lot of flexibility to the user interface, so this is really a great addition.

State-aware views with StateLayout

Sometimes you might need to display specific views when the app is in a specific state, for example, displaying loaders when the app is busy doing something, or an error message when something went wrong. With the **`StateLayout`** control, you can turn any layout element, like a **`Grid`** or **`StackLayout`**, into an individual state-aware element.

Each layout that you make state-aware, using the **`StateLayout`** attached properties, contains a collection of **`StateView`** objects. These objects can be used as templates for the different states supported by **`StateLayout`**. Whenever the **`CurrentState`** property is set to a value that matches the **`State`** property of one of the **`StateViews`**, its contents will be displayed instead of the main content.

This view is demonstrated in the Pages\Views\StateLayoutPage.xaml, and if you run the sample app, you can press the **Cycle All States** button to get an example of how each state can be used to display a specific view for each state. Figure 6 shows an example.

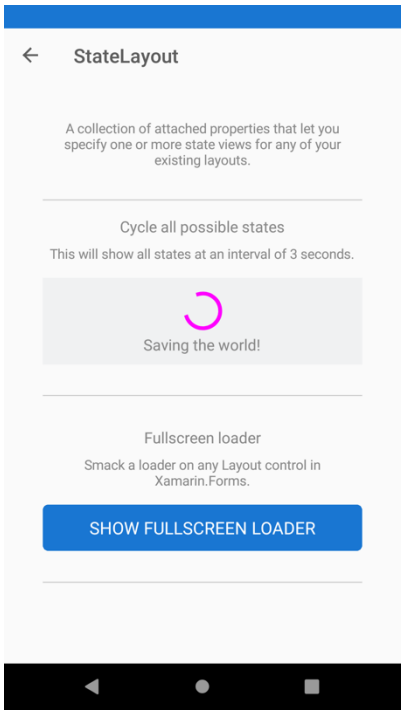


Figure 6: Assigning views to a specific state

Possible states are defined in the **`LayoutState`** enumeration, and values can be: **`None`**, **`Loading`**, **`Saving`**, **`Success`**, **`Error`**, **`Empty`**, and **`Custom`**. For example, let's consider the first piece of the XAML code that defines a **`StateView`** for a **`Grid`**, to be used when the state is loading.

```
<Grid xct:StateLayout.CurrentState="{Binding MainState}"
      xct:StateLayout.AnimateStateChanges="true">
    <xct:StateLayout.StateViews>
        <xct:StateView StateKey="Loading"
                       BackgroundColor="White"
                       VerticalOptions="FillAndExpand">
            <StackLayout VerticalOptions="Center"
                         HorizontalOptions="Center">
                <ActivityIndicator Color="#1abc9c"
                                   IsRunning="{Binding MainState,
                    Converter={StaticResource StateToBooleanConverter},
                            ConverterParameter={x:Static
                            xct:LayoutState.Loading}}" />
                <Label Text="Loading..." HorizontalOptions="Center" />
              </StackLayout>
          </xct:StateView>
      </xct:StateLayout.StateViews>
</Grid>
```

As you can see, the **Grid** uses attached properties of the **StateLayout**. In the example, the **CurrentState** property is bound to a property called **MainState** of type **LayoutState**, and is defined in the backing **StateLayoutViewModel** class. It represents the current state of the view, and when it's **None**, it's in a default state. By specifying the collection of **StateViews**, you can decide what happens when a state changes.

In this case, a **StateView** object specifies that if the state changes to **Loading**, the content of the **Grid** must be replaced by a **StackLayout** containing an **ActivityIndicator**. When the state changes back to **None**, the **Grid** will also return to display its original content. The type of state can be supplied by assigning the **StateKey** property of the **StateView** with one of the values from the **LayoutState** enumeration.

Notice how the **ActivityIndicator.IsRunning** property is bound to the current state and converted from **LayoutState** to **bool** via the **StateToBooleanConverter** class. The latter returns **true** if the value of the bound property (**MainState** in this case) equals the state supplied via the **ConverterParameter**.

The other states in the example work similarly, but with a different state. The key point is that you can supply a view for a different state directly inside the current view, without the need to implement custom logic for state changes and without the need to create a more complex UI hierarchy.

You can also implement custom states by using the **Custom** value from the **LayoutState** enumeration in combination with the **CustomStateKey** property of the **StateView** object. An example is available in the XAML code of the **StateLayoutPage.xaml** file.

```
<xct:StateView StateKey="Custom" CustomStateKey="ThisIsCustomToo">
    <Label Text="Hi, I'm a custom state too!" VerticalOptions="Center"
           VerticalTextAlignment="Center"
           HorizontalOptions="Center" HorizontalTextAlignment="Center"  />
</xct:StateView>
```

The binding will then work exactly as explained in the previous steps.

Implementing uniform grids

The third and last view offered by the Xamarin Community Toolkit is the **UniformGrid**. This is a simplified **Grid** with rows and columns of the same size. The usage, demonstrated in the Pages\Views\UniformGridPage.xaml file, is extremely simple, and the sample code is the following.

```
<xct:UniformGrid>
    <BoxView Color="Red" />
    <BoxView Color="Yellow" />
    <BoxView Color="Orange" />
    <BoxView Color="Purple" />
    <BoxView Color="Blue" />
    <BoxView Color="Green" />
    <BoxView Color="LightGreen" />
```

```
    <BoxView Color="Gray" />
    <BoxView Color="Pink" />
</xct:UniformGrid>
```

The result is visible in Figure 7. Notice that, unlike in the regular **`Grid`**, it is not possible to specify the **`RowDefinitions`** and **`ColumnDefinitions`** collections, because the **`UniformGrid`** automatically arranges its content wrapping and alignment as necessary.

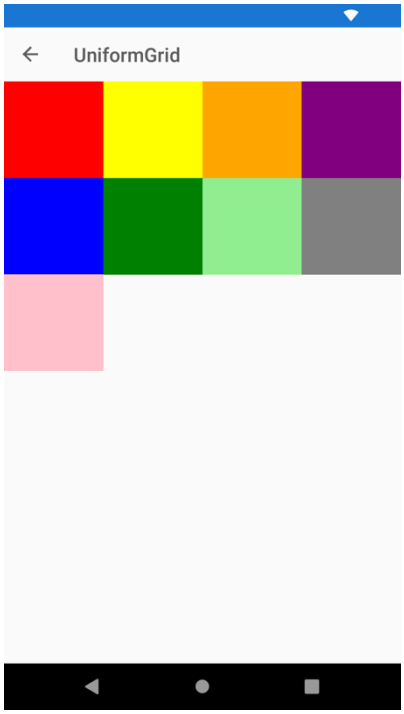


*Figure 7: The **`UniformGrid`** in action*

New views

New design standards have risen in the recent years, and consequently, new views have entered into common mobile app designs as well. The Xamarin Community Toolkit bridges the gap between the Xamarin.Forms code base and such new designs, introducing views that are now common in mobile apps, and that will help you be more productive by avoiding spending time on creating custom views.

Displaying profile pictures with the AvatarView

An avatar identifies a profile picture of a person or contact, and when the image is not available, the avatar usually displays the initials of the person's name. There are many programs that use avatars to identify people or contacts; Microsoft Outlook is an example.

The Xamarin Community Toolkit makes it easy to use avatars in your apps by offering the **`AvatarView`** control. For an understanding of how it works, look at Figure 8, which shows the sample app in action on the Pages\Views\AvatarViewPage.xaml file.

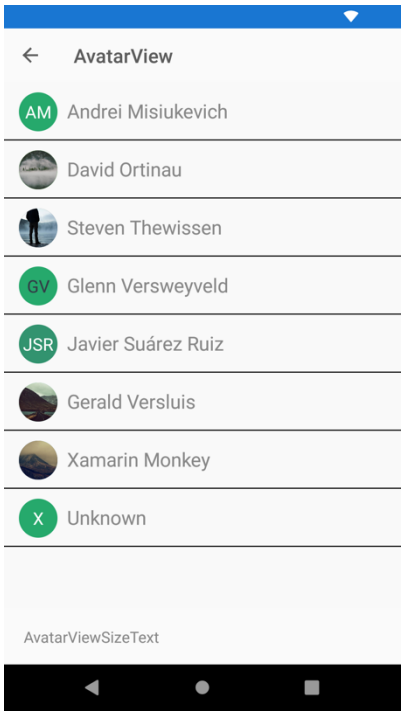


*Figure 8: The **`AvatarView`** in action*

The app shows a list of people, some of whom are real people currently working at Microsoft, whereas the Xamarin Monkey and Unknown are test data. Pictures are assigned to the **`AvatarView`** via the public URL, and you will shortly see how. The user interface of the page mainly consists of a **`CollectionView`**, which is populated with the **`Items`** property of the **`AvatarViewViewModel`** class, whose code is represented in Code Listing 1.

Code Listing 1

```
namespace Xamarin.CommunityToolkit.Sample.ViewModels.Views
{
      public class AvatarViewViewModel : BaseViewModel
      {
            public object[] Items { get; } =
            {
                  new { Initials = "AM",
                        Source = string.Empty,
                        Name = "Andrei Misiukevich" },
                  new { Initials = "DO",
                        Source =
                      "https://picsum.photos/500/500?image=472",
                        Name = "David Ortinau" },
                  new { Initials = "ST",
```

```
                    Source =
                "https://picsum.photos/500/500?image=473",
                    Name = "Steven Thewissen" },
            new { Initials = "GV",
                    Source = string.Empty,
                Name = "Glenn Versweyveld" },
            new { Initials = "JSR",
                    Source = string.Empty,
                Name = "Javier Suárez Ruiz" },
            new { Initials = "GV",
                    Source =
                "https://picsum.photos/500/500?image=474",
                    Name = "Gerald Versluis" },
            new { Initials = "XM",
                    Source =
                "https://picsum.photos/500/500?image=475",
                    Name = "Xamarin Monkey" },
            new { Initials = string.Empty,
                Source = string.Empty,
                    Name = "Unknown" }
        };
    }
}
```

The **Items** property is of type **object[]** and contains a collection of anonymous types. This has been probably implemented in this way for the sake of speed, but in the real world, you will want to expose a strongly typed **ObservableCollection**.

Each object instance allows for specifying an image URL (**Source** property), name initials if an image is not available (**Initials** property), and the full name (**Name** property). In the XAML code of the page, the **AvatarView** represents the **DataTemplate** of the **CollectionView**, and is declared as follows.

```
<xct:AvatarView ColorTheme="{x:Static xct:ColorTheme.Jungle}"
                FontSize="Medium"
                Size="{Binding Value,
                Source={x:Reference Slider}}"
                Text="{Binding Initials}">

    <xct:AvatarView.Source>
        <UriImageSource Uri="{Binding Source}" />
    </xct:AvatarView.Source>
</xct:AvatarView>
```

The most relevant points in this code are the following:

- The **Text** property allows for displaying the name initials only when the image is not available.

- The **`Source`** property gets the image to be displayed. In this particular example, it is retrieved under the form of a **`UriImageSource`** object since you pass a URI.
- The **`ColorTheme`** property allows for assigning one of the predefined color themes implemented in the Xamarin Community Toolkit, and for which an introduction will be given later in this chapter.

If no image and no initials are available, the **`AvatarView`** will simply display an X character, which is the initial of Xamarin.

Informational badges with BadgeView

The **`BadgeView`** allows you to display overlayed icons on a view, known as badges, and it is typically used to display new notifications such as the number of new contents in the app. The **`BadgeView`** is demonstrated in the Pages\Views\BadgeViewPage.xaml of the sample project, and Figure 9 shows several instances in action.

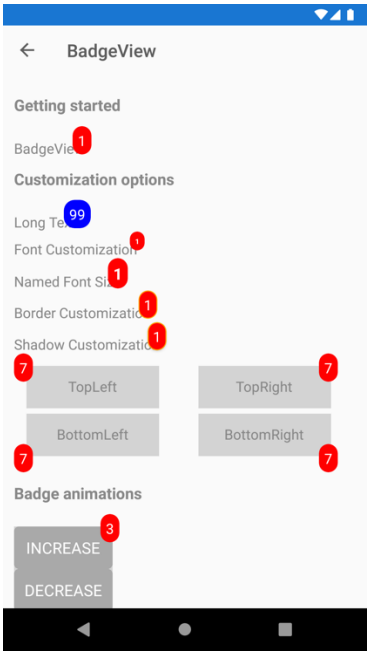


*Figure 9: The **`BadgeView`** in action*

As you can see, you can use this view to show how many new notifications the user got in the app. We can discuss how it works starting from the first instance at the top of the XAML, and then adding information incrementally. At a glance, the **`BadgeView`** can be thought of as a transparent container for another view, on which it overlays a badge with the string you want to display. In terms of XAML, this works as follows.

```
<xct:BadgeView
    BackgroundColor="Red" TextColor="White" Text="1">
    <Label
        Text="BadgeView"/>
</xct:BadgeView>
```

As you can see, the **Label** that displays the notification is surrounded by the **BadgeView**. You can specify the background color and the text color. The **Text** property is of type **string**, so you are not limited to displaying only numbers. You do not necessarily need to pass something, so you can also use an empty string if your purpose is only attracting the attention of the user.

Let's now walk through possible customizations you can do over the control, and that you will be able to find in the continuing XAML code.

Font customization

You can customize the font of the **BadgeView** via the **FontFamily**, **FontAttributes**, and **FontSize** properties, as you would do with any other view supporting text. **FontSize** also supports named size values (such as **Normal**, **Medium**, and **Large**).

Border customization

You can assign the **BorderColor** property with a value of type **Color** and set a different color for the border of the **BadgeView**, so you can have different border and background colors. In addition, you can set the **HasShadow** property with **true** if you want to display a shadow below the control. The default is **false**.

Specifying a position

By default, the badge is placed at the top-right corner. However, it is possible to specify a different position by assigning the **BadgePosition** property with one of the values from the **BadgePosition** enumeration, all self-explanatory, which can be **TopLeft**, **TopRight**, **BottomLeft**, or **BottomRight**. Figure 9 has clear examples of repositioning the badge.

Overlaying complex views

The child content of a **BadgeView** can be any visual element, which means you can design a layout (such as a **Grid** or **StackLayout**) and create a more complex view, and badges will work exactly as expected.

Capturing videos and photos

In the past, you could add camera capabilities to your Xamarin.Forms projects using the Media plugin. This allowed you to capture videos and photos from your device's camera and did an excellent job, but it was not optimized for .NET Standard, and was in the charge of an individual developer, which means it could be maintained only when and if possible.

The Xamarin Community Toolkit brings back camera capabilities, and it integrates perfectly with the latest version of Xamarin.Forms—plus, it is fully backed by Microsoft. Camera support is offered by the **CameraView** control, which allows for capturing videos and images from your device. It is demonstrated in the Pages\Views\CameraView.xaml file. The result of this sample page is visible in Figure 10. Notice that it is based on the simulator, so the camera is presenting a fake image.

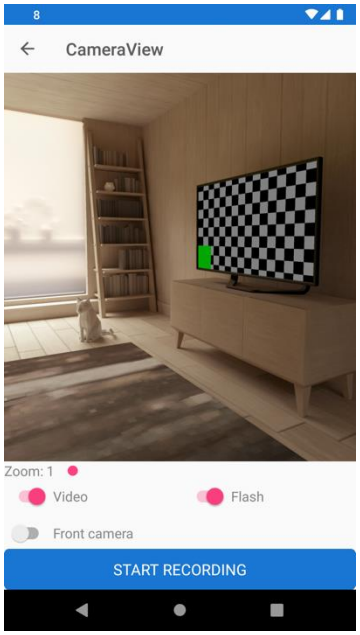


Figure 10: Capturing media from your device

You can control several options of the camera and record the captured content, and now you will see how. In the sample XAML code, the **`CameraView`** is defined as follows.

```
<xct:CameraView
    x:Name="cameraView"
    CaptureMode="Video"
    FlashMode="On"
    HorizontalOptions="FillAndExpand"
    MediaCaptured="CameraView_MediaCaptured"
    OnAvailable="CameraView_OnAvailable"
    VerticalOptions="FillAndExpand" />
```

The **`CaptureMode`** property can be assigned with **`Video`**, **`Photo`**, or **`Default`** (which goes for taking a picture if the property is not specified). The **`FlashMode`** option can be assigned with **`On`** or **`Off`**. When the camera starts running on the device, the **`OnAvailable`** event is raised. In terms of camera availability, you might want to check in your apps whether the value of the read-only **`CameraView.IsAvailable`** property returns **`true`**.

In the sample code-behind for the page, the code sets the camera zoom based on the value of a **`Slider`** defined in the user interface. The **`CameraView`** exposes two properties for managing the zoom, both of type **`double`**. **`Zoom`** represents the current zoom value for the camera, and **`MaxZoom`** represents the maximum zoom value for the camera.

Another interesting property is **`CameraOptions`**, which allows for selecting the current camera, and whose value can be **`Default`**, **`Back`**, **`Front`**, or **`External`**. **`Default`** matches the default camera settings of the device.

The **IsBusy** property is also interesting and returns **true** if the camera is busy capturing media and cannot be used. When working with media content, the **CameraView** allows you to handle two events: **MediaCaptured** and **MediaCaptureFailed**. The first one is raised once the user stops capturing content, which means taking a picture or recording a video.

The code-behind for the sample page shows how to handle the event as follows.

```
void CameraView_MediaCaptured(object? sender,
      MediaCapturedEventArgs e)
{
      switch (cameraView.CaptureMode)
      {
            default:
            case CameraCaptureMode.Default:
            case CameraCaptureMode.Photo:
                  previewPicture.IsVisible = true;
                  previewPicture.Rotation = e.Rotation;
                  previewPicture.Source = e.Image;
                  doCameraThings.Text = "Snap Picture";
                  break;
            case CameraCaptureMode.Video:
                  previewPicture.IsVisible = false;
                  doCameraThings.Text = "Start Recording";
                  break;
      }

}
```

If the camera is capturing a picture, an image view called **previewPicture** is made visible, and its **Source** property is assigned with the captured image, which is stored in the **Image** property of the **MediaCapturedEventArgs** object instance (and saved on the device if the camera settings allow for this).

The user interface also defines a **Button** called **doCameraThings**, whose text is changed based on the selection of taking a picture. Before discussing how things work when working with videos, let's have a look at the XAML code for both views.

```
<Button
    x:Name="doCameraThings"
    Command="{Binding ShutterCommand, Source={x:Reference cameraView}}"
    IsEnabled="False" Text="Start Recording" />
<Image
    x:Name="previewPicture"
    Aspect="AspectFit" BackgroundColor="LightGray"
    HeightRequest="250" IsVisible="False" />
```

The **Command** property of the **Button** is bound to the **ShutterCommand** property of the **CameraView**. This command is invoked when the shutter is triggered. Regarding videos, when the user stops capturing, the media content is just saved on the device.

Tip: Remember that a real application must ask for the user's permission before accessing the camera. This can be quickly accomplished via the Permissions class from the Xamarin Essentials library.

Grouping views with the Expander

The **Expander** is a view that can contain a hierarchy of visual elements, display them when expanded, and hide them when collapsed. In mobile app designs, this kind of view is also known as accordion. In the sample project, it is demonstrated in the Pages\Views\ExpanderPage.xaml file. If you look at the source code, you can see the following declaration.

```
<xct:Expander ExpandAnimationEasing="{x:Static Easing.CubicIn}"
              CollapseAnimationEasing="{x:Static Easing.CubicOut}"
              IsExpanded="{Binding IsExpanded}"
              Command="{Binding BindingContext.Command,
                        Source={x:Reference page}}"
              CommandParameter="{Binding .}">
```

With the **ExpandAnimationEasing** and **CollapseAnimationEasing** properties, you can assign the animation that runs when the **Expander** is expanded or collapsed. The **IsExpanded** property, of type **bool**, allows for programmatically controlling the status of the **Expander**.

You can also execute an action when the **Expander** is engaged via the **Command** property and its **CommandParameter** value. When the **Expander** is collapsed, the view displays the value of its **Header** property, defined as follows.

```
<xct:Expander.Header>
    <StackLayout Orientation="Horizontal" Spacing="0">
        <Label Text="{Binding Name}"
               HorizontalOptions="FillAndExpand"
               FontSize="32"
               FontAttributes="Bold"/>
        <Label Text="Enable nested:"
               FontSize="13"
               VerticalOptions="CenterAndExpand" />
        <Switch IsToggled="{Binding IsEnabled}" />
    </StackLayout>
</xct:Expander.Header>
```

The value of the **Header** is an individual view, which can also be a layout for a more complex visual hierarchy. When the **Expander** is expanded, the view displays the value of its **Content** property. This is an implicit property, so you can omit declaring it explicitly, and the view will display the visual element included between the enclosing tags as follows.

```
<xct:Expander>
     <!-- Content goes here… -->
</xct:Expander>
```

In the sample code, the **Content** is another **Expander** to demonstrate nested expanders. Notice that the **Content** property should only include an individual **View** object. If you wish to use a layout for a more complex visual hierarchy, you can use the **ControlTemplate** property, in which you define a **DataTemplate** that contains your structure.

```
<xct:Expander.ContentTemplate>
    <DataTemplate>
        <StackLayout Spacing="0" Margin="10"
                     Padding="1" BackgroundColor="Black">
            <BoxView HeightRequest="50" Color="White" />
            <BoxView HeightRequest="50" Color="Red" />
            <BoxView HeightRequest="50" Color="White" />
        </StackLayout>
    </DataTemplate>
</xct:Expander.ContentTemplate>
```

Figure 11 shows how the sample page appears.

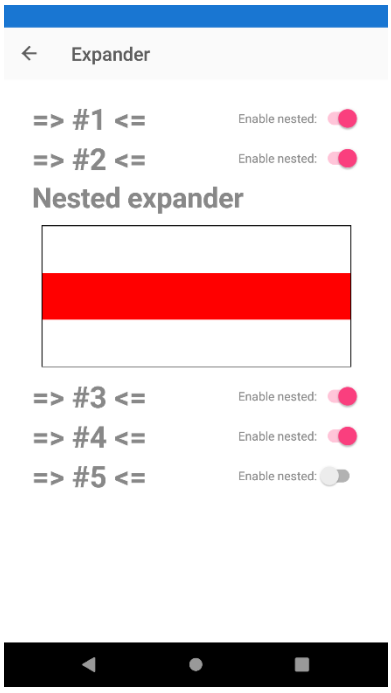


Figure 11: Displaying contents with the Expander

One common implementation of the header is using arrow icons. For instance, an up-arrow icon indicates a collapsed state, and a down-arrow icon indicates an expanded state.

Displaying certified profile pictures with GravatarImageSource

Gravatar is an online service for providing globally unique avatars, and it is widely used to create certified profile pictures. By creating a profile picture on Gravatar, you can be sure that your identity is safe. The profile picture is strictly related to your email address, and this is an important point to highlight, as you will see shortly.

The Xamarin Community Toolkit allows for displaying certified profile pictures from Gravatar via the **`GravatarImageSource`** class, which is demonstrated in the Pages\Views\GravatarImagePage.xaml file of the sample project. This object retrieves an image from Gravatar and returns an **`ImageSource`** object that can be assigned to an Image view.

A XAML markup extension is also available, and you are going to see both options in action. A good idea is first having a look at the sample page in action on the device, shown in Figure 12.

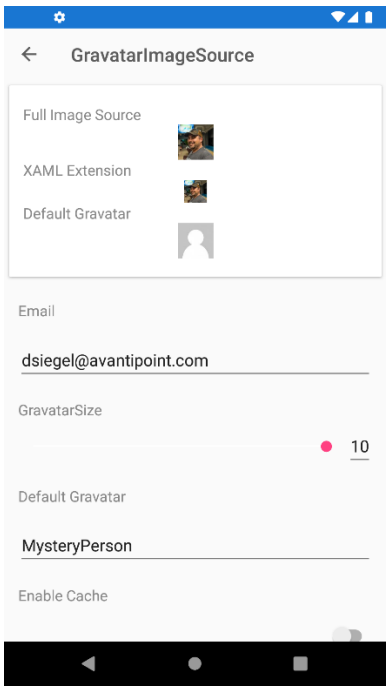


Figure 12: Displaying profile pictures from Gravatar

As you can see, an email address is specified, and it is how the **`GravatarImageSource`** can download the profile picture from Gravatar. In the example, you see the same picture twice, but in the XAML this happens with a different syntax. You also see a default picture, which you can use in several situations, such as download errors.

If you now look at the XAML code of the page, you will see that the first image is displayed via the following markup.

```
<Image>
    <Image.Source>
        <xct:GravatarImageSource Email="{Binding Email}"
                                 Size="{Binding Size}"
                                 CachingEnabled="{Binding EnableCache}" />
```

```
    </Image.Source>

</Image>
```

As you can see, the **GravatarImageSource** object is populating the **ImageSource** property of the **Image** view, and the source for the picture is specified via the **Email** property. In the sample project, the email is exposed by the **Email** property of the **GravatarImageViewModel** class, and it is of type **string**.

You can also specify a size for the picture, with the **Size** property of type **int** whose value can be between 0 and 100. You can also cache the image via the **CachingEnabled** property in order to optimize memory. Both **Size** and **EnableCache** binding properties are exposed by the ViewModel. As an alternative, you can use the **GravatarImage** markup extension, which offers an inline syntax and works as follows.

```
<Image Source="{xct:GravatarImage {Binding Email},
    Size=65, CachingEnabled={Binding EnableCache}}" />
```

The key point here is that you can use the **GravatarImage** markup extension directly in the binding specification for the **Image.Source** property. It is also possible to provide a default picture in case the **GravatarImageSource** is not able to retrieve the proper one, and this is possible by assigning the **Default** property with an object of type **DefaultGravatar**.

The sample XAML is the following.

```
<Image Source="{xct:GravatarImage '',
    Default={Binding DefaultGravatar},
    Size={Binding Size}, CachingEnabled={Binding EnableCache}}" />
```

In the example, the source for the image is intentionally empty so that the value of the **Default** property is used, as you can also see in Figure 12 with the third image from the top. The **DefaultGravatar** object is an enumeration defined as follows.

```
public enum DefaultGravatar
{
      FileNotFound,
      MysteryPerson,
      Identicon,
      MonsterId,
      Wavatar,
      Retro,
      Robohash,
      Blank
}
```

In the **GravatarImageViewModel** class, the **DefaultGravatar** property is initialized with the **MysteryPerson** value, but the user interface of the sample app allows you to select a different one via a **Picker** so that you can get a preview of how each default image looks. With this approach, you can quickly display a certified picture for a person and, if not available, you can warn your users with a default image.

Playing audio and video

The Xamarin Community Toolkit provides a new view called **`MediaElement`**, which allows for playing audio and video in your applications. Actually, during its development, the **`MediaElement`** control was included in the Xamarin.Forms code base, but now it has been moved to the Xamarin Community Toolkit library.

`MediaElement` can play media content from a remote URI, from the local library, from media files embedded inside the app resources, and from local folders. In the sample project, it is demonstrated in the Pages\Views\MediaElementPage.xaml file, and it is based on a free, public Microsoft video about Xamarin that can be streamed online.

Figure 13 shows the **`MediaElement`** in action. Notice how the player shows buttons that allow for controlling the media reproduction, such as Play and Pause.

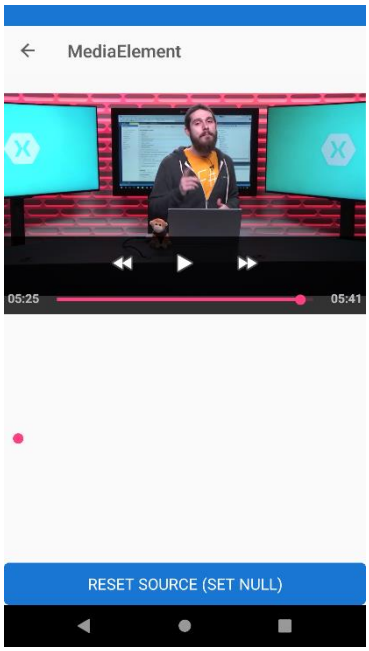


Figure 13: Playing audio and video

In the XAML, it is declared as follows.

```
<xct:MediaElement
    x:Name="mediaElement"
    Source="https://sec.ch9.ms/ch9/5d93/a1eab4bf-3288-4faf-81c4-
294402a85d93/XamarinShow_mid.mp4"
    ShowsPlaybackControls="True" MediaOpened="OnMediaOpened"
    MediaFailed="OnMediaFailed" MediaEnded="OnMediaEnded"
    HorizontalOptions="Fill" SeekCompleted="OnSeekCompleted" />
```

The **`Source`** property contains the URI of the media file. As you can learn through the official documentation, local files are also represented by URIs that start with the **ms-appx:///** or **ms-appdata:///** prefixes. The **`ShowsPlaybackControls`** property allows you to avoid the need to

create playback controls manually and will make the **MediaElement** use the playback control of each native platform.

You certainly have the option to create your custom controls and use data binding to provide a different look and feel to your player, but this is out of the scope of this chapter. You can make media start automatically by setting the **AutoPlay** property with **true**, and you can control the media volume using the **Volume** property, of type **double**. Its value must be between 0 and 1.

The **MediaElement** also exposes the **Aspect** property, which controls the stretching of the video, and supported values are **Fill**, **AspectFill**, and **AspectFit**. The **Duration** property, of type **TimeSpan?**, returns the duration of the currently opened media, while the **Position** property, of type **TimeSpan**, returns the current progress over the duration.

The **MediaElement** view also exposes self-explanatory methods such as **Play**, **Stop**, and **Pause** that you can invoke in your C# code to manually control the media file. Additionally, among others, this view exposes events like:

- **MediaOpened**: Raised when the media stream has been validated and is ready for playing.
- **MediaEnded**: Raised when the media stream reaches its end.
- **MediaFailed**: Raised when an error occurs on the media source.
- **SeekCompleted**: Raised when you move the reproduction to a different position.

The **MediaElement** is at the same time a very versatile view in its simplest form, and a completely customizable view for high-quality media playing designs.

Displaying live status information with shields

The **Shield** is a view that allows for displaying information about the status of a service or a call to action in a badge-like way. Shields are very popular nowadays; for example, they are used to display information on software status on websites like NuGet, GitHub, and Azure Pipelines. With the Xamarin Community Toolkit, you can create shields as demonstrated in Figure 14.

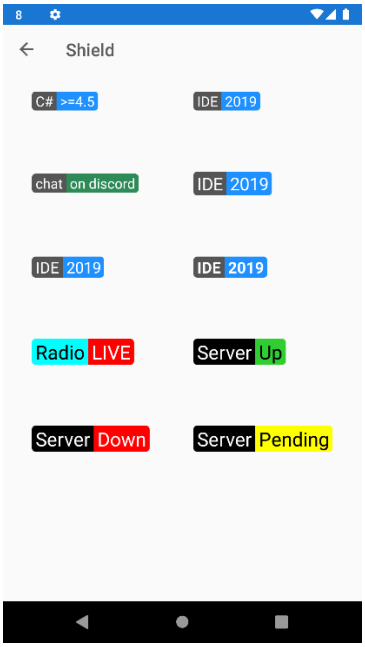


Figure 14: Creating shields

In the sample project, the **Shield** view is demonstrated in the Pages\Views\ShieldPage.xaml file. A shield is made of two parts: the subject and the status. The subject is the left part and represents the name of the item to which the shield refers, whereas the status provides information on the current status of the item. Both parts are represented by properties in the **Shield** view, called **Subject** and **Status** respectively, both of type **string?**.

There are many **Shield** declarations in the XAML code for the page, but we will consider just two of them, since they all work in the same way—just with different property values. The first **Shield** to consider is also the first one in the code and is declared as follows.

```
<xct:Shield Grid.Row="0" Grid.Column="0"
            Subject="C#"
            Status=">=4.5"
            StatusBackgroundColor="DodgerBlue"
            StatusTextColor="White"
            Tapped="OnShieldTapped" />
```

The **Subject** and **Status** properties are assigned with the strings you want to display in the two parts. You can customize colors for both parts, as well. For the status, you can use the **StatusBackgroundColor** property to customize the background and the **StatusTextColor** to customize the foreground color of the status string. Similarly, you can assign the **SubjectBackgroundColor** and **SubjectTextColor** properties to customize the subject background color and foreground color, respectively.

The **Shield** supports interaction; in fact, it exposes the **Tapped** event. In the sample project, the event is simply handled to display an alert as follows.

```
async void OnShieldTapped(object? sender, EventArgs e)
      => await DisplayAlert("Shield Event", "C# Shield Tapped", "Ok");
```

The second **Shield** that is taken into consideration is the following.

```
<xct:Shield Grid.Row="3" Grid.Column="0"
            Subject="Radio"
            StatusBackgroundColor="Red"
            SubjectBackgroundColor="Cyan"
            SubjectTextColor="Black"
            Status="LIVE"
            StatusTextColor="White"
            FontSize="Large" />
```

In this example, you see color customization for the subject as described previously, plus you see how to customize the font size. You can use both named font sizes and numbers. Also, you can use the **FontFamily** and **FontAttributes** properties to further customize the font appearance as you would do with any other control that supports text.

Shields are probably not very common in mobile apps, but if your project needs to display the status of some services, then they can be a valid choice with no effort.

Customizable sliders with RangeSlider

The **RangeSlider** view is an evolved, highly customizable version of the well-known **Slider**. It shows two thumbs that allow for selecting numerical values in a range. It is demonstrated in the Pages\Views\RangeSliderPage.xaml file.

If you run the sample app and enter the related example, you will be able to play with the **RangeSlider** by customizing its appearance. All the possible options are offered in the page, as shown in Figure 15, where you can see possible customizations available by scrolling the page.

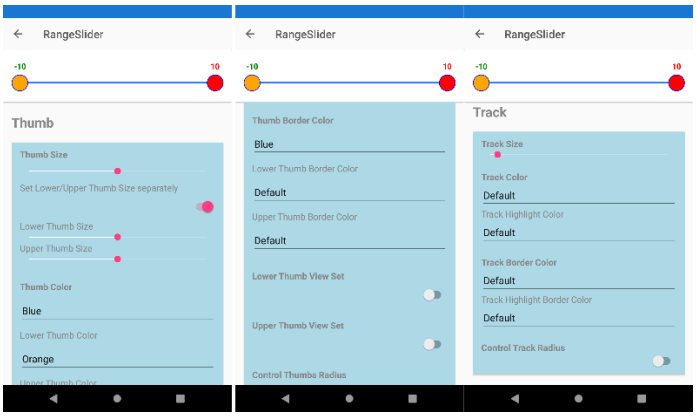


*Figure 15: Customizing a **RangeSlider***

For example, you can customize the thumbs with any **View**; you can change the size and color of the sliding track, the size and color of the thumbs, and the corner radius of the thumbs.

In the XAML for the sample page, most of the **`RangeSlider`** properties are data-bound to other views in the page, such as pickers that allow for selecting different colors, as you might have seen if you have played with the example in the app. Code Listing 2 shows how it is declared.

Code Listing 2

```
<xct:RangeSlider
    x:Name="RangeSlider"
    MaximumValue="10"
    MinimumValue="-10"
    StepValue="0.01"
    LowerValue="-10"
    UpperValue="10"
    ValueLabelStringFormat="{StaticResource
          CustomValueLabeStringFormat}"
    LowerValueLabelStyle="{StaticResource
          CustomLowerValueLabelStyle}"
    UpperValueLabelStyle="{StaticResource
          CustomUpperValueLabelStyle}"
    ThumbSize="{Binding Value,
          Source={x:Reference ThumbSizeSlider}}"
    ThumbColor="{Binding SelectedItem,
          Source={x:Reference ThumbColorPicker}}"
    LowerThumbColor="{Binding SelectedItem,
          Source={x:Reference LowerThumbColorPicker}}"
    UpperThumbColor="{Binding SelectedItem,
          Source={x:Reference UpperThumbColorPicker}}"
    ThumbBorderColor="{Binding SelectedItem,
          Source={x:Reference ThumbBorderColorPicker}}"
    LowerThumbBorderColor="{Binding SelectedItem,
          Source={x:Reference LowerThumbBorderColorPicker}}"
    UpperThumbBorderColor="{Binding SelectedItem,
          Source={x:Reference UpperThumbBorderColorPicker}}"
    TrackSize="{Binding Value,
          Source={x:Reference TrackSizeSlider}}"
    TrackColor="{Binding SelectedItem,
          Source={x:Reference TrackColorPicker}}"
    TrackHighlightColor="{Binding SelectedItem,
          Source={x:Reference TrackHighlightColorPicker}}"
    TrackBorderColor="{Binding SelectedItem,
          Source={x:Reference TrackBorderColorPicker}}"
    TrackHighlightBorderColor="{Binding SelectedItem,
          Source={x:Reference TrackHighlightBorderColorPicker}}"
    IsEnabled="{Binding IsToggled,
          Source={x:Reference IsEnabledSwitch}}"
    ValueLabelSpacing="{Binding Value,
          Source={x:Reference ValueLabelSpacingSlider}}">

    <xct:RangeSlider.LowerThumbView>
```

```
        <Label Text="L" VerticalTextAlignment="Center"
               HorizontalTextAlignment="Center"
               IsVisible="{Binding IsToggled,
               Source={x:Reference LowerThumbViewSwitch}}" />
    </xct:RangeSlider.LowerThumbView>
    <xct:RangeSlider.UpperThumbView>
        <Label Text="U" VerticalTextAlignment="Center"
               HorizontalTextAlignment="Center"
               IsVisible="{Binding IsToggled,
               Source={x:Reference UpperThumbViewSwitch}}" />
    </xct:RangeSlider.UpperThumbView>
</xct:RangeSlider>
```

Table 1 provides a description of the properties used in the XAML.

*Table 1: **`RangeSlider`** properties*

Property	Type	Description
`LowerThumbBorderColor`	**`Color`**	Gets or sets the border color of the lower thumb.
`LowerThumbColor`	**`Color`**	Gets or sets the color of the lower thumb.
`LowerThumbRadius`	**`double`**	Gets or sets the corner radius of the lower thumb.
`LowerThumbSize`	**`double`**	Gets or sets the size of the lower thumb.
`LowerThumbView`	**`View`**	Gets or sets a custom view to be used for the lower thumb.
`LowerValue`	**`double`**	Gets or sets the lower value of the range.
`LowerValueLabelStyle`	**`Style`**	Gets or sets the style used for the value label on the lower thumb.
`MaximumValue`	**`double`**	Gets or sets the maximum value for the range that can be selected with the **`RangeSlider`**.
`MinimumValue`	**`double`**	Gets or sets the minimum value for the range that can

Property	Type	Description
		be selected with the **RangeSlider**.
StepValue	**double**	Gets or sets the increment by which **MaximumValue** and **MinimumValue** change.
ThumbBorderColor	**Color**	Gets or sets the border color of both the lower and upper thumbs.
ThumbColor	**Color**	Gets or sets the color for both the lower and upper thumbs.
ThumbSize	**double**	Gets or sets size for both the lower and upper thumbs.
TrackBorderColor	**Color**	Gets or sets the color of the track bar border.
TrackColor	**Color**	Gets or sets the color of the track bar.
TrackHighlightBorderColor	**Color**	Gets or sets the border highlight color of the track bar, which is the part between the lower and the upper thumbs.
TrackHighlightColor	**Color**	Gets or sets highlight color of the track, which is the part between the lower and the upper thumbs.
TrackRadius	**double**	Gets or sets the corner radius of the track.
TrackSize	**double**	Gets or sets the size of the track bar.
UpperThumbBorderColor	**Color**	Gets or sets the border color of the upper thumb.
UpperThumbColor	**Color**	Gets or sets the color of the upper thumb.
UpperThumbRadius	**double**	Gets or sets the corner radius of the upper thumb.

Property	Type	Description
UpperThumbSize	**double**	Gets or sets the size of the upper thumb.
UpperThumbView	**View**	Gets or sets a custom view to be used for the upper thumb.
UpperValue	**double**	Gets or sets the upper value of the range.
UpperValueLabelStyle	**Style**	Gets or sets the style used for the value label on the upper thumb.
ValueLabelStringFormat	**double**	Gets or sets the string format used for the value labels on both the lower and upper thumbs.
ValueLabelSpacing	**double**	Gets or sets the spacing of the font used for the value labels on both the lower and upper thumbs.
ValueLabelStyle	**Style**	Gets or sets the style used for the value labels on both the lower and upper thumbs.

Properties described in Table 1 are bindable properties, so for example, you could bind the **Value** to a command in the ViewModel and handle the value change. However, the **RangeSlider** also exposes events. These are summarized in Table 2.

*Table 2: **RangeSlider** events*

Event	Description
ValueChanged	Occurs when the value changes.
LowerValueChanged	Occurs when the lower value of the range changes.
UpperValueChanged	Occurs when the upper value of the range changes.
DragStarted	Occurs when a drag action is started on the lower or upper thumb.
LowerDragStarted	Occurs when a drag action is started with the lower thumb.
UpperDragStarted	Occurs when a drag action is started with the upper thumb.

Event	Description
DragCompleted	Occurs when a drag action is completed on the lower or upper thumb.
LowerDragCompleted	Occurs when a drag action is completed on the lower thumb.
UpperDragCompleted	Occurs when a drag action is completed on the upper thumb.

In summary, the biggest benefits of using a **RangeSlider** are the high level of customization, and that it allows for selecting a range using two thumbs instead of selecting a single value as with the regular **Slider**.

Displaying toast notifications and snackbars

The Xamarin Community Toolkit offers great shortcuts to display toast notifications and snackbars quickly. From a layout perspective, they are similar because they are both represented by a box with some information inside, which typically auto-dismisses after a certain amount of time. However, toast notifications are normally used to display system messages. Snackbars typically show a message related to the latest operation done in the app, and they should contain a single-line message and no icon.

With regard to this, the Xamarin Community Toolkit implements two extension methods that extend the **VisualElement** class: **DisplayToastAsync** and **DisplaySnackBarAsync**, both demonstrated in the Pages\Views\SnackBarPage.xaml file. The sample page has four buttons, each opening a snackbar or toast notification. I will start by discussing toast notifications. They are implemented via the **Clicked** event handler for the **DisplayToastClicked** button as follows.

```
async void DisplayToastClicked(object? sender, EventArgs args)
{
      await this.DisplayToastAsync(GenerateLongText(5));
      StatusText.Text = "Toast is closed by timeout";

}
```

You can display a toast notification by invoking the **DisplayToastAsync** method on the current page instance. The method takes the string to display; in this case, an intentionally long string is generated by the **GenerateLongText** method. The toast notification disappears after 3,000 milliseconds by default, but you can provide a different timeout by invoking the second overload of **DisplayToastAsync**, whose second argument is the new timeout expressed in milliseconds. The previous code produces the result shown in Figure 16.

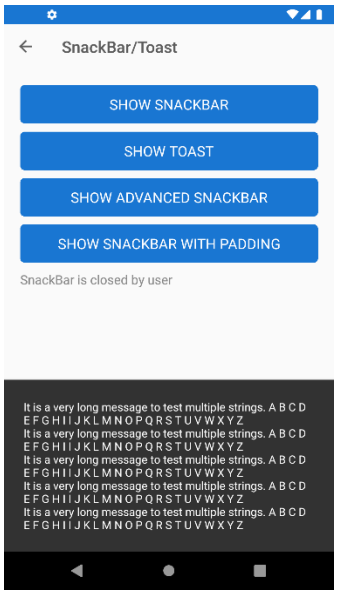


Figure 16: Displaying a toast notification

As you can see, the toast notification does not allow for performing an additional action, which a snackbar does. You can display snackbars by invoking the **`DisplaySnackBarAsync`** method, which optionally takes an argument of type **`SnackBarOptions`**. In its simplest form, you pass the string you want to display and an action, as demonstrated in the **`DisplaySnackBarClicked`** button event handler.

```
var result = await this.DisplaySnackBarAsync(GenerateLongText(5),
             "Run action", () =>
{
      Debug.WriteLine("SnackBar action button clicked");
      return Task.CompletedTask;

});
```

The method provides several overloads; the one considered here shows the string passed as the first argument and allows for executing the **`Func<Task>`** action object passed as the second argument. Figure 17 shows what this looks like.

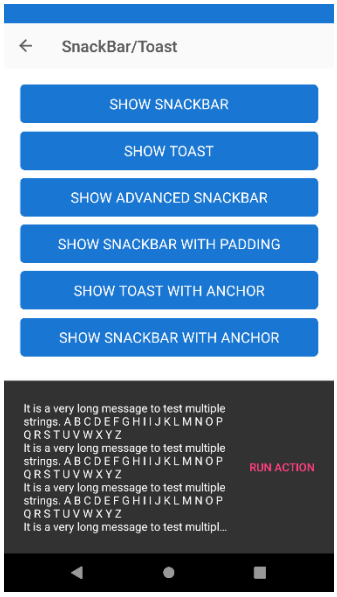


Figure 17: Displaying a snackbar

Snackbars can be customized through an instance of the **SnackBarOptions** object. If you look at the C# code for the **DisplaySnackBarAdvancedClicked** button event handler, it is easier to understand how to create advanced snackbars. In the first part, the code customizes the way the text message is displayed.

```
const string SmileIcon = "\uf118";
var options = new SnackBarOptions
{
      MessageOptions = new MessageOptions
      {
            Foreground = Color.DeepSkyBlue,
            Font = Font.OfSize("FARegular", 40),
            Padding = new Thickness(10, 20, 30, 40),
            Message = SmileIcon

      },
```

The **SnackBarOptions** has a **MessageOptions** property, of type **MessageOptions**, which allows for formatting the text message with the self-explanatory properties you see in the code. Other options that can be customized are set via the following properties.

```
      Duration = TimeSpan.FromMilliseconds(5000),
      BackgroundColor = Color.Coral,
      IsRtl = CultureInfo.CurrentCulture.TextInfo.IsRightToLeft,
```

Notice how easy it is to set the text options with a right-to-left presentation. The duration is expressed in milliseconds, after which the snackbar will be automatically dismissed. The last property is about actions that can be executed from within the snackbar. It is possible to supply multiple actions, as demonstrated in the following code, which assigns the **Actions** property with an object of type **List<SnackBarActionOptions>**.

```
        Actions = new List<SnackBarActionOptions>
        {
                new SnackBarActionOptions
                {
                        ForegroundColor = Color.Red,
                        BackgroundColor = Color.Green,
                        Font = Font.OfSize("Times New Roman", 15),
                        Padding = new Thickness(10, 20, 30, 40),
                        Text = "Action1",
                        Action = () =>
                        {
                                Debug.WriteLine("1");
                                return Task.CompletedTask;
                        }
                },
                new SnackBarActionOptions
                {
                        ForegroundColor = Color.Green,
                        BackgroundColor = Color.Red,
                        Font = Font.OfSize("Times New Roman", 20),
                        Padding = new Thickness(40, 30, 20, 10),
                        Text = "Action2",
                        Action = () =>
                        {
                                Debug.WriteLine("2");
                                return Task.CompletedTask;
                        }
                }
        }

};
```

You can customize the appearance of each individual action by assigning the **ForegroundColor**, **BackgroundColor**, **Font**, **Padding**, and **Text** properties. The action executed is an object of type **Func<Task>**. The way the method is invoked is the following.

```
var result = await this.DisplaySnackBarAsync(options);
```

Tip: Snackbars and toast notifications can be anchored to a specific view because both the `DisplayToastAsync` and `DisplaySnackBarAsync` methods extend the `VisualElement` class, so you can invoke such methods on every view. The sample app has examples about this scenario, in particular the "Show Snackbar with Anchor" example.

Displaying pop-ups

The Xamarin Community Toolkit exposes the Popup view, which allows for rendering native pop-ups on each supported platform. Pop-ups can also be customized in several ways. The sample project provides many examples, located under the Pages\Views\Popups folder.

> ***Note: The sample project provides many examples on pop-ups, but most of them are built by adding property assignments or views to the pop-up content. For this reason, I will start discussing pop-ups from the simplest examples possible, explaining only the most relevant features.***

The best thing to do is to start from the simplest pop-up, and then walk through customization possibilities. The Simple Popup example looks like Figure 18.

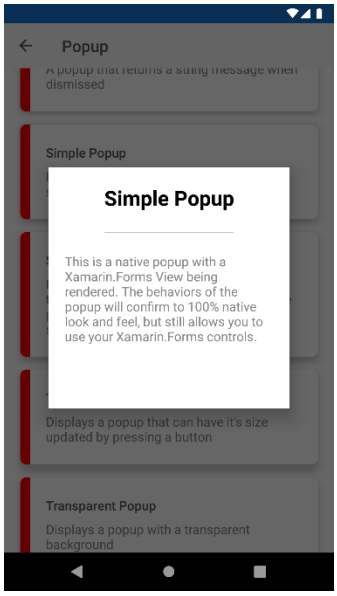


Figure 18: Displaying a simple pop-up

The content of a pop-up is generally a layout, so that you can create a complex visual hierarchy with a title, text, and other views. Code Listing 3 shows the XAML markup for the simple pop-up shown in Figure 18.

Code Listing 3

```
<?xml version="1.0" encoding="utf-8" ?>
<xct:Popup xmlns="http://xamarin.com/schemas/2014/forms"
           xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
           xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
           xmlns:local="clr-namespace:Xamarin.CommunityToolkit.Sample.
                        Pages.Views.Popups"
           Size="{x:Static local:PopupSize.Small}"
           x:Class="Xamarin.CommunityToolkit.Sample.Pages.
                    Views.Popups.SimplePopup">
```

```
    <xct:Popup.Resources>
        <ResourceDictionary>
            <Style x:Key="Title" TargetType="Label">
                <Setter Property="FontSize" Value="26" />
                <Setter Property="FontAttributes" Value="Bold" />
                <Setter Property="TextColor" Value="#000" />
                <Setter Property="VerticalTextAlignment"
                        Value="Center" />
                <Setter Property="HorizontalTextAlignment"
                        Value="Center" />
            </Style>
            <Style x:Key="Divider" TargetType="BoxView">
                <Setter Property="HeightRequest" Value="1" />
                <Setter Property="Margin" Value="50, 25" />
                <Setter Property="Color" Value="#c3c3c3" />
            </Style>
            <Style x:Key="Content" TargetType="Label">
                <Setter Property="HorizontalTextAlignment"
                        Value="Start" />
                <Setter Property="VerticalTextAlignment"
                        Value="Center" />
            </Style>
            <Style x:Key="PopupLayout" TargetType="StackLayout">
                <Setter Property="Padding"
                        Value="{OnPlatform Android=20, UWP=20, iOS=5}" />
            </Style>
        </ResourceDictionary>
    </xct:Popup.Resources>

    <StackLayout Style="{StaticResource PopupLayout}">
        <Label Style="{StaticResource Title}"
               Text="Simple Popup" />
        <BoxView Style="{StaticResource Divider}" />
        <Label Style="{StaticResource Content}"
               Text="This is a native popup with a Xamarin.Forms View
                     inside. The popup will confirm to 100% native
                     UI and use your Xamarin.Forms controls." />
    </StackLayout>
</xct:Popup>
```

Specific Cast

A pop-up is declared inside an individual XAML file and is exposed via the **Popup** class. You can control the pop-up size with the **Size** property, of type **Xamarin.Forms.Size**. In Code Listing 3, you can see that such a property is bound to an enumeration called **PopupSize**, which is not part of the library, rather it is part of the sample code, so you can walk through it as an example.

The **StackLayout** actually r…which means you can show …atever you need. The styl… are applied to the child views, …the result you get is shov… pop-ups, you need to invoke a …xtension method called …s the **Navigation** class and takes …up instance as an arg…od is demonstrated in the …trolViewModel.cs fil…\Views, and works like this:

```
…ation.ShowPopup
```

…**opupAsync** can… if the pop-up is enabled to do so. If you look at …pup.xaml file, …will see the following assignment in the pop-up

```
…String"
```

With this assignment, the method will be able to return an object of type **string**, but you can choose a different type to return. Whatever return type you decide on, the syntax for the method invocation becomes the following.

```
var result = await Navigation.ShowPopupAsync(popup);
```

By default, pop-ups can be dismissed by tapping outside of them (behavior known as *light dismiss*), but sometimes you want to avoid this to make sure users tap on a button and perform a mandatory action. In this case, you can set the **IsLightDismissEnabled** property as follows.

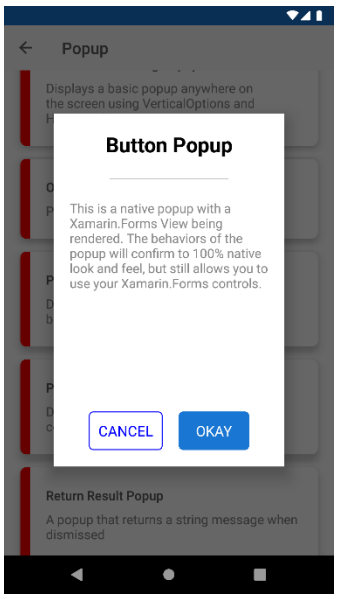
```
IsLightDismissEnabled="False"
```

You can implement actions inside pop-ups. This can be quickly accomplished by adding **Button** views to the content of the pop-up. Two examples, called **Popup with 1 Button** and **Popup with Multiple Buttons**, demonstrate this. The following XAML from the second example shows how to implement multiple buttons.

```
<StackLayout Style="{StaticResource PopupLayout}">
    <Label Style="{StaticResource Title}"
           Text="Button Popup" />
    <BoxView Style="{StaticResource Divider}" />
    <Label Style="{StaticResource Content}"
           Text="This is a native popup with a Xamarin.Forms View being
rendered. The behaviors of the popup will confirm to 100% native look and
feel, but still allows you to use your Xamarin.Forms controls." />
    <StackLayout Style="{StaticResource ButtonGroup}">
        <Button Text="Cancel"
                Style="{StaticResource CancelButton}"
                Clicked="Cancel_Clicked" />
        <Button Text="OKAY"
                Clicked="Okay_Clicked" />
    </StackLayout>
</StackLayout>
```

You can simply handle the **Clicked** event… bind butt…s to commands to execute actions. Figure 19 shows how the pop-up with buttons from the sample project appears.

FreeText



igure 19: Implementing actions inside pop-ups

n of pop-ups by simply assigning the **`HorizontalOptions`** and
of type **`LayoutOptions`**. This allows you to place pop-ups in
nter of the screen (the default). In the
n find several examples. For instance, the following
snippet shows h t the top-right corner.

```
popup.VerticalOptions = new LayoutOptions(LayoutAlignment.Start, true);
popup.HorizontalOptions = new LayoutOptions(LayoutAlignment.End, true);
```

The resulting pop-up appears like in Figure 20.

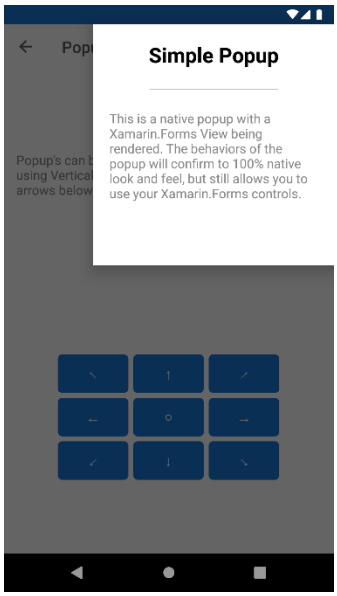


Figure 20: Selecting the pop-up position on screen

Stamp annotation

In summary, native pop-ups can now be displayed very quickly and without the need for building custom views. In addition, customization options are very easy to implement since most of the customizations are applied via property assignments, or by adding views to the pop-up content.

Organizing the UI with tabs

The Xamarin Community Toolkit makes it easy to organize the user interface within tabs via the **`TabView`** control. This control provides deep customization options and offers shortcuts to implement layouts that satisfy the most modern mobile design very quickly.

Due to the high number of possibilities and customization options, the official sample project includes several pages to demonstrate the usage of the **`TabView`**, located under the Pages\Views\TabView folder. I will consider the examples that highlight the most relevant properties and characteristics of the view, starting from the simplest example called **Getting Started** that is coded in the GettingStartedPage.xaml file. In the sample app running, this example looks like the one shown in Figure 21.

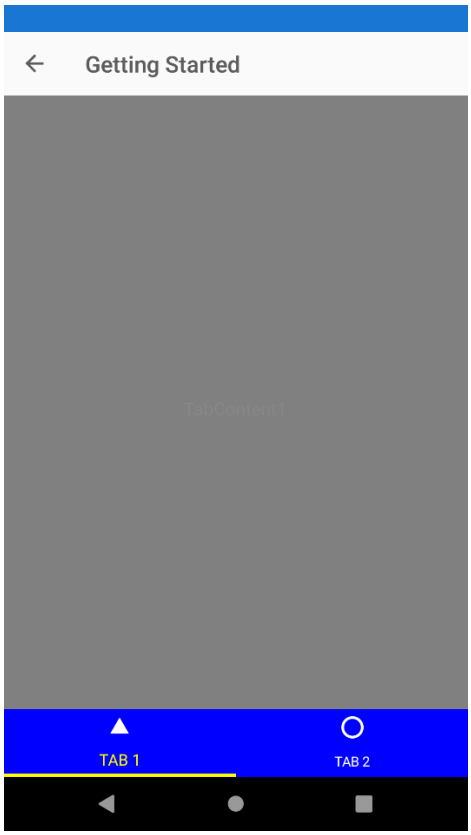


Figure 21: Simple layout with tabs

As you can see, a tab strip contains two tab definitions, each with its own text and icon. Colors can be defined in code. The opening tag for the **`TabView`** is the following.

```
<xct:TabView
    TabStripPlacement="Bottom"
```

```
    TabStripBackgroundColor="Blue"
    TabStripHeight="60"
    TabIndicatorColor="Yellow"
    TabContentBackgroundColor="Yellow">
```

The **TabStripPlacement** property allows you to set the position of the tab strip, and its value can be **Bottom** or **Top**. The **TabStripBackgroundColor** property, of type **Color**, allows you to specify a background color for the tab strip, whereas the **TabStripHeight** allows you to specify the tab strip height.

The **TabIndicatorColor** property, of type **Color**, is assigned with the color you want to use to highlight the currently selected tab, whereas the **TabContentBackgroundColor** property allows for specifying the background color of the tab content, which is something you can easily understand by selecting the second tab in the sample page.

The **TabView** works as a container for **TabViewItem** objects, each representing a tab. For example, the first tab in the sample page is defined as follows.

```
<xct:TabViewItem
    Icon="triangle.png" Text="Tab 1" TextColor="White"
    TextColorSelected="Yellow" FontSize="12">
    <Grid BackgroundColor="Gray">
        <Label HorizontalOptions="Center" VerticalOptions="Center"
            Text="TabContent1" />
    </Grid>
</xct:TabViewItem>
```

These are the relevant points:

- The tab icon is assigned via the **Icon** property and is of type **ImageSource**, so you can assign icons and images as you would do with any image view.
- The **TextColor** and **TextColorSelected** properties, both of type **Color**, respectively allow for assigning the color for the text in normal state and in selected state.
- The **TabViewItem** has a default, implicit **Content** property that contains the layout or view that you want to display when the tab is selected, in this case a simple **Grid** with a **Label** inside.

The visibility of the tab strip can be managed via the **IsTabStripVisible** property. You can use this property to hide and show the tab strip at runtime based on your conditions. There is basically no limit to the number of **TabViewItem** objects you can add, because the **TabView** has built-in support for scrolling. If you open the **Scrollable Tabs** example, you can see nine tabs by scrolling the tab strip horizontally without specifying any additional property.

An interesting feature of tabs is that they support badges. This is demonstrated in the **Tab Badge** example (TabBadgePage.xaml file), as you can see in Figure 22.

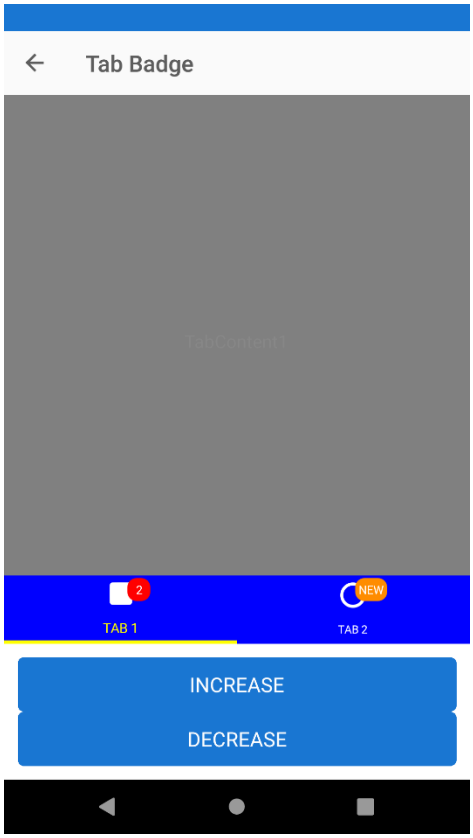


Figure 22: Display badges on tabs

Creating badges is very easy. Consider the following code snippet, related to the second tab on the strip.

```
<xct:TabViewItem
    Icon="circle.png"
    IconSelected="square"
    Text="Tab 2"
    TextColor="White"
    TextColorSelected="Yellow"
    BadgeText="NEW"
    BadgeBackgroundColor="DarkOrange"
    BadgeBackgroundColorSelected="Red"
    BadgeBorderColor="Green"
    BadgeBorderColorSelected="LightGreen"
    BadgeTextColor="White"
    FontSize="10"
    FontAttributesSelected="12">
```

Names of the properties related to badges start with the **Badge** literal and are self-explanatory. The **BadgeText** property can be used to add text to the badge. The **BadgeBackgroundColor** and **BadgeBackgroundColorSelected** properties, both of type **Color**, respectively allow for setting the background color for the badge when the tab is not selected and when it is selected.

Similarly, you can set colors for the badge border via the **BadgeBorderColor** and **BadgeBorderColorSelected** properties. Finally, you can specify the color for the text in the badge via the **BadgeTextColor** property. In this example, some changes are also applied to the default font of the text in the **TabView**.

The font can be controlled via the following properties:

- **FontFamily** and **FontFamilySelected** allow for setting the font type to both the normal and selected states.
- **FontAttributes** and **FontAttributesSelected** allow for setting the font attributes to both the normal and selected states. If you look at the previous code snippet, the value for **FontAttributesSelected** is **12**. Actually, this is the combination of the 1 and 2 values of the **FontAttributes** enumeration (**Bold** and **Italic**).
- **FontSize** and **FontSizeSelected** allow for specifying the font size depending on the state of the tab.

Before going into customizing tabs, it can be useful to summarize a few more properties:

- The **TabWidth** property, of type **double**, allows you to change the width of each individual tab.
- Because the **TabView** can contain any view, it can also contain another **TabView** object to give you the option to nest tab views and create complex visual hierarchies.
- You can apply animations to the transition between one state and another of the tabs. This implies that the **IsTabTransitionsEnabled** property is **true**. Animations are represented by classes that implement the **ITabViewItemAnimation** interface and are demonstrated in the TabItemAnimationPage.xaml file and in its code-behind C# file.

Full tab customization

Both the **TabView** and **TabViewItem** objects can be completely customized. The sample app has an example called **Customizing Tabs**, whose result is shown in Figure 23.

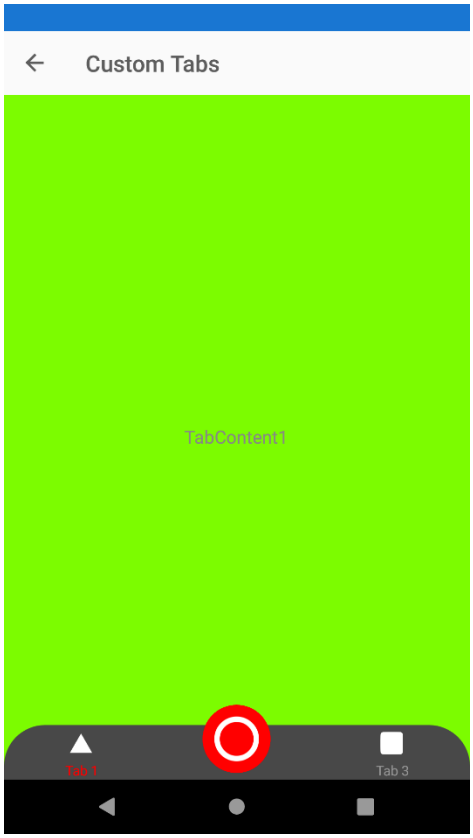


Figure 23: Customizing tabs

As you can see, the tab strip has a completely different look with rounded corners, and the center tab item has a completely different appearance. Let's look into this by analyzing small pieces of code. The **TabView** definition is using a **BoxView** with rounded corners as the background.

```
<xct:TabView
    Style="{StaticResource CustomTabStyle}">
    <xct:TabView.TabStripBackgroundView>
        <BoxView
            Color="#484848"
            CornerRadius="36, 36, 0, 0"/>

    </xct:TabView.TabStripBackgroundView>
```

This means that the background for the tab strip can not only be a color, but also a view to be set via the **TabStripBackgroundView** property. The style applied to the **TabView** is very simple.

```
<Style
    x:Key="CustomTabStyle" TargetType="xct:TabView">
    <Setter
        Property="IsTabTransitionEnabled" Value="True" />
    <Setter
        Property="TabStripHeight" Value="48" />
```

```
    <Setter
        Property="TabContentBackgroundColor" Value="#484848" />
    <Setter
        Property="TabStripPlacement" Value="Bottom" />

</Style>
```

All three **TabViewItem** objects redefine their base layout by assigning the **ControlTemplate** property with objects of type **ControlTemplate**, defined in the same page. While for the first and third tab, the control templates are quite easy to understand, the control template for the second tab is more complex (and interesting), so I will focus on this, which is defined as follows.

```
<xct:TabViewItem Text="Tab 2" Icon="circle.png"
    ControlTemplate="{StaticResource FabTabItemTemplate}"
    Style="{StaticResource TabItemStyle}"

    TabTapped="OnFabTabTapped" />
```

The first thing to notice is that **TabViewItem** objects support interaction via the **TabTapped** event. The sample project displays a simple alert on tap.

```
void OnFabTabTapped(object? sender, TabTappedEventArgs e) =>
     DisplayAlert("FabTabGallery", "Tab Tapped.", "Ok");
```

The **TabTappedEventArgs** object exposes the **Position** property (0-based), of type **int**, which allows you to understand the position of the tapped tab inside the strip. The redefined control template for the current tab is shown in Code Listing 4.

Code Listing 4

```
<ControlTemplate
    x:Key="FabTabItemTemplate">
    <Grid>
        <ImageButton
            InputTransparent="True"
            Source="{TemplateBinding CurrentIcon}"
            Padding="10"
            HorizontalOptions="Center"
            BackgroundColor="#FF0000"
            HeightRequest="60"
            WidthRequest="60"
            Margin="6">
            <ImageButton.CornerRadius>
                <OnPlatform x:TypeArguments="x:Int32">
                    <On Platform="iOS" Value="30"/>
                    <On Platform="Android" Value="60"/>
                    <On Platform="UWP" Value="36"/>
                </OnPlatform>
            </ImageButton.CornerRadius>
            <ImageButton.IsVisible>
```

```
                <OnPlatform x:TypeArguments="x:Boolean">
                    <On Platform="Android, iOS, UWP">True</On>
                    <On Platform="macOS">False</On>
                </OnPlatform>
            </ImageButton.IsVisible>
        </ImageButton>
        <BoxView
            InputTransparent="True"
            HorizontalOptions="Center"
            CornerRadius="30"
            BackgroundColor="#FF0000"
            HeightRequest="60"
            WidthRequest="60"
            Margin="6">
            <BoxView.IsVisible>
                <OnPlatform x:TypeArguments="x:Boolean">
                    <On Platform="Android, iOS, UWP">False</On>
                    <On Platform="macOS">True</On>
                </OnPlatform>
            </BoxView.IsVisible>
        </BoxView>
    </Grid>
</ControlTemplate>
```

At the core of the template there is an **`ImageButton`** view, which is the main view in the default template. Notice how the **`Source`** property assigns the default icon via the **`TemplateBinding`** extension. All the other properties are related to the position, colors, and size. For the corner radius and for its visibility, the code makes different decisions depending on the platform the app is running on. A **`BoxView`** is used to display the red circle in the template. If you now look back at Figure 23, it should be clearer how powerful customization options are, especially if you have knowledge of how control templates work in Xamarin.Forms (and more generally, in development platforms based on XAML).

Note: You have learned all the most important things about implementing and customizing tabs and the tab strip. It will now be easier to look at the remaining sample pages, which leverage the same concepts.

Improving accessibility with SemanticOrderView and SemanticEffect

Accessibility is a feature offered by the operating systems, and on most devices, and makes it possible for visually impaired people to use a phone, tablet, and PC more easily because the device will literally speak, describing the content of each individual visual element in the system UI or inside an app.

Xamarin.Forms has support for accessibility, but the Xamarin Community Toolkit goes a step further by providing the **SemanticOrderView** control and the **SemanticEffect** effect. These are both discussed here because they belong to the same area. In the sample project, the **SemanticOrderView** is represented in the Pages\Views\SemanticOrderViewPage.xaml file.

Tip: You will need to turn on accessibility on your device; otherwise, you will not be able to hear the text-to-speech engine working in the sample app.

It is declared as follows.

```
<xct:SemanticOrderView x:Name="acv">
    <StackLayout>
        <Label x:Name="second" Text="Second" Margin="0,20" />
        <Button x:Name="third" Text="Third" Margin="0,20" />
        <Label x:Name="fourth" Text="Fourth" Margin="0,20" />
        <Button x:Name="fifth" Text="Fifth and last" Margin="0,20" />
        <Button x:Name="first" Text="First" Margin="0,20" />
    </StackLayout>

</xct:SemanticOrderView>
```

It basically simplifies the way the user can understand the order of visual elements in a list. The **SemanticEffect** object makes some steps forward, helping users to identify the type of content in the user interface. If you look into the Pages\Effects\SemanticEffectPage.xaml file, you will see several code snippets of interest. First, a series of **Label** views declared as follows.

```
<Label Text="I have no heading" xct:SemanticEffect.HeadingLevel="None"/>
<Label Text="I am a heading 1" xct:SemanticEffect.HeadingLevel="Level1"/>
<Label Text="I am a heading 2" xct:SemanticEffect.HeadingLevel="Level2"/>
<Label Text="I am a heading 3" xct:SemanticEffect.HeadingLevel="Level3"/>
<Label Text="I am a heading 4" xct:SemanticEffect.HeadingLevel="Level4"/>
<Label Text="I am a heading 5" xct:SemanticEffect.HeadingLevel="Level5"/>
<Label Text="I am a heading 6" xct:SemanticEffect.HeadingLevel="Level6"/>
<Label Text="I am a heading 7" xct:SemanticEffect.HeadingLevel="Level7"/>
<Label Text="I am a heading 8" xct:SemanticEffect.HeadingLevel="Level8"/>
<Label Text="I am a heading 9" xct:SemanticEffect.HeadingLevel="Level9"/>
```

The **HeadingLevel** property of the **SemanticEffect** allows for specifying the heading level of a string inside a text hierarchy. This property is of type **HeadingLevel**, an enumeration that supports up to nine heading levels. Another property of the effect is **Description** and it works like in the following **Label**.

```
<Label Text="I am a label with an automation ID"
       AutomationId="labelAutomationIdTest"
       xct:SemanticEffect.Description="This is a semantic description" />
```

When this property is assigned, the device will speak, saying the value of the description. The last property of the effect is **Hint**.

```
<Button
```

```
    Text="Button with hint"
    xct:SemanticEffect.Hint="This is a hint that describes the button. For
example, 'sends a message'"/>
```

The **`Hint`** property contains a suggestion or explanation on what a view does. Figure 24 shows what the sample page looks like, but it obviously cannot represent the spoken phrases.

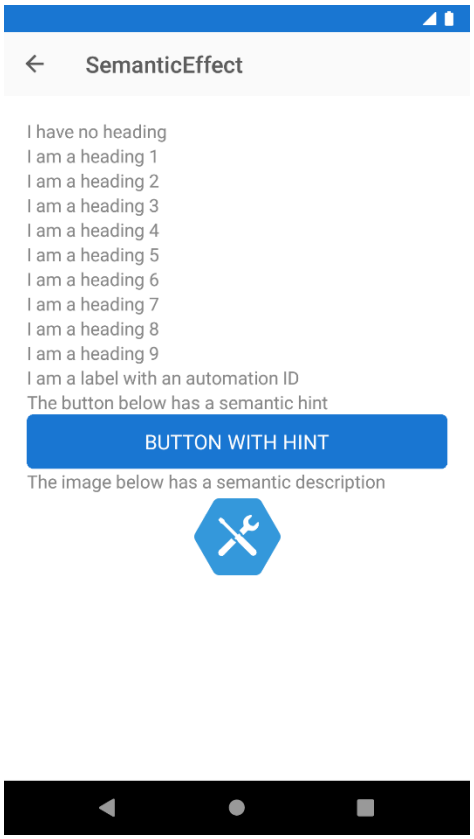


*Figure 24: Improving accessibility with the **`SemanticEffect`***

Microsoft has also published a video about using these new elements, where they are properly demonstrated with live actions and audio, so I recommend you watch it for a better understanding.

Chapter summary

Modern mobile designs often include views that are not part of the Xamarin.Forms codebase, but that are now becoming of common use. The Xamarin Community Toolkit bridges this gap by providing several views of common use, so that developers do not need to use third-party controls or develop such views on their own, thus saving time, eliminating external code dependencies, and improving productivity.

However, new views are only a part of modern designs. In fact, these also depend on device features and screen form factors. The next chapter describes how the Xamarin Community Toolkit also simplifies the way you can manage the user interface, especially on specific (and popular) devices.

Chapter 3 Improved UI Management with Effects

In Xamarin.Forms, Effects allows you to customize views with small styling changes, without the need to implement more complex, custom renderers. The Xamarin Community Toolkit offers seven effects that address very common needs. This chapter walks through all of them, making the necessary considerations from the technical point of view.

As usual, the sample app from the official repository will be used for quicker reference. If you want to use the code in a different page, remember to import the following XML namespace in the XAML declaration of the page.

```
xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
```

Setting icons' tint color with IconTintColorEffect

The **`IconTintColorEffect`** is demonstrated in the Effects\IconTintColorEffectPage.xaml file, and it allows you to change the tint color of the icons you use quickly with **`Image`** and **`ImageButton`** views. This effect actually works with monochrome icons. To understand how it works, look at Figure 25.

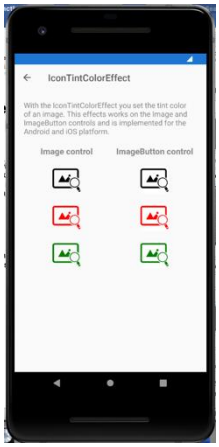


Figure 25: Changing an icon's tint color

On the left side, you can see three **`Image`** views with their **`Source`** property assigned with a monochrome icon (the one in the example is available here), and on the right side you can see three **`ImageButton`** views with their **`Source`** property assigned with the same icon.

The first icon, the black one, is the original one for both views. For image views, the tint color has been quickly changed with the following code.

```
<Image Source="https://image.flaticon.com/icons/png/512/48/48639.png"

    xct:IconTintColorEffect.TintColor="Red" Grid.Row="2" />

<Image Source="https://image.flaticon.com/icons/png/512/48/48639.png"

    xct:IconTintColorEffect.TintColor="Green" Grid.Row="3" />
```

As you can see, you simply need to assign the **TintColor** property of the effect with an object of type **Color**. The way it works for **ImageButton** views is exactly the same, as you can see in the following code snippet.

```
<ImageButton
    Source="https://image.flaticon.com/icons/png/512/48/48639.png"
    xct:IconTintColorEffect.TintColor="Red"
    Grid.Row="2" Grid.Column="1" />

<ImageButton
    Source="https://image.flaticon.com/icons/png/512/48/48639.png"
    xct:IconTintColorEffect.TintColor="Green"
    Grid.Row="3" Grid.Column="1" />
```

This effect can be very useful when you use icons to present the state of an item in your user interface, and you want to change the icon color to represent a state change.

Managing views' lifecycles with LifecycleEffect

The **LifecycleEffect** is demonstrated in the Effects\LifecycleEffectPage.xaml file, and it allows you to understand when an object deriving from **VisualElement** is loaded into or unloaded from the visual tree of the page.

The effect exposes two events, **Loaded** and **Unloaded**, which can be really useful to take actions only when a view is loaded or unloaded. If you look at Code Listing 5, you can see how the effect is applied to both layouts and individual views.

Code Listing 5

```
<?xml version="1.0" encoding="utf-8" ?>
<pages:BasePage

x:Class="Xamarin.CommunityToolkit.Sample.Pages.Effects.LifeCycleEffectPage"
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:pages="clr-namespace:Xamarin.CommunityToolkit.Sample.Pages"
    xmlns:xct="http://xamarin.com/schemas/2020/toolkit">
    <ContentPage.Content>
        <StackLayout x:Name="stack">
```

```
            <StackLayout.Effects>
                <xct:LifecycleEffect Loaded="LifeCycleEffect_Loaded"
Unloaded="LifeCycleEffect_Unloaded" />
            </StackLayout.Effects>
            <Label
                HorizontalOptions="CenterAndExpand"
                Text="When you press the button, the Image will appear and
after 3 seconds will be removed!"
                VerticalOptions="CenterAndExpand">
                <Label.Effects>
                    <xct:LifecycleEffect Loaded="LifeCycleEffect_Loaded"
                        Unloaded="LifeCycleEffect_Unloaded" />
                </Label.Effects>
            </Label>
            <Image
                x:Name="img"
                IsVisible="false"

Source="https://raw.githubusercontent.com/xamarin/XamarinCommunityToolkit/m
ain/assets/XamarinCommunityToolkit_128x128.png">
                <Image.Effects>
                    <xct:LifecycleEffect Loaded="LifeCycleEffect_Loaded"
                        Unloaded="LifeCycleEffect_Unloaded" />
                </Image.Effects>
            </Image>
            <Button Clicked="Button_Clicked"
                    Text="Present Image and Remove it">
                <Button.Effects>
                    <xct:LifecycleEffect Loaded="LifeCycleEffect_Loaded"
                        Unloaded="LifeCycleEffect_Unloaded" />
                </Button.Effects>
            </Button>
            <Label Text="Log:" />
            <Label x:Name="lbl" TextColor="Red" />
        </StackLayout>
    </ContentPage.Content>
</pages:BasePage>
```

The sample application only displays a text message when the **`Loaded`** or **`Unloaded`** events are raised, but these are the places where you would take your own custom actions. As demonstrated in the code-behind file, you can use the same event handler for multiple views. Code Listing 6 demonstrates this.

Code Listing 6

```
            void LifeCycleEffect_Loaded(object? sender, EventArgs e)
            {
                  if (sender is Button)
```

```
                lbl.Text += "Button loaded \n";
            if (sender is Image)
                lbl.Text += "Image loaded \n";
            if (sender is Label)
                lbl.Text += "Label loaded \n";
            if (sender is StackLayout)
                lbl.Text += "StackLayout loaded \n";
        }

        void LifeCycleEffect_Unloaded(object? sender, EventArgs e)
        {
            if (sender is Button)
                lbl.Text += "Button unloaded \n";
            if (sender is Image)
                lbl.Text += "Image unloaded \n";
            if (sender is Label)
                lbl.Text += "Label unloaded \n";
            if (sender is StackLayout)
                lbl.Text += "StackLayout unloaded \n";
        }
```

With the **is** operator, you can quickly check the type of the object that has raised the events and make the appropriate decisions.

Managing an entry border with EntryBorderEffect

Like most views, the **Entry** view has a default behavior on iOS and Android. More specifically, on iOS the **Entry** is surrounded by a border, while on Android it displays an underline. However, especially if you build cross-platform apps with the same design, a graphic designer might ask you to make the look and feel consistent between platforms by removing both the border on iOS and the underline on Android.

Before the Xamarin Community Toolkit was available, you had to create your own custom view and implement a custom renderer or effect. Now you can get this done by simply applying the **EntryBorderEffect** to the **Entry** views you want to make consistent.

In the sample app, this is demonstrated in the Effects\EntryBorderEffectsPage.xaml file. You just need to add the following simple lines to the **Entry** definition.

```
<Entry.Effects>
    <xct:RemoveBorderEffect/>
</Entry.Effects>
```

Figure 26 shows how the **Entry** views appear without and with the effect applied, on both iOS and Android.

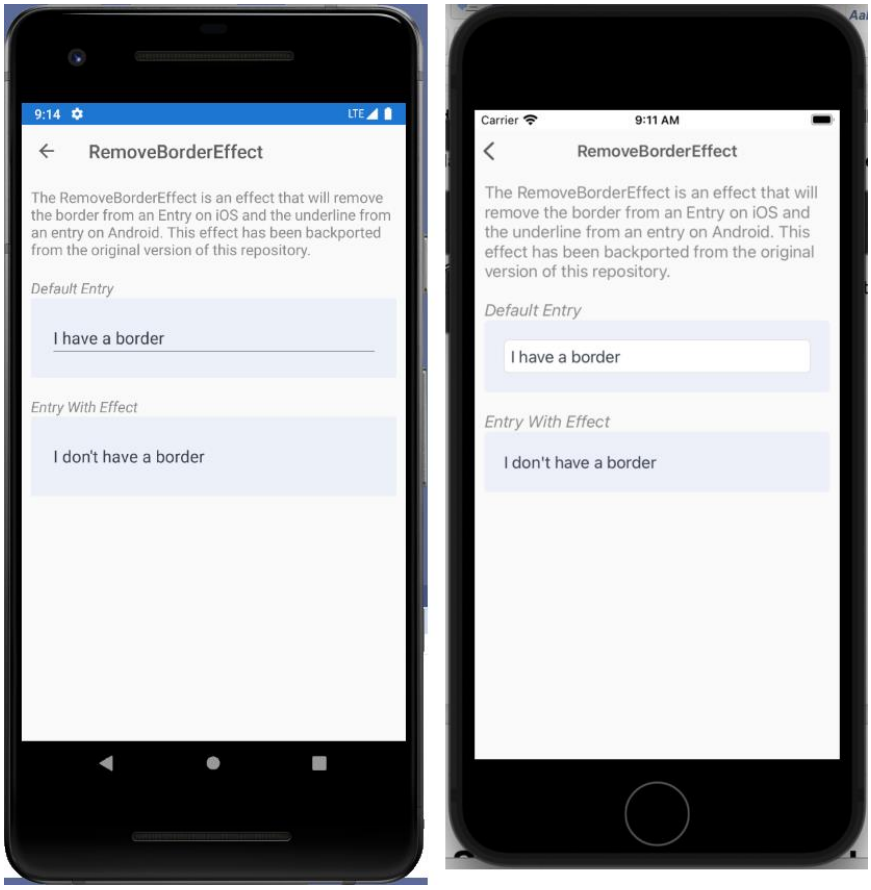


*Figure 26: Applying the **`RemoveBorderEffect`***

This is a really cool effect that will save a lot of time with cross-platform design systems.

iOS: Managing the safe area with SafeAreaEffect

Note: The `SafeAreaEffect` does not apply to Android and is simply ignored at runtime.

The most recent iPhones have larger screens, and consequently, a larger area that app developers can leverage to build even more beautiful user interfaces. The Apple layout guidelines divide the screen area on the new iPhones into two parts. The first part is the full screen area as a whole. The second part is called the safe area, and it is made of a rectangle that represents the same screen area available on older iPhones.

Figure 27, which is credited to Apple and taken from the Adaptivity and Layout documentation page, provides a clean representation of how the screen area is organized.

Figure 27: The safe area on iPhone devices (source: Apple, Inc.)

When you work with Xamarin.iOS and Xamarin.Forms, your app will automatically fill the entire screen area. However, there might be exceptions and reasons to avoid filling the entire screen and use only the safe area, such as:

- You want to really make sure your app looks and runs properly on older iPhones.
- You build for both Android and iOS, and you don't want to write additional, platform-specific code for your UI.
- Your app adheres to a design that is thought to be cross-platform, or that is built upon older iPhone devices and that cannot be changed.

Because by default your app will fill the entire screen when running on iOS, if you want it to run only inside the safe area, you have to take the responsibility for handling this scenario in code. Luckily, with Xamarin Community Toolkit, it's very quick and simple to make your apps use the safe area only via the **`SafeAreaEffect`**.

This is demonstrated in the Effects\SafeAreaEffectPage.xaml file and can be applied in both XAML and C#. Actually, the official sample app provides an example based on C# only, but I will show both ways starting from XAML. You apply the **`SafeAreaEffect`** to the desired view as follows.

```
<StackLayout SafeAreaEffect.SafeArea="true">

</StackLayout>
```

So, you simply need to assign the **`SafeArea`** property of the effect with **`true`**.

Tip: This example is applied to a `StackLayout` for consistency with the example offered by the sample app. However, in the real world, you might need to apply the effect at the page level for a more consistent layout of the user interface, especially with different OSes.

As an alternative, you can apply the effect in code-behind with the following code (which requires a **`using Xamarin.CommunityToolkit.Effects;`** directive).

```
SafeAreaEffect.SetSafeArea(view, value);
```

`view` is the view to which the effect is applied, and the value is **`true`** (safe area is on) or **`false`** (safe area is off). Actually, the second argument of **`SetSafeArea`** is not a **`bool`** object; it is a structure of type **`SafeArea`** from the **`Helpers`** class. With this structure, you can decide which sides of the safe area can be enabled with the following constructor overloads.

```
// Uniform Safe Area.
var safeArea = new Helpers.SafeArea(true);
// Horizontal and vertical Safe Area.
var safeArea = new Helpers.SafeArea(true, true);
// Left, top, right, bottom.
var safeArea = new Helpers.SafeArea(true, true, true, true);
```

You can replace the **`true`** arguments with **`false`** where you do not want the safe area to be enabled, and then you can pass the **`safeArea`** structure instance to the **`SetSafeArea`** method as follows.

```
SafeAreaEffect.SetSafeArea(view, safeArea);
```

In the official code example, the safe area is enabled programmatically with a **`Switch`** view, which raises a **`Toggled`** event when the state changes, and that is handled as follows.

```
void ActivationToggle_Toggled(object? sender, Forms.ToggledEventArgs e) =>

      SafeAreaEffect.SetSafeArea(stack, e.Value);
```

Figure 28 shows how the safe area looks with the official sample, on an iPhone 12 simulator.

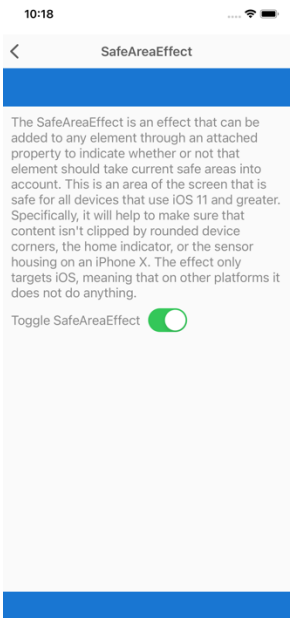


Figure 28: The safe area handled on an iPhone 12 simulator

The blue area is the part of the screen that is available to this specific kind of device, whereas the content is located inside the safe area.

Preselecting text with SelectAllTextEffect

Sometimes, you might want to allow preselection of all the text inside an **`Entry`** or **`Editor`** view. There are several reasons for doing this, such as making it easier to copy the text to the clipboard or deleting the content of a prefilled form.

This is another problem that the Xamarin Community Toolkit solves in a clever way, since you previously had to create a custom renderer or your own effect. Automatic text preselection happens by simply applying the **`SelectAllTextEffect`** to an **`Entry`** or **`Editor`**, and this is demonstrated in the Effects\SelectAllTextEffectsPage.xaml file of the sample project.

In XAML, the sample code looks like this:

```
<Entry Text="https://github.com/xamarin/XamarinCommunityToolkit"
       TextColor="{StaticResource DarkLabelTextColor}"
       PlaceholderColor="{StaticResource DarkLabelPlaceholderColor}">
    <Entry.Effects>
        <xct:SelectAllTextEffect/>
    </Entry.Effects>
</Entry>
```

As you can see, it's very quick and easy. Similarly, the effect can be applied to an **`Editor`** view as follows.

```
<Editor TextColor="{StaticResource DarkLabelTextColor}"
```

```
        PlaceholderColor="{StaticResource DarkLabelPlaceholderColor}">
    <Editor.Effects>
        <xct:SelectAllTextEffect/>
    </Editor.Effects>
</Editor>
```

If you run the sample code, you will see that the page displays an **`Entry`** and an **`Editor`** without the effect, plus an **`Entry`** and an **`Editor`** with the effect applied. Figure 29 shows how all the text in the **`Entry`** is automatically selected when focused on.

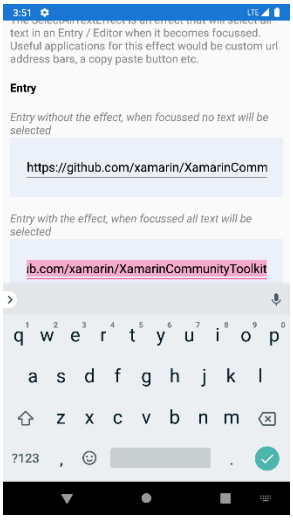


*Figure 29: Automatic text preselection with the **`SelectAllTextEffect`** effect*

You will get the same result with the **`Editor`** at the bottom of the page.

Adding shadows with ShadowEffect

Adding some shadow to views is not just something nice from an aesthetic point of view—it can really improve the experience of the user who needs to interact with visual elements on the page.

In Xamarin.Forms, only a very few visual elements support shadowing, and with limited customization options, so in most cases you have to create your own custom renderers. The **`ShadowEffect`** offered by the Xamarin Community Toolkit allows you to apply shadows to basically any view, and in the sample project it is demonstrated in the Effects\ShadowEffectPage.xaml file.

For a better understanding, let's start by looking at the page resulting from the sample code, shown in Figure 30.

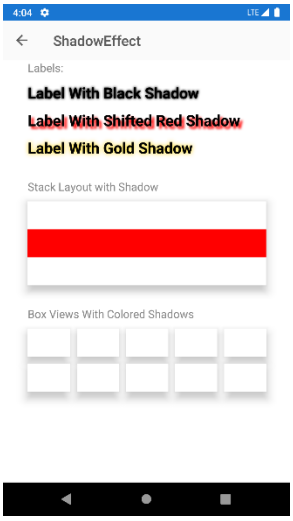


Figure 30: Examples of shadows

With this reference in mind, let's now discuss the usage of the **`ShadowEffect`**. In its simplest form, you apply it to a view by specifying the shadow color as follows.

```
<Label Text="Label With Black Shadow" Style="{StaticResource ShadowLabel}"
       xct:ShadowEffect.Color="Black"/>
```

The biggest benefit of this effect is that it allows you to specify the position of the shadow, with the **`OffsetX`** and **`OffsetY`** properties, both of type **`double`**, as follows.

```
<Label Text="Label With Shifted Red Shadow"
       Style="{StaticResource ShadowLabel}" xct:ShadowEffect.Color="Red"
       xct:ShadowEffect.OffsetX="10" xct:ShadowEffect.OffsetY="10" />
```

More specifically, **`OffsetX`** specifies the horizontal distance of the shadow from the view, and negative values place shadows to the left of the view. **`OffsetY`** specifies the vertical distance from the view, and negative values place the shadow at the top of the element. Another property of the effect is **`Radius`**, which you set like in the following example.

```
<Label Text="Label With Gold Shadow" Style="{StaticResource ShadowLabel}"
       xct:ShadowEffect.Color="Gold" xct:ShadowEffect.Radius="20"/>
```

This property, of type **`double`**, allows you to set how large the shadow is: the higher the value, the greater the blur of the shadow. The **`ShadowEffect`** object also exposes the **`Opacity`** property, which allows, as the name implies, specifying the opacity of the shadow with a value from 0 to 1.

Improving interaction with TouchEffect

The **`TouchEffect`** allows changing the appearance of a view depending on its state, such as normal, pressed, and hovered by, also providing the option to include animations. In the sample project, it is demonstrated with the Effects\TouchEffectPage.xaml file.

The **`TouchEffect`** is certainly the most complex effect in the library, so it's a good idea to discuss its capabilities in separate paragraphs. Before going into the details, you can keep Figure 31 as a reference for the discussion. It is based on the official sample project, and I will recall views displayed in the sample page for a better understanding.

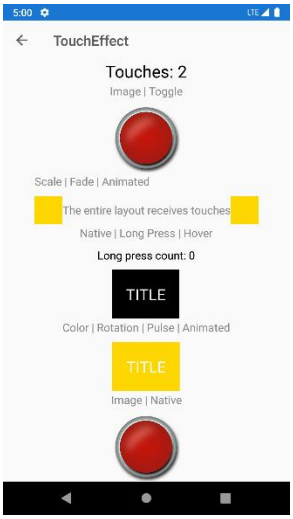


*Figure 31: Examples of **`TouchEffect`** applied to different views*

At the end of this section, all the pieces will be put together for a comprehensive understanding of the effect.

> ***Note: The sample project extends the effect with commanding capabilities. This is not actually part of the `TouchEffect`; rather, it is part of the Model-View-ViewModel support offered by the Toolkit, discussed in Chapter 6.***

Changing a view's appearance

The **`TouchEffect`** supports three different states of a view: normal, pressed, and hovered. For each state, it is possible to change the background color or image. It is also possible to implement an animation when a state changes, which is discussed in the next paragraph.

Table 3 describes the most important properties of the **`TouchEffect`** that allow for managing the three states.

Table 3: Representative TouchEffect properties to manage view states

`NormalBackgroundColor`	Set the background color for the normal state of a view.
`NormalBackgroundImageSource`	Set the background image for the normal state of a view.
`NormalBackgroundImageAspect`	Set the aspect of the image for the normal state of a view. Supported values are

	Fill, **AspectFill**, and **AspectFit** from the **Aspect** enumeration.
PressedBackgroundColor	Set the background color for the pressed state of a view.
PressedBackgroundImageSource	Set the background image for the pressed state of a view.
PressedBackgroundImageAspect	Set the aspect of the image for the pressed state of a view. Supported values are **Fill**, **AspectFill**, and **AspectFit** from the **Aspect** enumeration.
HoveredBackgroundColor	Set the background color for the hovered state of a view.
HoveredBackgroundImageSource	Set the background image for the hovered state of a view.
HoveredBackgroundImageAspect	Set the aspect of the image for the hovered state of a view. Supported values are **Fill**, **AspectFill**, and **AspectFit** from the **Aspect** enumeration.
NormalScale	Set the scale for the normal state of the view. The default is 1.
HoveredScale	Set the scale for the hovered state of the view.
PressedScale	Set the scale for the pressed state of the view.
NormalOpacity	Set the opacity (transparency) for the normal state of the view. The default is 1.
HoveredOpacity	Set the opacity (transparency) for the hovered state of the view. The default is 1.
PressedOpacity	Set the opacity (transparency) for the pressed state of the view. The default is 1.

The following XAML code, taken from the sample project, demonstrates how to set up the normal and pressed states for the first image you see at the top in Figure 31.

```
<Image xct:TouchEffect.NormalBackgroundImageSource="{xct:ImageResource
Id=Xamarin.CommunityToolkit.Sample.Images.button.png}"
   xct:TouchEffect.PressedBackgroundImageSource="{xct:ImageResource
Id=Xamarin.CommunityToolkit.Sample.Images.button_pressed.png}"
```

```
xct:TouchEffect.IsToggled="False"
xct:TouchEffect.Command="{Binding Command, Source={x:Reference Page}}"/>
```

You can additionally control the appearance of the view with the **IsToggled** property. When **true**, the view keeps the pressed state. When **false**, the view returns to the normal state after it has been pressed.

Animating a state change

The **TouchEffect** provides support for animating the view it is applied to and for animating the transition between one state and another. For animations, the **TouchEffect** provides the properties summarized in Table 4 for animating state changes.

Table 4: Animating state changes

AnimationDuration	The duration of the animation in milliseconds.
AnimationEasing	One of the animations exposed by the **Xamarin.Forms.Easing** class.
NormalAnimationDuration	The duration of the animation of the normal state in milliseconds.
NormalAnimationEasing	One of the animations exposed by the **Xamarin.Forms.Easing** class.
HoveredAnimationDuration	The duration of the animation of the hovered state in milliseconds.
HoveredAnimationEasing	One of the animations exposed by the **Xamarin.Forms.Easing** class.
PressedAnimationDuration	The duration of the animation of the pressed state in milliseconds.
PressedAnimationEasing	One of the animations exposed by the **Xamarin.Forms.Easing** class.
NativeAnimation	In conjunction with **AnimationDuration** and **AnimationEasing**, allows for running native animations instead of cross-platform ones.
NativeAnimationColor	Set the color of the native animation
NativeAnimationRadius	Set the radius of the native animation

NativeAnimationShadowRadius	Set the radius of the shadow for the native animation.

The XAML syntax for assigning a specific animation type to the properties that support the **Easing** class is the following.

```
xct:TouchEffect.AnimationEasing="{x:Static Easing.CubicInOut}"
```

The following XAML code demonstrates how the sample project animates the **TITLE** label, the one with the dark yellow background you can see in Figure 31.

```
<StackLayout Style="{StaticResource GridRowContentStyle}">

    <Label Text="Color | Rotation | Pulse | Animated" />

    <StackLayout Orientation="Horizontal"
             HorizontalOptions="CenterAndExpand"
             Padding="20"
             xct:TouchEffect.AnimationDuration="500"
             xct:TouchEffect.PulseCount="2"
             xct:TouchEffect.NormalBackgroundColor="Gold"
             xct:TouchEffect.PressedBackgroundColor="Orange"
             xct:TouchEffect.PressedRotation="15"
             xct:TouchEffect.Command="{Binding Command,
                  Source={x:Reference Page}}">
        <Label Text="TITLE"
           TextColor="White"
           FontSize="Large"/>
    </StackLayout>
</StackLayout>
```

There is actually more, because a view can be rotated and translated during an animation. Table 5 describes the properties that allow you to accomplish this.

Table 5: Animating state changes

NormalRotation	Apply a rotation of the specified degrees when the view is in normal state.
NormalRotationX	Apply a rotation starting from the specified point on the x-axis when the view is in normal state.
NormalRotationY	Apply a rotation starting from the specified point on the y-axis when the view is in normal state.
HoveredRotation	Apply a rotation of the specified degrees when the view is in hovered state.

HoveredRotationX	Apply a rotation starting from the specified point on the x-axis when the view is in hovered state.
HoveredRotationY	Apply a rotation starting from the specified point on the y-axis when the view is in hovered state.
PressedRotation	Apply a rotation of the specified degrees when the view is in pressed state.
PressedRotationX	Apply a rotation starting from the specified point on the x-axis when the view is in pressed state.
PressedRotationY	Apply a rotation starting from the specified point on the y-axis when the view is in pressed state.
NormalTranslationX	Moves the view starting from the specified point on the x-axis when the view is in normal state.
NormalTranslationY	Moves the view starting from the specified point on the y-axis when the view is in normal state.
HoveredTranslationX	Moves the view starting from the specified point on the x-axis when the view is in hovered state.
HoveredTranslationY	Moves the view starting from the specified point on the y-axis when the view is in hovered state.
PressedTranslationX	Moves the view starting from the specified point on the x-axis when the view is in pressed state.
PressedTranslationY	Moves the view starting from the specified point on the y-axis when the view is in pressed state.

The sample project does not contain an example of these properties, but you can try yourself by modifying the previous snippet as follows.

```
<StackLayout Orientation="Horizontal"
         HorizontalOptions="CenterAndExpand"
         Padding="20"
         xct:TouchEffect.AnimationDuration="500"
```

```
            xct:TouchEffect.PulseCount="2"
            xct:TouchEffect.NormalBackgroundColor="Gold"
            xct:TouchEffect.PressedBackgroundColor="Orange"
            xct:TouchEffect.PressedRotation="15"
            xct:TouchEffect.PressedTranslationX="50"
            xct:TouchEffect.PressedTranslationY="100"
            xct:TouchEffect.PressedAnimationDuration="500"
            xct:TouchEffect.Command="{Binding Command,
                Source={x:Reference Page}}">
        <Label Text="TITLE"
           TextColor="White"
           FontSize="Large"/>

</StackLayout>
```

If you now run this code, you will see the view translating on the x and y axes when it is pressed.

Chapter summary

The effects provided by the Xamarin Community Toolkit make it very quick and easy to implement functionalities of extremely common requirements, and it will save a lot of time during development. This includes (but it is not limited to) removing borders to **`Entry`** views, managing the safe area on modern iPhones, preselecting text inside **`Entry`** and **`Editor`** views, or adding interaction capabilities to any view.

Because saving development time is at the core of this library, in the next chapter you will learn how to save more with reusable converters.

Chapter 4 Saving Time with Reusable Converters

In Xamarin.Forms, and more generally, in platforms based on XAML, data-binding happens between a source property and a target property. If both are of the same type, data-binding is quick. If properties are of different types, for example when you need to bind an **`Image.Source`** property to an image that is represented by a Base64 string, data-binding requires an appropriate value converter that converts the Base64 string into an **`ImageSource`** type. This is accomplished via value converters, which are objects that implement the **`IValueConverter`** interface.

As for the other chapters, the assumption here is that you already know how value converters work. If not, I recommend that you read the official documentation before you continue. The Xamarin Community Toolkit provides a collection of reusable value converters that are extremely common in a lot of applications, so that you do not need to reinvent the wheel every time, getting rid of many source code files.

This chapter describes the available value converters in the library, and it will walk through the examples available with the official sample project. All the converters are located under the **Converters** folder of the Xamarin.CommunityToolkit project.

Note: For consistency with a book of the* Succinctly *series, it is not possible to show and discuss the class definition of each converter. For this reason, you will learn how to use each converter and which parameters they need, without going into the details of the architecture. Converter classes are very easy to understand, so you can always look at their definition analyzing the .shared.cs files located under the Converters folder of the Xamarin.CommunityToolkit project. In addition, notice that the `ItemTappedEventArgsConverter` and `ItemSelectedEventArgsConverter` will be discussed in Chapter 5, because they work in combination with the `EventToCommandBehavior` class, discussed with behaviors.

Converting from bool to Object

Sometimes you might need to convert objects from type **`bool`** to a different type. An example is a user choice made through views like the **`Switch`** or the **`CheckBox`**, whose **`true`** or **`false`** value will change the value or status of a completely different view or class instance.

For such situations, the Xamarin Community Toolkit exposes the **`BoolToObjectConverter`** class, which is demonstrated in the Pages\Converters\BoolToObjectConverter.xaml file of the sample project. For a better understanding, consider Figure 32, where you can see that the color of a **`BoxView`** changes depending on the **`true`** or **`false`** value of the **`Switch`**.

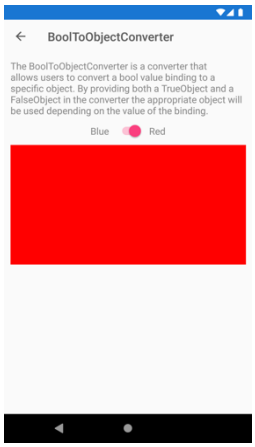


*Figure 32: Binding and converting a value of type **`bool`** to **`Color`***

In this example, the **`bool`** value coming from the **`Switch`** view is converted into a **`Color`**. However, the conversion is possible to any other type for maximum flexibility. Now, let's see how this works in code. The **`BoxView`** that displays the color is declared as follows.

```
<BoxView
    BindingContext="{x:Reference Name=ColorToggle}"
    HeightRequest="200"
    WidthRequest="200"
    Color="{Binding Path=IsToggled,
    Converter={StaticResource BoolToObjectConverter}}" />
```

Notice how the **`Color`** property is bound to the **`IsToggled`** property of the **`Switch`**, and how the **`BoolToObject`** converter has been specified to convert from one type to another. The converter declaration in the page resources looks like the following.

```
<ResourceDictionary>
    <converters:BoolToObjectConverter x:Key="BoolToObjectConverter"
     FalseObject="#0000FF" TrueObject="#FF0000" />
</ResourceDictionary>
```

The **`FalseObject`** and **`TrueObject`** properties specify which values must be returned when the source property is **`false`** or **`true`**, respectively. Both properties are of type **`object`**, so you can return anything that derives from **`object`** itself (which means any .NET type).

Converting from byte arrays to images

In the real world, images are often stored inside databases or retrieved via API calls, and the way they are stored in a data source is typically a Base64 string or a byte array, in case the image is retrieved by a **`Stream`** object.

The **Image** view in Xamarin.Forms cannot display a byte array as an image directly, so a converter is required, and this is something developers do in almost every project. The Xamarin Community Toolkit simplifies your code base by offering the **ByteArrayToImageSourceConverter**, which is demonstrated in the Pages\Converters\ByteArrayToImageSourceConverterPage.xaml.

In this page, you can see that the converter is declared in the simplest way possible.

```
<ResourceDictionary>
    <xct:ByteArrayToImageSourceConverter
        x:Key="ByteArrayToImageSourceConverter" />
</ResourceDictionary>
```

It can be used when binding a property of type **ImageSource**, for example the **Source** property of the **Image** view or of the **AvatarView**, like in the current example.

```
<xct:AvatarView Size="300"
     Source="{Binding Avatar,
     Converter={StaticResource ByteArrayToImageSourceConverter}}"
     HorizontalOptions="Center" VerticalOptions="Center" />
```

In the sample project, the **ByteArrayToImageSourceViewModel** class (which is the data source of the page) exposes a property called **Avatar**, of type **byte?[]**. This property will contain the profile picture of a person, which is retrieved from the GitHub contributors to the library. The ViewModel contains all the code that downloads the image and assigns it to the property, but the relevant part is the following.

```
using var client = new HttpClient();
using var response = await client.GetAsync(avatarUrl);

if (!response.IsSuccessStatusCode)
      return;

var imageBytes =
    await response.Content.ReadAsByteArrayAsync().ConfigureAwait(false);

Avatar = imageBytes;
```

The **Avatar** property is therefore assigned with a byte array; at the XAML level, when the assignment happens, the data-binding engine will invoke the converter that will return an **ImageSource** object, which is accepted by the **Source** property of the **AvatarView**. If you run the app, you will see how the image will appear after a few seconds required to download it from GitHub.

Note: The reason I'm not providing a screenshot from the sample app here is that the image retrieved by the code represents a real person. Though the image is hosted on GitHub and possibly is public domain, I consider it more appropriate not to show it here without explicit permission.

Converting dates to coordinated universal time

When you have to work with different markets in different time zones, you likely need to represent dates in a way that is the same across countries. This is accomplished by converting dates to the coordinated universal time (UTC).

In C# and .NET, this can be done using the **DateTimeOffset** structure. However, most of the views that allow working with dates (such as the **DatePicker**) natively support objects of type **DateTime**. For this reason, the Xamarin Community Toolkit provides the **DateTimeOffsetConverter**, which makes it easy to bind an object of type **DateTimeOffset** by converting an object of type **DateTime** into its **DateTimeOffset** counterpart (and vice versa).

If you look into the XAML of the Pages\Converters\DateTimeOffsetConverterPage.xaml file, you will see that the converter is declared as follows.

```
<ResourceDictionary>
    <xct:DateTimeOffsetConverter x:Key="DateTimeOffsetConverter" />
</ResourceDictionary>
```

The usage is then very simple, as you can bind a property of type **DateTimeOffset** passing the converter, as demonstrated in the following declaration of the **DatePicker** view.

```
<DatePicker Date="{Binding TheDate,
       Converter={StaticResource DateTimeOffsetConverter}}"
       Margin="16" HorizontalOptions="Center" />
```

The **TheDate** property is declared in the **DateTimeOffsetConverterViewModel** class, whose simple definition is the following.

```
public class DateTimeOffsetConverterViewModel : BaseViewModel
{
      DateTimeOffset theDate = DateTimeOffset.Now;

      public DateTimeOffset TheDate
      {
            get => theDate;
            set => SetProperty(ref theDate, value);
      }
}
```

If you run the sample project, you will see the converter in action in the appropriate page, shown in Figure 33.

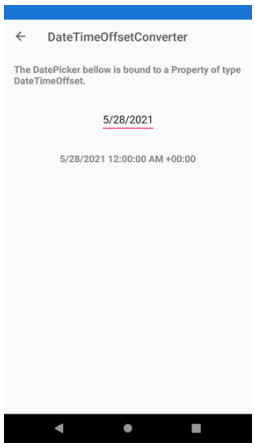


Figure 33: Converting a date to UTC

This converter is very useful, because it simplifies the way you address date problems with different time zones.

Converting decimal numbers to integers

If you need to convert decimal numbers of type **`double`** into integers of type **`int`**, you can leverage the **`DoubleToIntConverter`** class without implementing a conversion logic on your own. It is demonstrated in the Pages\Converters\DoubleToIntConverterPage.xaml file.

For an understanding of the result, look at Figure 34, where you can see an input decimal number converted into an integer.

Figure 34: Converting decimal numbers into integers

The declaration of the converter in XAML is simple, as usual.

```
<ResourceDictionary>
    <xct:DoubleToIntConverter x:Key="DoubleToIntConverter" />
</ResourceDictionary>
```

The relevant part of the code is in the declaration of the **Label**, which is bound to a property called **Input**, of type **double**, and shows the result of the conversion to **int**.

```
<Label
    Padding="7,0,0,0"
    Text="{Binding Path=Input,
    Converter={StaticResource DoubleToIntConverter},
    ConverterParameter=1}"
    TextColor="{StaticResource NormalLabelTextColor}" />
```

Notice how a **ConverterParameter** is passed to the converter. This value is the ratio for the conversion, which represents the division of the multiplier by the denominator, and that equals 1 by default. This division is used to determine how many decimal numbers the input **double** should be rounded by. This can be better understood by looking at the **Convert** method of the converter.

```
public object Convert(object? value, Type? targetType,
       object? parameter, CultureInfo? culture)
      => value is double result
            ? (int)Math.Round(result * GetParameter(parameter))
            : throw new ArgumentException("Value is not a valid double",
            nameof(value));
```

The **parameter** argument matches the **ConverterParameter** passed in XAML. The **GetParameter** method returns the supplied ratio, if available and if a valid number; otherwise, it returns the value of a property called **Ratio**, declared in the converter and assigned with 1.

```
double GetParameter(object? parameter)
      => parameter == null
      ? Ratio
      : parameter switch
      {
            double d => d,
            int i => i,
            string s => double.TryParse(s, out var result)
                  ? result
                  : throw new ArgumentException(
                  "Cannot parse number from the string",
                  nameof(parameter)),
                  _ => 1,

};
```

This ratio is then multiplied by the input **double** number. This allows you to have control of how the conversion is done.

Converting enumerations into Boolean values

If you need to convert values from an **`enum`** into a **`bool`**, you can leverage the **`EnumToBoolConverter`** class. This converter is demonstrated in the Pages\Converters\EnumToBoolConverterPage.xaml file, and it allows you to specify which values from the enumeration must be treated as **`true`**; then you can use the resulting **`bool`** as you like.

The sample page allows for selecting a state for a hypothetical issue on GitHub, and it looks like the one shown in Figure 35.

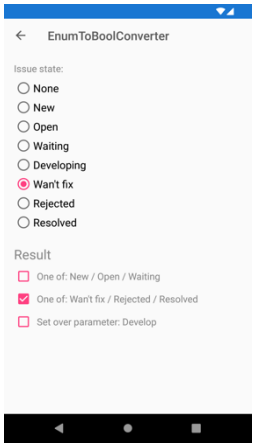


Figure 35: Converting enumeration values into Boolean values

States are represented by the **`IssueState`** enumeration, defined in the **`EnumToBoolConverterViewModel`** class as follows.

```
public enum IssueState
{
      None = 0,
      New = 1,
      Open = 2,
      Waiting = 3,
      Developing = 4,
      WantFix = 5,
      Rejected = 6,
      Resolved = 7

}
```

The ViewModel also defines a property called **`SelectedState`** of type **`IssueState`**, used to bind the current state to the user interface. Converter instances are declared in the page as follows.

```
<xct:EnumToBoolConverter x:Key="OpenConverter">
    <xct:EnumToBoolConverter.TrueValues>
        <vm:IssueState>New</vm:IssueState>
```

```
        <vm:IssueState>Open</vm:IssueState>
        <vm:IssueState>Waiting</vm:IssueState>
    </xct:EnumToBoolConverter.TrueValues>
</xct:EnumToBoolConverter>
<xct:EnumToBoolConverter x:Key="ClosedConverter">
    <xct:EnumToBoolConverter.TrueValues>
        <vm:IssueState>WantFix</vm:IssueState>
        <vm:IssueState>Rejected</vm:IssueState>
        <vm:IssueState>Resolved</vm:IssueState>
    </xct:EnumToBoolConverter.TrueValues>
</xct:EnumToBoolConverter>

<xct:EnumToBoolConverter x:Key="ManualConverter" />
```

The first two instances specify the values that should be considered **true** directly in the definition, whereas the third one does not specify any value. This is intentional; you will shortly see how to assign a value at a later stage.

There are multiple instances of the converter because the user interface contains three check boxes at the bottom, each representing one of the possible states, and each bound to an instance of the converter.

In the XAML, you will see a number of **RadioButton** views, each bound to a value of the **IssueState** enumeration, but more specifically, you will see how three **CheckBox** views are bound to the **SelectedState** property, so they also need to point to the appropriate converter to convert this type into **bool**.

```
<Label Grid.Row="2" Grid.ColumnSpan="2" FontSize="Large"
       Margin="0,16,16,0">Result</Label>
<CheckBox Grid.Row="3" IsEnabled="False"
          IsChecked="{Binding SelectedState,
    Converter={StaticResource OpenConverter}, Mode=OneWay}" />
<Label Grid.Row="3" Grid.Column="1" Text="One of: New / Open / Waiting" />
<CheckBox Grid.Row="4" IsEnabled="False"
          IsChecked="{Binding SelectedState,
    Converter={StaticResource ClosedConverter}, Mode=OneWay}" />
<Label Grid.Row="4" Grid.Column="1" Text="One of: Wan't fix / Rejected /
Resolved" />
<CheckBox Grid.Row="5" IsEnabled="False"
          IsChecked="{Binding SelectedState,
    Converter={StaticResource ManualConverter},
    ConverterParameter={x:Static vm:IssueState.Developing}, Mode=OneWay}" />

<Label Grid.Row="5" Grid.Column="1" Text="Set over parameter: Develop" />
```

Notice how the last **CheckBox** is passing a value from the **IssueState** enumeration to the bound converter, and this demonstrates how you can pass a value to be evaluated as **true** at data-binding time; you are not limited to declaring **true** values in advance.

Converting objects into Boolean values and comparing for equality

The Xamarin Community Toolkit exposes the **EqualConverter**, which allows developers to convert any value binding to a **bool**, depending on whether or not it is equal to a specific value. The initial binding contains the object that will be compared, and the **ConverterParameter** contains the object to which to compare it.

In the sample project, this is demonstrated in the Pages\Converters\EqualConverterPage.xaml. More specifically, it demonstrates how to return **true** or **false** if the input value equals 100, which is the value specified in the **ConverterParameter**. Figure 36 shows what the example looks like.

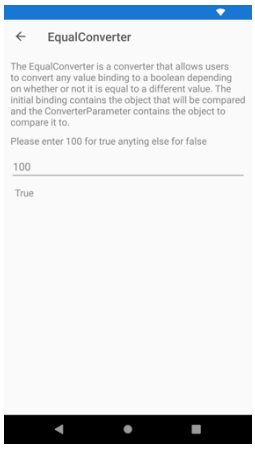


Figure 36: Converting objects into Boolean values for equality comparison

The converter is simply declared as follows.

```
<ResourceDictionary>
    <xct:EqualConverter x:Key="EqualConverter" />
</ResourceDictionary>
```

The last **Label** from the top shows how the converter is actually used.

```
<Label
    Padding="7,0,0,0"
    BindingContext="{x:Reference Name=ExampleText}"
    Text="{Binding Path=Text, Converter={StaticResource EqualConverter},
    ConverterParameter=100}" TextColor="{StaticResource
    NormalLabelTextColor}" />
```

The **EqualConverter** class checks if an object, the user input in this case, equals the value specified in the **ConverterParameter** and returns **true** or **false**, depending on the result of the comparison for equality.

As an opposite tool, the **NotEqualConverter** works exactly like **EqualConverter**, except for the fact that it returns **true** if two objects are not equal and **false** if they are equal.

Displaying images from resources

As you might know, it is possible to store images locally in the app resources. To accomplish this, you add the desired image files to the shared project. Then, for each image, in the **Properties** tool window, you set the **Build Action** property as **EmbeddedResource**, whereas the **Copy To Output Directory** property is set as **Do not copy**.

Retrieving and displaying an image from the app resources is very easy to do; if you had an **`Image`** view called **`Image1`**, you would write the following line of code.

```
Image1.Source = ImageSource.FromResource(imageId,
                Application.Current.GetType().GetTypeInfo().Assembly);
```

The **`imageId`** object is the image identifier, and has the following form: *AssemblyName.Folder.Filename*. If you had an image called **myimage.jpg** inside a folder called **Resources** in a project whose resulting assembly name is **MyProject**, the image ID would be **MyProject.Resources.myimage.jpg**.

Tip: The image ID is case-sensitive, so it must exactly match the casing of assembly names, folder names, and image file names.

This approach works if you can directly access your views in C# code, but it does not work if you have **`Image`** views that are data-bound to properties in a ViewModel, because data-binding lives in XAML. You need a converter, and the Xamarin Community Toolkit offers one ready to use, called **`ImageResourceConverter`**. What it does is basically implement code similar to the line you saw previously but building the image ID is still your responsibility.

In the sample project, it is demonstrated in the Pages\Converters\ImageResourceConverterPage.xaml file and, if you run the example in the app and tap the button to change the image, the result of the conversion is represented in Figure 37.

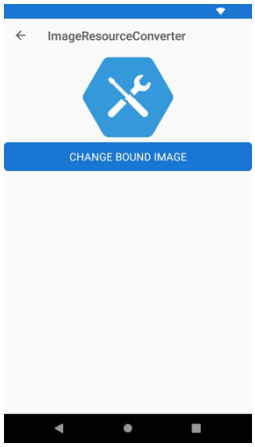


Figure 37: Displaying images from app resources

The sample project uses two local images under the **Images** folder, called button.png and logo.png. Code Listing 7 shows instead the code for the ViewModel, located at ViewModels\ImageResourceConverterViewModel.cs, where there is code that builds the image ID the proper way.

Code Listing 7

```
using System.Windows.Input;
using Xamarin.Forms;

namespace Xamarin.CommunityToolkit.Sample.ViewModels.Converters
{
      public class ImageResourceConverterViewModel : BaseViewModel
      {
            const string img1 = "button.png";
            const string img2 = "logo.png";
            const string imagesPath = "Images";

            string defaultNamespace;

            string? imageName;

            public string? ImageName
            {
                  get => imageName;
                  set => SetProperty(ref imageName, value);
            }

            ICommand? changeImageCommand;

            public ICommand ChangeImageCommand =>
                   changeImageCommand ??= new Command(() =>
                  {
                        ImageName = (ImageName?.Equals(
                        BuildEmbededImagePath(img1)) ?? false) ?
                                          BuildEmbededImagePath(img2) :
                                          BuildEmbededImagePath(img1);
                  });

            public ImageResourceConverterViewModel()
            {
                  defaultNamespace = System.Reflection.Assembly.
                  GetExecutingAssembly().GetName().Name;
                  ImageName = BuildEmbededImagePath(img1);
            }

            string BuildEmbededImagePath(string imgName)
                  => $"{defaultNamespace}.{imagesPath}.{imgName}";
      }
```

```
}
```

As you can see, the constructor retrieves the assembly name with the first line of code. This is required to build the image ID, which is actually performed by the **BuildEmbeddedImagePath** method that takes the image file name as an argument. The resulting image ID is assigned to the **ImageName** property, and the image name varies depending on the user tapping a button on the user interface, an action handled via the **ChangeImageCommand**. In the XAML, the converter is declared the usual way.

```
<pages:BasePage.Resources>
    <xct:ImageResourceConverter x:Key="ImageResourceConverter"/>
</pages:BasePage.Resources>
```

The **Image** view is bound to the **ImageName** property of the ViewModel, and invokes the **ImageResourceConverter** as follows.

```
<Image WidthRequest="150"
       HeightRequest="150"
       Source="{Binding ImageName,
       Converter={StaticResource ImageResourceConverter}}"/>
```

In this way, you can quickly display an image from the app resources, taking advantage of the data-binding automation—and without the need to write C# code and handle the views' properties manually.

Converting an index into an array item

Sometimes, you might need to bind an item inside an array to the user interface, but you only have the index of the item itself. Simplifying this process is possible with the **IndexToArrayItemConverter** class, which is demonstrated in the Pages\Converters\IndexToArrayItemConverter.xaml page. The way it works is quite simple. If you look at the sample code, you will see the following array definition, along with the declaration of the converter.

```
<ResourceDictionary>
    <xct:IndexToArrayItemConverter x:Key="IndexToArrayItemConverter" />
    <x:Array x:Key="ValuesArray" Type="{x:Type x:String}">
        <x:String>Value1</x:String>
        <x:String>Value2</x:String>
        <x:String>Value3</x:String>
        <x:String>Value4</x:String>
        <x:String>Value5</x:String>
    </x:Array>
</ResourceDictionary>
```

The declared array contains a series of **string** values, and the converter is declared the usual way. The sample app allows entering an index of the array from an **Entry**, whose **Text** property is bound to another property in the ViewModel called **Index**. **Index**'s sole purpose is connecting the **Entry** with the **Label**, which displays the correspondent item of the array. It is declared as follows.

```
<Label
    Padding="7,0,0,0"
    Text="{Binding Path=Index, Converter={StaticResource
           IndexToArrayItemConverter},
           ConverterParameter={StaticResource ValuesArray}}"
    TextColor="{StaticResource NormalLabelTextColor}" />
```

The **Index** property is an **int** that represents the index of the desired item in the array. The **Converter** property of the binding points to the converter, whereas the **ConverterParameter** property allows for specifying the array in which the converter must search the item that matches the supplied index. Behind the scenes, the **IndexToArrayItemConverter** returns the result of the invocation to the **Array.GetValue** method, which returns the item corresponding to the supplied index in the form of an **object** instance.

Converting from integer to Boolean

If you need to bind integer values to **false** and **true** Boolean values, the Xamarin Community Toolkit provides an easy solution: the **IntToBoolConverter** class, demonstrated in the Pages\Converters\IntToBoolConverter.xaml page. If the integer is 0, the converter returns **false**. Any other integer values will be converted to **true**. Declaring the converter is simple.

```
 <ResourceDictionary>
     <xct:IntToBoolConverter x:Key="IntToBoolConverter" />
 </ResourceDictionary>
```

The sample code that demonstrates the conversion is the following, where **Number** is an integer property defined in the backing ViewModel, and just stores the input value.

```
<Entry
    x:Name="ExampleText"
    Placeholder="0 for false other for true"
    Text="{Binding Number}"
    TextColor="{StaticResource NormalLabelTextColor}" />
<Label
    Padding="7,0,0,0"
    Text="{Binding Path=Number,
    Converter={StaticResource IntToBoolConverter}}"
    TextColor="{StaticResource NormalLabelTextColor}" />
```

The converter implementation is very easy, since it just returns the result of the **!=** operator compared to 0, but it is still very useful and of quick reuse.

Inverting Boolean bindings

When you data-bind properties of type **`bool`**, the binding happens when the source property is **`true`**. However, there are situations where you need to make the binding happen when the source property is **`false`**, such as displaying an error state view when the result of an API call is not successful.

To accomplish this, the Xamarin Community Toolkit provides the **`InvertedBoolConverter`** class, demonstrated in the Pages\Converters\InvertedBoolConverter.xaml file. The converter is declared as follows.

```
<ResourceDictionary>
    <xct:InvertedBoolConverter x:Key="InvertedBoolConverter" />
</ResourceDictionary>
```

In the sample code, a **`Label`** displays a string that represents the opposite state of the **`Switch`** view, as follows.

```
<Label
    BindingContext="{x:Reference Name=ColorToggle}"
    Text="{Binding Path=IsToggled,
    Converter={StaticResource InvertedBoolConverter}}" />
```

`ColorToggle` is the source view of the binding and is defined as follows:

```
<Switch x:Name="ColorToggle" IsToggled="False" />
```

In the sample app, the result of the code is shown in Figure 38.

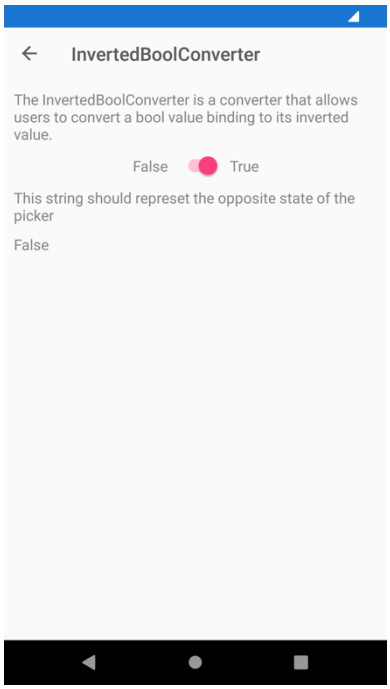


Figure 38: Inverting the binding of a ***`bool`*** *value*

Tip: In my real-world experience, I have implemented and used a similar converter to control the visibility of views based on the status of the data context.

Detecting null string values

The Xamarin Community Toolkit provides two converters that allow for detecting if a string is null, and that return a **`bool`** value if this is the case. The **`IsNullOrEmptyConverter`** returns **`false`** if a string is null or if it only contains white spaces, whereas the **`IsNotNullOrEmptyConverter`** returns **`true`** if a string is not null or if it does not contain only white spaces.

In the sample project, they are demonstrated in the Pages\Converters\IsNullOrEmptyConverterPage.xaml and Pages\Converters\IsNotNullOrEmptyConverterPage.xaml files, respectively. They both display a list of strings, and they are bound to an individual string via the properties defined in the **`IsNullOrEmptyConverterViewModel`** class, as follows.

```
public ObservableCollection<string> DummyItemSource { get; } =
      new ObservableCollection<string>
{
      "Dummy Item 0",
      "Dummy Item 1",
      "Dummy Item 2",
      "Dummy Item 3",
      "Dummy Item 4",
```

```
        "Dummy Item 5",
};

public string? SelectedItem
{
        get => selectedItem;
        set => SetProperty(ref selectedItem, value);

}
```

The result of the binding is then different depending on the converter. Figure 39 demonstrates this, where on the left you can see the result of the **`IsNullOrEmptyConverter`** class, and on the right the result of the **`IsNotNullOrEmptyConverter`** one.

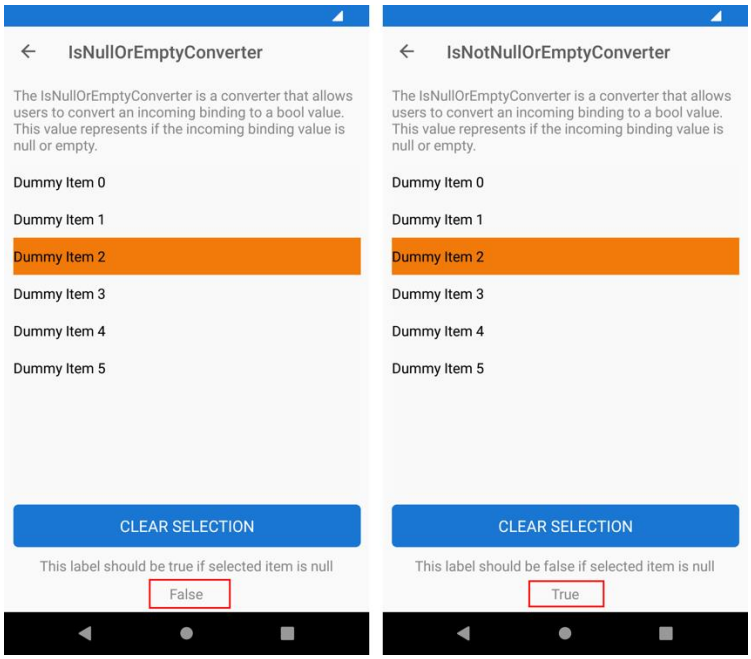


Figure 39: Checking for null values

The declaration of the converters is very simple, as usual. The following is how you declare the **`IsNullOrEmptyConverter`**.

```
<ResourceDictionary>
    <xct1:IsNullOrEmptyConverter x:Key="IsNullOrEmptyConverter" />
</ResourceDictionary>
```

Following is instead how you declare the **`IsNotNullOrEmptyConverter`**.

```
<ResourceDictionary>
    <xct:IsNotNullOrEmptyConverter x:Key="IsNotNullOrEmptyConverter" />
</ResourceDictionary>
```

In both examples, there is a **`CollectionView`** that displays the list of strings with the following data template.

```
<CollectionView VerticalOptions="StartAndExpand"
 HorizontalOptions="Center" SelectionMode="Single"
 ItemsSource="{Binding  DummyItemSource}"
 SelectedItem="{Binding SelectedItem}">
    <CollectionView.ItemTemplate>
        <DataTemplate>
            <Label Text="{Binding .}" Margin="10" TextColor="Black" />
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>
```

Converters are then used in the last **Label** that displays the result of the converters themselves. More specifically, for the **IsNullOrEmptyConverter**, the declaration is the following.

```
<Label VerticalOptions="CenterAndExpand" HorizontalOptions="CenterAndExpand"
       Text="{Binding Path=SelectedItem,
       Converter={StaticResource IsNullOrEmptyConverter}}" />
```

For the **IsNotNullOrEmptyConverter**, the declaration is the following.

```
<Label VerticalOptions="CenterAndExpand" HorizontalOptions="CenterAndExpand"
       Text="{Binding Path=SelectedItem,
       Converter={StaticResource IsNotNullOrEmptyConverter }}" />
```

They are both used against the same string: only the result changes.

Checking data collections for null values

If your app displays data collections with a **CollectionView** control, you know that you can quickly implement empty states or error states via the **EmptyView** and **EmptyViewTemplate** properties, which allow for displaying a different view if the bound collection is null or empty. This is possible because the **CollectionView** checks the bound collection automatically. However, if your app has existed for years, it is very likely using **ListView** controls to display data collections, and implementing empty and error states is your own responsibility.

In order to simplify this, the Xamarin Community Toolkit exposes the **ListIsNotNullOrEmptyConverter** and the **ListIsNullOrEmptyConverter** classes, which respectively return **true** if a collection is not null or if it is null. They are demonstrated in the ListIsNotNullOrEmptyConverterPage.xaml and ListIsNullOrEmptyConverterPage.xaml files, both located under Page\Converters. Both sample pages work similarly, so I'm taking into consideration the ListIsNullOrEmptyConverterPage.xaml.

If you run the sample app and open this sample page, you will see that it's first a display about the fact that the data collection is empty. Then, if you click **Add**, an item will be added to the list so that the empty state will disappear in favor of the collection. Figure 40 demonstrates this.

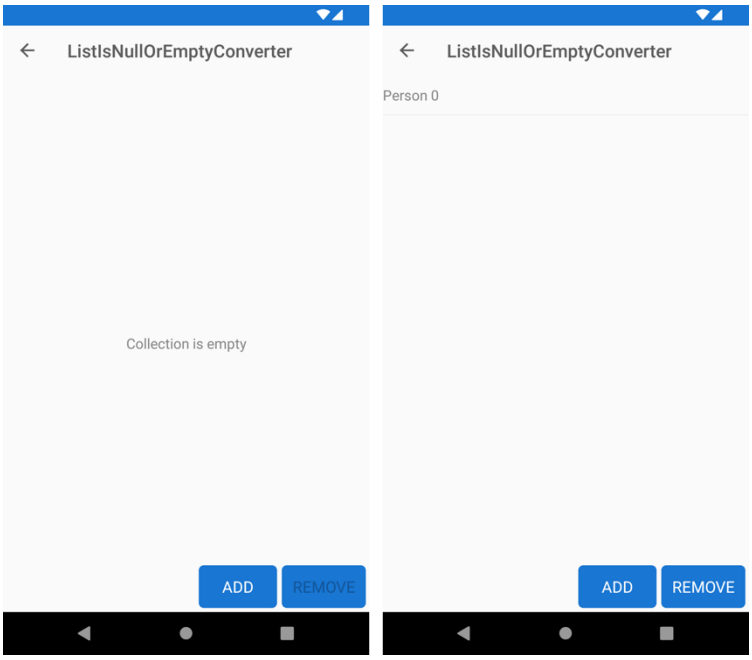


Figure 40: Displaying and hiding data lists

The bound data collection is called **`Items`**. It is exposed from the **`ListIsNullOrEmptyConverterViewModel`** class, and it's empty at startup. The way the **`ListView`** is displayed or hidden relies on both converters, assigned to the binding of the **`ListView`** and the **`Label`** for the empty state as follows (only the relevant lines are shown here).

```
<ListView ItemsSource="{Binding Items}"IsVisible="{Binding Items,
          Converter={xct:ListIsNotNullOrEmptyConverter}}">

…

<Label Text="Collection is empty" IsVisible="{Binding Items,
        Converter={xct:ListIsNullOrEmptyConverter}}"
```

The **`IsVisible`** property of the **`ListView`** will be **`true`** if this is the value returned by the **`ListIsNotNullOrEmptyConverter`**, which means that the data collection is not empty. Instead, the **`IsVisible`** property of the **`Label`** will be **`true`** if this is the value returned by the other converter, **`ListIsNullOrEmptyConverter`**.

With these simple converters, you can quickly implement empty states and error states against **`ListView`** controls, without the need to implement custom logic.

Converting a list of strings into one string

If you need to bind and convert an **IEnumerable<string>** object (and obviously derived collections such as **ObservableCollection<string>**) into one single string, you can leverage the **ListToStringConverter** class, which is demonstrated in the Pages\Converters\ListToStringConverterPage.xaml file. The ViewModel defines the following collection of string objects.

```
public ObservableCollection<string> DummmyItemSource { get; } =
      new ObservableCollection<string>
{
      "Dummy Item 0",
      "Dummy Item 1",
      "Dummy Item 2",
      "Dummy Item 3",
      "Dummy Item 4",
      "Dummy Item 5",

};
```

The converter is declared as follows.

```
<ResourceDictionary>
    <xct:ListToStringConverter x:Key="ListToStringConverter" Separator="/" />
</ResourceDictionary>
```

Notice how you can specify a separator for each string. The sample code converts this collection into one string with the following XAML.

```
<Label Text="{Binding DummyItemSource,
    Converter={StaticResource ListToStringConverter}}"
    TextColor="Black"/>
```

The resulting string will be the following.

```
Dummy Item 1/ Dummy Item 2/ Dummy Item 3/ Dummy Item 4/ Dummy Item 5
```

Changing string casing

Changing the casing of an incoming string in a data-binding is extremely common. For this reason, the Xamarin Community Toolkit provides the **TextCaseConverter**, which is demonstrated in the Pages\Converters\TextCaseConverterPage.xaml file. The converter is declared as follows.

```
<ResourceDictionary>
    <xct:TextCaseConverter x:Key="TextCaseConverter" Type="Upper" />
</ResourceDictionary>
```

The **Type** property allows you to specify the target casing. Possible options are **Upper**, **Lower**, **FirstUpperRestLower**, and **None**. In the XAML file, a **Label** displays the result of the conversion of some input text entered via the **Entry** view.

```
<Entry
    x:Name="ExampleText"
    Placeholder="Enter text here"
    Text="{Binding Input}"
    TextColor="{StaticResource NormalLabelTextColor}" />
<Label
    Padding="7,0,0,0"
    BindingContext="{x:Reference Name=ExampleText}"
    Text="{Binding Path=Text, Converter={StaticResource TextCaseConverter}}"
    TextColor="{StaticResource NormalLabelTextColor}" />
```

If you run the sample code, you will see the string converted to uppercase.

Tip: Keep in mind that the `FirstUpperRestLower` option capitalizes only the first character of the string. If you want to capitalize the first letter of each word in a string, you need to implement your own converter that returns the result of the invocation to the `System.Globalization.CultureInfo.CurrentCulture.TextInfo.ToTitleCase` method.

Using multiple converters together

The Xamarin Community Toolkit provides an easy way to use multiple converters at the same time, which might be very useful if you consider the number of converters in the toolkit, in the Xamarin.Forms code base, and possibly in your code base. To accomplish this, you can use the **MultiConverter** class, which is demonstrated in the Pages\Converters\MultiConverter.xaml page.

For a better understanding of how it works and of how powerful it is, let's consider the code of the sample page. The purpose of the sample page is converting to uppercase an input string and, by default, checking if it is different from the **ANDREI** string shown in Code Listing 1. This is possible by combining the **TextCaseConverter** class with the **NotEqualConverter** class, which you saw previously.

The **MultiConverter** class can contain multiple converters, as follows.

```
<xct:MultiConverter x:Key="myMultiConverter">
    <xct:TextCaseConverter />
    <xct:NotEqualConverter />
</xct:MultiConverter>
```

Basically, you just need to declare the converter and assign a key as the identifier, and then you nest all the converters you want to be used simultaneously. However, this is not enough, because converters might need parameters. These are supplied in the form of an **Array** of **MultiConverterParameter** objects, as follows.

```
<x:Array
    x:Key="multiParams"
    Type="{x:Type xct:MultiConverterParameter}">
    <xct:MultiConverterParameter
        ConverterType="{x:Type xct:TextCaseConverter}"
        Value="{x:Static xct:TextCaseType.Upper}" />
    <xct:MultiConverterParameter
        ConverterType="{x:Type xct:NotEqualConverter}"
        Value="ANDREI" />
</x:Array>
```

For each **MultiConverterParameter**, you need to assign the **ConverterType** property with the converter you want to use via the **x:Type** reference and the value. This must be of the type that is accepted by the converter as a parameter; otherwise, an exception will be thrown.

The last step is assigning the **MultiConverter** to the binding, and this is done exactly as you would do with an individual converter. In the sample code, this can be seen in the last **Label** definition.

```
<Label
    Text="{Binding EnteredName, Converter={StaticResource myMultiConverter},
    ConverterParameter={StaticResource multiParams}}"
    HorizontalOptions="CenterAndExpand" />
```

As you can see, you pass the identifier of the **MultiConverter** to the **Converter** property of the binding, and the identifier of the array of parameters to the **ConverterParameter** property of the binding. Running the sample app will demonstrate in practice how it works, but the biggest benefit to remember is that you can use multiple converters simultaneously, including converters defined outside of the Xamarin Community Toolkit.

Converting multiple Boolean values into a single one

The Xamarin Community Toolkit provides the **VariableMultiValueConverter** class, which allows for converting multiple **bool** values into a single one, based on the specified conditions. This converter is demonstrated in the Pages\Converters\VariableMultiValueConverterPage.xaml file. For a better understanding, consider Figure 41, which shows the converter in action.

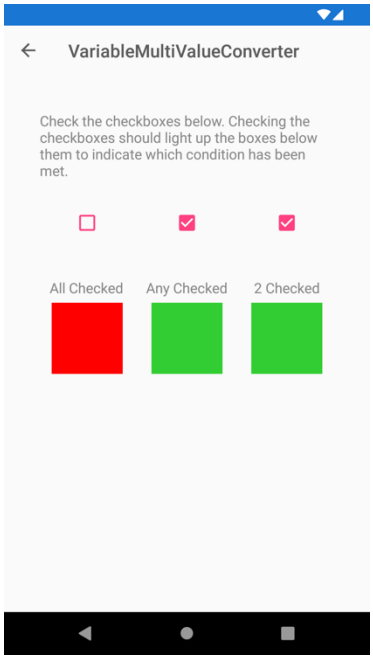


Figure 41: Converting multiple Booleans into a single one

If you play with the various check boxes, you will see the colored boxes change their color from red to green when the corresponding check boxes are flagged, and vice versa, depending on which conditions have been met.

Conditions can be better understood with some code. First, you declare as many **`VariableMultiValueConverter`** instances and as many conditions as you need, in this case three.

```
<pages:BasePage.Resources>
    <xct:VariableMultiValueConverter
     x:Key="AllTrueConverter" ConditionType="All" />
    <xct:VariableMultiValueConverter
     x:Key="AnyTrueConverter" ConditionType="Any" />
    <xct:VariableMultiValueConverter x:Key="TwoTrueConverter"
     ConditionType="Exact" Count="2" />
</pages:BasePage.Resources>
```

The **`ConditionType`** is an object of type **`MultiBindingCondition`**, an enumeration that exposes the following options: **`None`**, **`All`**, **`Any`**, **`Exact`**, **`GreaterThan`**, and **`LessThan`**. You will also need to supply a value for the **`Count`** property when you use the **`Exact`**, **`GreaterThan`**, and **`LessThan`** options.

Each of the declared converters must be assigned to the **`DataTrigger`** of the view to which it is applied, more specifically to the **`MultiBinding`** collection. The following code demonstrates this for the first **`BoxView`**, and the same thing is replicated on the other **`BoxView`** objects but changing the converter to which they refer.

```
<BoxView Grid.Row="2" Grid.Column="0" BackgroundColor="Red"
```

```
        VerticalOptions="Center" HorizontalOptions="Center"
        HeightRequest="80" WidthRequest="80">
    <BoxView.Triggers>
        <DataTrigger TargetType="BoxView" Value="true">
            <DataTrigger.Binding>
                <MultiBinding Converter="{StaticResource AllTrueConverter}">
                        <Binding Path="CheckBoxValue1" />
                        <Binding Path="CheckBoxValue2" />
                        <Binding Path="CheckBoxValue3" />
                    </MultiBinding>
            </DataTrigger.Binding>
            <Setter Property="BackgroundColor" Value="LimeGreen" />
       </DataTrigger>
    </BoxView.Triggers>
</BoxView>
```

With this approach, multiple **bool** values can talk to each other, and their combination can be converted into one single **bool**.

Chapter summary

Data-binding is one of the most powerful features in development platforms based on XAML. In many situations, you will need to bind views that support specific data types to objects of different types, and this is where value converters come in. The Xamarin Community Toolkit provides a lot of converters of common usage, so that you do not need to reinvent the wheel every time, which simplifies your code base. Value converters have then to do with data, and so do the validation behaviors described in the next chapters.

Chapter 5 Data Validation with Reusable Behaviors

In Xamarin.Forms (and, more generally, in XAML-based platforms), behaviors let you add functionality to user interface views without having to subclass them. The actual functionality is implemented in a behavior class, but it is simply attached to a control in XAML for easy usage.

As you can imagine, this chapter assumes you already have knowledge of behaviors. If you are new to this feature, I strongly recommend you first get started with the official documentation. Here you will find guidance on the behaviors that are most relevant for data validation in the Xamarin Community Toolkit. Keep the official sample project open in Visual Studio, as this will serve as the reference for the discussion.

Note: Among others, the Xamarin Community Toolkit also provides behaviors to animate visual elements. These are not covered in this chapter, whose goal is explaining how to perform data validation in an efficient way. In addition, the `EventToCommandBehavior` is not discussed in this chapter because it is related to working with the Model-View-ViewModel pattern, so it makes more sense to discuss it in Chapter 6.

Behaviors' common properties

All the behaviors exposed by the Xamarin Community Toolkit derive from the **`Xamarin.CommunityToolkit.Internals.Behaviors.ValidationBehavior`** class. This base class provides a number of properties that all derived behaviors expose. Table 6 summarizes the most relevant and used properties.

Table 6: Behaviors' common properties

`IsValid`	Returns **`true`** when the bound data is considered valid.
`IsNotValid`	Returns **`true`** when the bound data is considered invalid.
`IsValidStyle`	An object of type **`Style`** that is applied to the visual element when the bound data is considered valid.
`InvalidStyle`	An object of type **`Style`** that is applied to the visual element when the bound data is considered invalid.
`Value`	The data to be validated.

`IsRunning`	Returns **`true`** when the validation process is running.
`Flags`	Provides an enumerated value that specifies how to handle validation. Supported values are **`None`**, **`ValidateOnAttaching`**, **`ValidateOnFocusing`**, **`ValidateOnValueChanging`**, and **`ForceMakeValidWhenFocused`**.

Most of these properties will be recalled very often in the next paragraphs, so it's important for you to know what they are about.

> ***Note: In the sample project, the `IsValidStyle` and `IsInvalidStyle` properties have the same value for most behaviors, so you will get screenshots only when appropriate.***

Validating an email address

Validating an email address that the user enters via an **`Entry`** view is a very common task, and the Xamarin Community Toolkit makes it simpler by exposing the **`EmailValidationBehavior`** class. In the sample project, this is demonstrated in the Behaviors\EmailValidationBehaviorPage.xaml file.

For a better understanding, consider Figure 42, where an invalid email address has been entered.

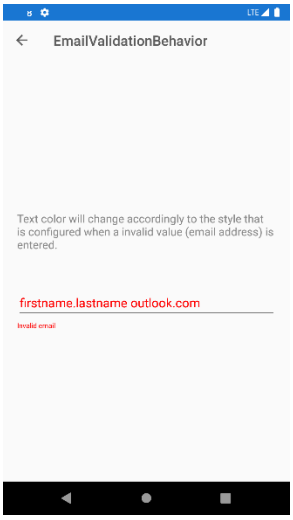


Figure 42: Email address validation

The entered string has been recognized as invalid by the **EmailValidationBehavior** object, which automatically performs formal validation. This behavior exposes two properties, **IsValid** and **IsNotValid**, which return **true** if the email address is either valid or invalid.

Based on these property values, you can style other UI elements; for example, with an invalid email address, the label you see in Figure 42 is set as visible, and the color for the invalid text is in red. When the behavior recognizes the supplied email address as valid, the views' styles return to their original state. Code Listing 8 shows how this is implemented.

Code Listing 8

```
<?xml version="1.0" encoding="UTF-8"?>
<pages:BasePage xmlns="http://xamarin.com/schemas/2014/forms"
                xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
                xmlns:pages="clr-
namespace:Xamarin.CommunityToolkit.Sample.Pages"

x:Class="Xamarin.CommunityToolkit.Sample.Pages.Behaviors.EmailValidationBeh
aviorPage">

    <pages:BasePage.Resources>
        <Style x:Key="InvalidEntryStyle" TargetType="Entry">
            <Setter Property="TextColor" Value="Red" />
        </Style>
    </pages:BasePage.Resources>

    <StackLayout Padding="{StaticResource ContentPadding}"
                 Spacing="50"
                 VerticalOptions="CenterAndExpand">
        <Label Text="Text color will change accordingly to the style that
is configured when a invalid value (email address) is entered." />
        <StackLayout Spacing="3">
            <Entry Placeholder="Email">
                <Entry.Behaviors>
                    <xct:EmailValidationBehavior
                        x:Name="EmailValidator"
                        DecorationFlags="Trim"
                        InvalidStyle="{StaticResource InvalidEntryStyle}"/>
                </Entry.Behaviors>
            </Entry>
            <Label Text="Invalid email"
                   TextColor="Red"
                   FontSize="10"
                   IsVisible="{Binding IsNotValid,
                   Source={x:Reference EmailValidator}}"
                   />
        </StackLayout>
    </StackLayout>
```

```
</pages:BasePage>
```

Notice how the **InvalidStyle** property of the behavior is assigned with a style that is applied to the **Entry** only when the email address is recognized as invalid. The **DecorationFlags** property allows for managing white spaces in the string. Other supported values, other than **Trim** and all self-explanatory, are **None**, **NormalizeWhiteSpace**, **NullToEmpty**, **Trim**, **TrimEnd**, and **TrimStart**. Finally, notice how the **Label** becomes visible only when the **IsNotValid** property of the behavior becomes true. This is a common approach to display an error message close to the input box.

Validating characters in a string

The Xamarin Community Toolkit makes it easy to validate characters in a string. This is possible via the **CharactersValidationBehavior** class, which you can attach to an **Entry** or **Editor** views. In the sample project, it is demonstrated in the Behaviors\CharactersValidationBehavior.xaml file and its code-behind.

The **CharactersValidationBehavio**r requires you to specify values for the following properties:

- **CharacterType**: This is of type **CharacterType** and establishes the validation criterion. This is an enumeration so multiple values can be combined together. More details are coming shortly.
- **MaximumCharacterCount**: The maximum allowed length for the input string.
- **MinimumCharacterCount**: The minimum number of characters for the input string.

Now consider Figure 43. Notice that the sample code offers two examples, and the second one is based on visual states. I will use the first one.

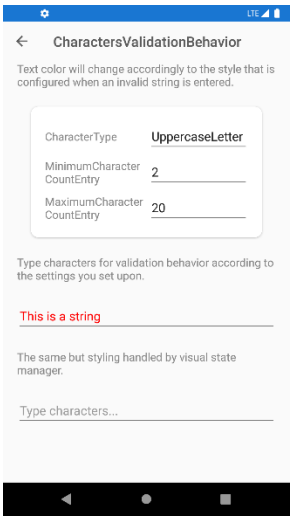


Figure 43: Validating characters in a string

In the example, the **UppercaseLetter** criterion requires the input string to be all uppercase; otherwise, it will be considered invalid. In addition, minimum and maximum lengths are also specified. Because the string in the figure is not all uppercase, it is considered invalid and displayed in red. The XAML code for this example looks like the following.

```
<Entry Placeholder="Type characters...">
  <Entry.Behaviors>
     <xct:CharactersValidationBehavior CharacterType="{Binding SelectedItem,
          Source={x:Reference CharacterTypePicker}}"
          InvalidStyle="{StaticResource InvalidEntryStyle}"
          ValidStyle="{StaticResource ValidEntryStyle}"
          MaximumCharacterCount="{Binding Text,
                  Source={x:Reference MaximumCharacterCountEntry}}"
           MinimumCharacterCount="{Binding Text,
                  Source={x:Reference MinimumCharacterCountEntry}}"/>
    </Entry.Behaviors>
</Entry>
```

MinimumCharacterCount and **MaximumCharacterCount** are data-bound to the **Text** property of other entries in the user interface. **CharacterType** is data-bound to the **SelectedItem** property of a **Picker** view, whose **ItemsSource** is assigned in code-behind with the following code.

```
public List<CharacterType> CharacterTypes { get; } =
       new List<CharacterType>()
{
      CharacterType.LowercaseLetter,
      CharacterType.UppercaseLetter,
      CharacterType.Letter,
      CharacterType.Digit,
      CharacterType.Alphanumeric,
      CharacterType.Whitespace,
      CharacterType.NonAlphanumericSymbol,
      CharacterType.LowercaseLatinLetter,
      CharacterType.UppercaseLatinLetter,
      CharacterType.LatinLetter,
      CharacterType.Any

};
```

The values of the **CharacterType** enumeration are really self-explanatory. Just to give you some examples, if you select **Alphanumeric**, the input string is considered valid if it contains both letters and numbers; if you select **Digit**, the input string is considered valid if it only contains numbers. The way the **Entry** displays the input string (in the example, green when valid, and red when invalid) is determined by the **ValidStyle** and **InvalidStyle** properties, respectively bound to the following styles.

```
<pages:BasePage.Resources>
    <Style x:Key="InvalidEntryStyle"
           TargetType="Entry">
```

```
        <Setter Property="TextColor" Value="Red"/>
    </Style>

    <Style x:Key="ValidEntryStyle"
           TargetType="Entry">
        <Setter Property="TextColor" Value="Green"/>
    </Style>
</pages:BasePage.Resources>
```

In summary, the **CharactersValidationBehavior** is very useful to validate characters in a string, or a string as a whole, but it is not the only behavior available for string validation. In the next section, you will learn about the **MaxLengthReachedBehavior**.

Detecting when the maximum length of a string is reached

The **CharactersValidationBehavior** provides several criteria for character validation, but sometimes you only need to set up the maximum length for a string and make sure the user does not enter a longer one. For this simpler scenario, you can use the **MaxLengthReachedBehavior** class.

In the sample project, this is demonstrated in the Behaviors\MaxLengthReachedBehaviorPage.xaml file. The XAML code for the example is the following.

```
<Entry Placeholder="Start typing until MaxLength is reached..."
       MaxLength="{Binding Path=Text, Source={x:Reference MaxLengthSetting}}"
       Margin="{StaticResource ContentPadding}">
    <Entry.Behaviors>
        <xct:MaxLengthReachedBehavior MaxLengthReached=
             "MaxLengthReachedBehavior_MaxLengthReached"
              ShouldDismissKeyboardAutomatically="{Binding Path=IsToggled,
              Source={x:Reference AutoDismissKeyboardSetting}}" />
    </Entry.Behaviors>
</Entry>
```

In the example, all the relevant properties are data-bound to other visual elements. Here's what you need to know:

- The **MaxLength** property must be set on the **Entry** that needs validation with the maximum number of allowed characters.
- The **ShouldDismissKeyboardAutomatically** property of the behavior, of type **bool**, allows you to decide whether the keyboard should be automatically dismissed once the maximum length has been reached.
- The **MaxLengthReached** event of the behavior is raised when the maximum length has been reached. With the related event handler, you can decide what action to take when this happens. In the sample project, the code moves the focus to the next entry.

You can quickly try yourself using the sample code. This behavior is very useful when the validation requirement is only checking for the maximum number of characters in a string.

Validating numbers

The Xamarin Community Toolkit offers the **`NumericValidationBehavior`** class, which makes it possible to validate a number against its length and decimal places. In the sample project, it is demonstrated in the Behaviors\NumericValidationBehaviorPage.xaml file. Consider Figure 44, where a number is highlighted in red and considered invalid.

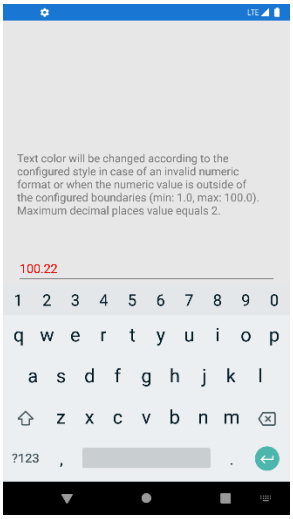


Figure 44: Validating numbers

The following XAML puts in place the number validation.

```
<Entry Placeholder="Number">
    <Entry.Behaviors>
        <xct:NumericValidationBehavior
             InvalidStyle="{StaticResource InvalidEntryStyle}"
             MinimumValue="1.0"
             MaximumValue="100.0"
             MaximumDecimalPlaces="2"/>
    </Entry.Behaviors>
</Entry>
```

The **`MinimumValue`** and **`MaximumValue`** properties allow you to set the boundaries of the number. Outside such boundaries, the number is considered invalid. The **`MaximumDecimalPlaces`** property allows you to specify how many decimal numbers can be accepted after the separator.

Validating uniform resource identifiers (URI)

Uniform resource identifiers (also referred to as URIs) are typically used to represent web addresses. A URI is made up of the protocol (https://) and the address (www.microsoft.com). Validating URIs is another extremely common requirement in any app, and the Xamarin Community Toolkit makes this easy via the **`UriValidationBehavior`** class. All you have to pass to the behavior is the type of URI, which can be **`Absolute`**, **`Relative`**, or **`RelativeOrAbsolute`** (which are the ways the **`System.Uri`** class represents URIs).

In the sample project, this is demonstrated in the Behaviors\UriValidationBehaviorPage.xaml file. The following is the XAML sample code.

```
<Entry Placeholder="UriKind: Absolute">
    <Entry.Behaviors>
        <xct:UriValidationBehavior UriKind="Absolute"
             InvalidStyle="{StaticResource InvalidEntryStyle}"/>
    </Entry.Behaviors>
</Entry>
<Entry Placeholder="UriKind: Relative">
    <Entry.Behaviors>
        <xct:UriValidationBehavior UriKind="Relative"
             InvalidStyle="{StaticResource InvalidEntryStyle}"/>
    </Entry.Behaviors>
</Entry>
<Entry Placeholder="UriKind: RelativeOrAbsolute">
    <Entry.Behaviors>
        <xct:UriValidationBehavior UriKind="RelativeOrAbsolute"
             InvalidStyle="{StaticResource InvalidEntryStyle}"/>
    </Entry.Behaviors>
</Entry>
```

As for the other behaviors, you can specify a style for the invalid state via the **`InvalidStyle`** property, and one for the valid state via the **`ValidStyle`** property. If you run the sample project and check this behavior, you will see how an invalid URI will be highlighted in red.

Validating strings for equality

Sometimes you might need to validate a string and consider it valid only if it equals another string. This is the case when the user specifies a new password and then needs to confirm the password inside another input box, so the validation passes only if the second string matches the first one.

For this particular scenario, you can use the **`RequiredStringValidationBehavior`** class, demonstrated in the Behaviors\RequiredStringValidationBehavior.xaml file. The following XAML code demonstrates how easy it is to use.

```
<Entry Placeholder="Confirm Password">
```

```
    <Entry.Behaviors>
        <xct:RequiredStringValidationBehavior
             InvalidStyle="{StaticResource InvalidEntryStyle}"
             Flags="ValidateOnValueChanging"
             RequiredString="{Binding
             Source={x:Reference passwordEntry},Path=Text}" />
    </Entry.Behaviors>
</Entry>
```

Tip: If you want to use this behavior for password validation, it is good practice to assign the `IsPassword` property of the `Entry` with `true`.

You simply need to bind the **`RequiredString`** property to the **`Text`** property of an **`Entry`**, **`Label`**, **`Editor`**, or to any other string you want to match, and set the **`InvalidStyle`** with the style you want to use for invalid strings. Notice that, in this particular case, the **`Flags`** property has been assigned with **`ValidateOnValueChanging`**, which causes validation to happen at every key stroke. If you run the code, you will see the text highlighted in red if it does not match the text in the first **`Entry`**.

Hints about multiple validation behaviors

It is possible to combine multiple validation behaviors and have them attached to the same view by using the **`MultiValidationBehavior`** class. The following XAML code demonstrates how to implement more complex password validation.

```
<Entry IsPassword="True" Placeholder="Password">
    <Entry.Behaviors>
        <xct:MultiValidationBehavior
             InvalidStyle="{StaticResource InvalidEntryStyle}"  >
            <xct:CharactersValidationBehavior x:Name="digit"
CharacterType="Digit" MinimumCharacterCount="1"
xct:MultiValidationBehavior.Error="1 number" RegexPattern="" />
            <xct:CharactersValidationBehavior x:Name="upper"
CharacterType="UppercaseLetter" MinimumCharacterCount="1"
xct:MultiValidationBehavior.Error="1 upper case" RegexPattern="" />
            <xct:CharactersValidationBehavior x:Name="symbol"
CharacterType="NonAlphanumericSymbol" MinimumCharacterCount="1"
xct:MultiValidationBehavior.Error="1 symbol" RegexPattern=""  />
            <xct:CharactersValidationBehavior x:Name="any"
CharacterType="Any" MinimumCharacterCount="8"
xct:MultiValidationBehavior.Error="8 characters" RegexPattern="" />
        </xct:MultiValidationBehavior>
    </Entry.Behaviors>
</Entry>
```

Note: The original XAML sample code displays validation messages in a different language, so here you see the English counterparts.

The Behaviors\MultiValidationBehaviorPage.xaml file includes several code snippets that target different validation behaviors concurrently.

Handling focus changes

It is not uncommon to perform actions when a view's focus changes. For example, suppose you have a form with multiple input boxes and you want to automatically move the focus to the next **`Entry`** once the user stops typing. Because the **`Entry`** and **`Editor`** expose a **`Completed`** event that is raised when the user stops typing and presses Enter, you can attach the **`SetFocusOnEntryCompletedBehavior`**, which is capable of intercepting such an event, and automatically set the focus on the specified **`Entry`**.

In the sample project, this is demonstrated in the Behaviors\SetFocusOnEntryCompleteBehaviorPage.xaml file. The following code snippet demonstrates how to move to the next **`Entry`** when the user completes typing in the first one.

```
<Entry x:Name="Entry1"
       Placeholder="Entry 1"
       ReturnType="Next"
       xct:SetFocusOnEntryCompletedBehavior.
       NextElement="{x:Reference Entry2}"/>
```

It is very simple to use: you just need to assign the **`NextElement`** property of the behavior using the **`x:Reference`** syntax, passing the name of the **`Entry`** that will receive the focus. Another interesting behavior is called **`UserStoppedTypingBehavior`**, which is demonstrated in the Behaviors\UserStoppedTypingBehaviorPage.xaml file of the sample project. This behavior is useful to assume that the user stopped typing inside an input box, without handling events or implementing custom logic.

Unlike the other examples, for this behavior the sample project works with a **`SearchBar`**, and the XAML code is the following.

```
<SearchBar Placeholder="Start searching..."
           Margin="{StaticResource ContentPadding}">
    <SearchBar.Behaviors>
        <xct:UserStoppedTypingBehavior Command="{Binding SearchCommand}"
             StoppedTypingTimeThreshold="{Binding Path=Text,
             Source={x:Reference TimeThresholdSetting}}"
             MinimumLengthThreshold="{Binding Path=Text,
             Source={x:Reference MinimumLengthThresholdSetting}}"
             ShouldDismissKeyboardAutomatically="{Binding Path=IsToggled,
             Source={x:Reference AutoDismissKeyboardSetting}}" />
    </SearchBar.Behaviors>
```

```
</SearchBar>
```

StoppedTypingTimeThreshold represents how many milliseconds must pass before the behavior can consider the user to have really stopped typing, combined with a minimum number of characters the user must have typed (**MinimumLengthThreshold**). When the combination of these properties is a true condition, the **Command** property invokes the data-bound command to perform an action automatically. If you run the sample project, the bound command displays the text you typed in the search bar after you stop typing, and after the specified number of milliseconds.

Chapter summary

Data validation is not always an option in every kind of application, and mobile apps are no exception. The Xamarin Community Toolkit exposes several reusable behaviors that make it quick and easy to validate strings without implementing custom logic. In fact, you can quickly validate email addresses, URIs, characters in a string, numbers, and equality of strings, and you can even perform multiple validations concurrently.

Data is often related to an important programming pattern such as Model-View-ViewModel (MVVM), and the next chapter discusses how the Xamarin Community Toolkit extends the development possibilities when using MVVM.

Chapter 6 Supporting Advanced Model-View-ViewModel Development

Model-View-ViewModel (MVVM) is a programming pattern that allows for separating data (Model), logic (ViewModel), and user interface (View), and is generally available to development platforms based on XAML, such as Xamarin.Forms, Windows Presentation Foundation (WPF), and Universal Windows Platform (UWP).

One of the biggest benefits of using MVVM is that it is possible to work on the user interface without touching the code for the logic, making it possible for graphic designers to work on the XAML markup using specific tools. Developers can also work on the logic and the data architecture without changing the user interface, so there is also some kind of role separation.

This chapter discusses how the Xamarin Community Toolkit enhances MVVM development, especially for developers who need to create their own framework and cannot use popular libraries.

Note: It is not possible to explain the Model-View-ViewModel pattern in this chapter, so here I assume you have at least a basic knowledge of how it works.

The commanding pattern

Actually, MVVM is based on another pattern called commanding, which relies on exposing properties of type **`ICommand`** from ViewModels rather than handling events in a view's code behind. Commanding is widely used in popular, ready-to-use MVVM frameworks such as Prism and MVVMLight, and it has been discussed in both the documentation and in this blog post by Microsoft.

The main purpose of commanding is simplifying the way events are handled from a logic-separation point of view, and it can certainly be used outside of MVVM, or if you prefer (or need) to create your own MVVM framework.

In order to make the implementation of commanding simpler, it is common practice to implement a very popular behavior called **`EventToCommandBehavior`**, which quickly allows the generating of a command for an event. This behavior is already available inside existing libraries, such as the ones mentioned previously, but you would need to implement it on your own in other cases.

So that you don't have to do this every time for each of your projects, the Xamarin Community Toolkit helps by providing a reusable implementation of the **`EventToCommandBehavior`**, with the addition of other classes and interfaces that will be discussed in this chapter. For your reference, Figure 45 shows how the example looks like in the sample app.

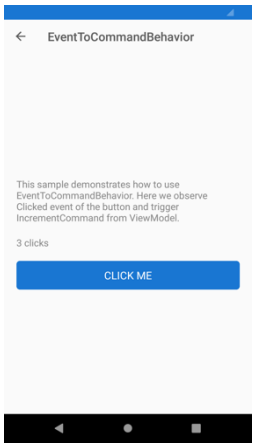


Figure 45: Handling events as commands

Actually, you will not notice any difference in the functionalities because the counter is simply incremented at every mouse click. But behind the scenes, this is leveraging a more data-oriented approach.

Using EventToCommandBehavior to map events to commands

The **`EventToCommandBehavior`** is demonstrated in the Behaviors\EventToCommandBehaviorPage.xaml file of the sample project. Code Listing 9 shows the full XAML code for the page.

Code Listing 9

```
<?xml version="1.0" encoding="UTF-8"?>
<pages:BasePage xmlns="http://xamarin.com/schemas/2014/forms"
                xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
                xmlns:pages="clr-
namespace:Xamarin.CommunityToolkit.Sample.Pages"
                xmlns:vm="clr-
namespace:Xamarin.CommunityToolkit.Sample.ViewModels.Behaviors"

x:Class="Xamarin.CommunityToolkit.Sample.Pages.Behaviors.EventToCommandBeha
viorPage">

    <pages:BasePage.BindingContext>
        <vm:EventToCommandBehaviorViewModel />
    </pages:BasePage.BindingContext>

    <ContentView Padding="{StaticResource ContentPadding}">
        <StackLayout VerticalOptions="CenterAndExpand"
                     Spacing="20">
```

```
            <Label Text="This sample demonstrates how to use
EventToCommandBehavior. Here we observe Clicked event of the button and
trigger IncrementCommand from ViewModel." />
            <Label Text="{Binding ClickCount,
                    StringFormat='{0} clicks'}" />
            <Button Text="Click Me"
                    TextColor="White"
                    BackgroundColor="{StaticResource
                    NormalButtonBackgroundColor}">
                <Button.Behaviors>
                    <xct:EventToCommandBehavior
                        EventName="Clicked"
                        Command="{Binding IncrementCommand}" />
                </Button.Behaviors>
            </Button>
        </StackLayout>
    </ContentView>

</pages:BasePage>
```

As you can see, the **BindingContext** of the page has been assigned with a sample ViewModel called **EventToCommandBehaviorViewModel**. This will be discussed shortly with a focus on the architecture of the behavior. For now, it is important to emphasize that this ViewModel represents the data context of the page, and that it exposes a command called **IncrementCommand**, whose purpose is incrementing a counter.

The key point of the code is in the **Button**. If you look at Code Listing 9, in the **Behaviors** collection, you can see how the **EventToCommandBehavior** class is used to map the **Clicked** event of the button, via the **EventName** property, to a command in the ViewModel. This means that when the **Clicked** event of the button is fired, the behavior redirects the action to the bound command instead of an event handler.

You might argue that the **Button** already has a **Command** property, and that using the behavior is not necessary, which is certainly true, but obviously this has been used to provide the simplest, yet most effective example possible.

In real-world scenarios, it is very common to use this behavior with other views where there is no built-in support for commands. For instance, in a **ListView** you could use the behavior as follows.

```
<ListView>
    <ListView.Behaviors>
        <xct:EventToCommandBehavior
            EventName="SelectionChanged"
            Command="{Binding SelectionChangedCommand}" />
    </ListView.Behaviors>
</ListView>
```

This would allow you to avoid handling an event in a view's code-behind to manage data or logic that is in the ViewModel. At the same time, this approach would make it possible to handle the action in the ViewModel, which is the place where the logic is, favoring layer separation and without violating the principles of the MVVM pattern.

Retrieving event arguments

If you used regular event handlers to intercept user actions instead of using the **EventToCommandBehavior** class, you could also get the event arguments, typically a specialized instance of the **EventArgs** class, that contain information on the current object.

For example, when the user taps an item in the list, the **ItemTapped** event is raised and information on the tapped item is contained in the **ItemTappedEventArgs** object. When the user selects an item in the list, the **ItemSelected** event is raised and information on the selected item is contained in the **ItemSelectedEventArgs** object.

The Xamarin Community Toolkit provides two converters that work in combination with the **EventToCommandBehavior** class and allow for retrieving the appropriate event arguments. They are demonstrated in the Pages\Converters\ItemTappedEventArgsPage.xaml and Pages\Converters\ItemSelectedEventArgsPage.xaml files, and they work very similarly. Both assign their result to the **EventArgsConverter** property of the **EventToCommandBehavior** instance, and they work as follows.

```
<ListView.Behaviors>
    <xct:EventToCommandBehavior EventName="ItemTapped"
         Command="{Binding ItemTappedCommand}"
         EventArgsConverter="{StaticResource
         ItemTappedEventArgsConverter}" />
</ListView.Behaviors>
```

…

```
<ListView.Behaviors>
    <xct:EventToCommandBehavior EventName="ItemSelected"
         Command="{Binding ItemSelectedCommand}"
         EventArgsConverter="{StaticResource
         ItemSelectedEventArgsConverter}" />
</ListView.Behaviors>
```

When you make this assignment, the **EventToCommandBehavior** instance checks if the **EventArgsConverter** property contains an object. If any, the assigned converter is called and the retrieved data can be accessed. An example is in the code-behind file of both sample pages, and the following code shows how this happens for the **ItemTappedEventArgsConverter** inside the **ItemTappedEventArgsViewModel** class.

```
public ICommand ItemTappedCommand { get; } =
      CommandFactory.Create<Person>(person =>
      Application.Current.MainPage.DisplayAlert(
            "Item Tapped: ", person?.Name, "Cancel"));
```

The **ItemTappedCommand** can access the bound instance of the **Person** class because the behavior has retrieved the appropriate **ItemTappedEventArgs** instance. The **ItemSelectedEventArgsConverter** works the same way; only the returned object is different.

Architecture: The CommandFactory class

The support provided by the Xamarin Community Toolkit to MVVM development is not limited to the **EventToCommandBehavior** class. The **Xamarin.CommunityToolkit.ObjectModel** namespace offers additional objects that can be used in a variety of scenarios.

In order to understand more, let's have a look at the ViewModel bound to the page discussed previously, whose code is shown in Code Listing 10. The code file is called EventToCommandBehaviorViewModel.cs, and it is located under ViewModels\Behaviors.

Code Listing 10

```
using System.Windows.Input;
using Xamarin.CommunityToolkit.ObjectModel;

namespace Xamarin.CommunityToolkit.Sample.ViewModels.Behaviors
{
      public class EventToCommandBehaviorViewModel : BaseViewModel
      {
            int clickCount;

            public EventToCommandBehaviorViewModel() =>
                  IncrementCommand =
                  CommandFactory.Create(() => ClickCount++);

            public int ClickCount
            {
                  get => clickCount;
                  set => SetProperty(ref clickCount, value);
            }

            public ICommand IncrementCommand { get; }
      }
}
```

The **IncrementCommand** property, of type **ICommand**, is the command that is bound to the button in the user interface. Different from a common MVVM approach, where the property would be of type **Command** and where this would be instantiated with a new instance of the **Command** object, with the Xamarin Community Toolkit tools, you can create an instance of the command by invoking the **Create** method from the **CommandFactory** class.

In this particular example, the method takes an **Action** as the argument, whose purpose is incrementing the **ClickCount** property by one unit. However, the **Create** method has 16 more overloads. These overloads have in common the possibility to take an **Action** as the first parameter that represents the **Execute** call of a command, and a **Func<bool>** as the second parameter that represents the **CanExecute** value of a command.

Among the method overloads, there are some that allow for creating instances of asynchronous commands, which really makes a difference if compared to the classical commanding pattern implementation.

Implementing asynchronous commands

Though it is possible to execute asynchronous tasks as the target action of **Command** objects, these were not really thought to be asynchronous. With regard to this, the Xamarin Community Toolkit makes another step forward by providing the **IAsyncCommand** interface, which allows for implementing asynchronous commands.

Tip: The next example is not part of the official project, so here you learn something outside of the sample repository.

There are specific overloads of the **CommandFactory.Create** method that allow for instantiating asynchronous commands. For a better understanding, consider the following code, whose purpose is still incrementing the value for the **ClickCount** property, but asynchronously.

```
public IAsyncCommand IncrementCommandAsync { get; }

public EventToCommandBehaviorViewModel()
{
      IncrementCommandAsync = CommandFactory.Create(async () =>
                              await IncrementCounterAsync());
}


public async Task IncrementCounterAsync()
{
      await Task.Run(() => ClickCount++);

}
```

The command is declared as of type **IAsyncCommand**, and an instance is created via an overload of the **CommandFactory.Create** method. The action to be executed is an asynchronous method, invoked via the **await** operator. The benefit is having the possibility of implementing asynchronous commands very quickly.

In addition, following the existing Xamarin.Forms code base, the Xamarin Community Toolkit also exposes the **AsyncCommand** class, which implements the **IAsyncCommand** interface and can be used like you would do with a normal **Command** instance. Here is an example:

```
public EventToCommandBehaviorViewModel()
{
      IncrementCommandAsync =
          new AsyncCommand(() => IncrementAcounterAsync());
}

public IAsyncCommand IncrementCommandAsync { get; }

public async Task IncrementAcounterAsync()
{
      await Task.Run(() => ClickCount++);

}
```

This approach is useful if you want to stay familiar with the syntax normally used with synchronous commands. Remember that the **`AsyncCommand`** class's constructor takes a **`Func<Task>`** object as the first argument, representing the action to be executed by the command, and optionally a second argument of type **`Func<bool>`** that determines whether the command can be executed.

Chapter summary

The Xamarin Community Toolkit enhances your development experience with the MVVM pattern by allowing you to handle events as commands via the **`EventToCommandBehavior`** class. It also makes it easier to define asynchronous commands via the **`IAsyncCommand`** interface and the **`CommandFactory.Create`** method.

The library actually does more: it provides tools to simplify the way applications can be localized, as discussed in the next chapter.

Chapter 7 Localization Made Easy

Localizing applications is a very hot topic, not only in the mobile world. There are many techniques and approaches to implement localization and string translations, and this absolutely depends on your architecture. However, especially if you do not need to add or update translations after your app has been published, it is possible to consider a shared approach.

This is the path drawn by the Xamarin Community Toolkit, which simplifies the way translations are displayed, leveraging new classes and the XAML data-binding feature. This chapter describes the classes offered by the library to make your life easier when you need to bring your app to multiple countries.

Getting started

The Xamarin Community Toolkit makes it easier to localize your mobile applications by exposing objects that point to translated strings, and that are capable of dynamically updating the user interface at runtime. Such objects are discussed in detail in the next section. For now, for a better understanding of the results you can get, focus on the sample app.

When the sample app is running, tap **Settings** at the upper-right corner of the screen. As you can see in Figure 46, you will get sample settings in English, and you will have an option to switch to a different language (Spanish in the example). As soon as you change the language and tap **Save**, the user interface will display strings in Spanish. In the sample app, this happens only in the **Settings** page, but with the objects provided by the Toolkit, this can happen across the whole app at the same time.

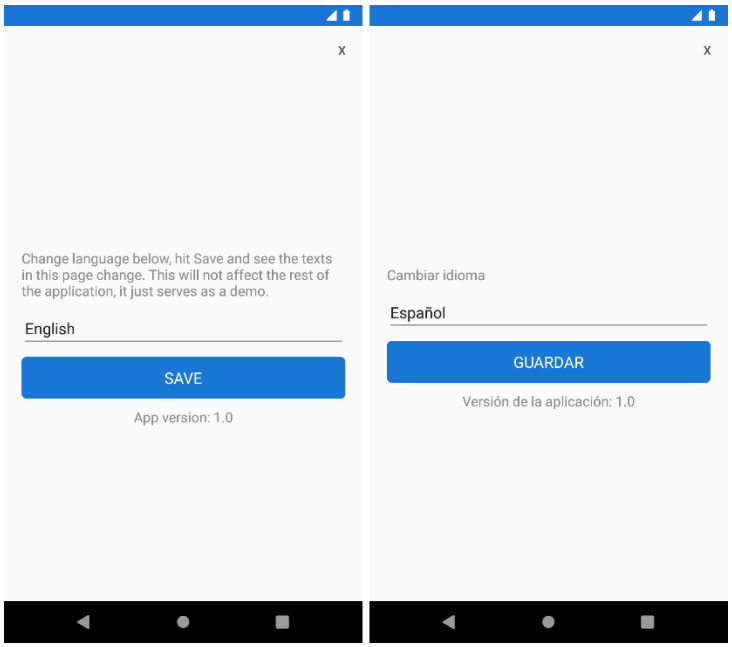


Figure 46: String translation at runtime

The goal of the Xamarin Community Toolkit is twofold: first, it makes it easier to implement localization; second, it makes it possible to quickly update the user interface with new strings at runtime. While the second goal might be optional in your app, since you might want to start with a localized version already instead of asking the user to choose a language, the first goal is certainly common to every app that needs to be translated.

In the next section, you will learn how this happens. The files of the sample project you might want to keep open are the SettingPage.xaml, located under Pages, and the SettingViewModel.cs, located under ViewModels.

Understanding the sample project's structure

As I mentioned in the introduction of this chapter, the way you store and retrieve translated strings in your apps totally depends on your architecture. In more complex architectures, this might involve storing strings in a database and providing an API that returns translations to the callers. In simpler architectures, you can add translated strings to resource (.resx) files, and then you can leverage the Xamarin Community Toolkit to bind them to the user interface.

If you expand the **Resx** folder in the sample project, you will see a file called AppResources.resx. This contains the master strings, which in this case are in English. If you expand this file, you will see two more files: AppResources.en.resx and AppResources.es.resx. These, respectively, contain the English and Spanish strings. Figure 47 shows an example based on the English strings.

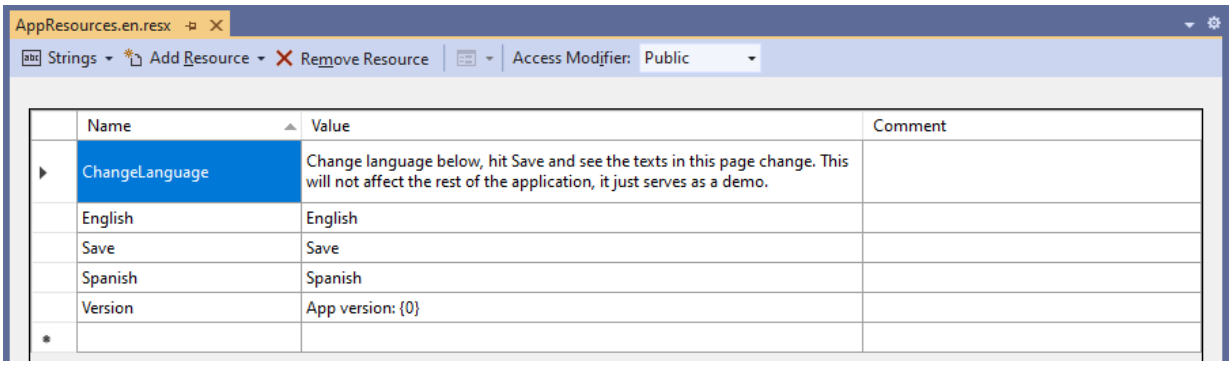


Figure 47: Resource file for English translations

Note: The reason why there are two files with English strings is that one is the master; the second one needs to be there because there should always be a file for each language so that files are independent from the master.

Each language is represented with a read-only **`struct`** called **`Language`**, which is declared as follows.

```
public readonly struct Language
{
	public Language(string name, string ci)
	{
		Name = name;
		CI = ci;
	}

	public string Name { get; }

	public string CI { get; }
}
```

If you want to use a similar translation approach in your apps, you can consider reusing this structure, which simply stores the language name and the shortened ISO code of the culture information for that language. This structure is then used in the **`SettingViewModel`** class, where a list is created to populate the **`Picker`** view that allows for choosing a language in the Settings page that you have seen previously in action.

The list of languages and the currently selected language are represented by the following properties, together with their backing fields.

```
IList<Language> supportedLanguages = Enumerable.Empty<Language>().ToList();

Language selectedLanguage = new Language(AppResources.English, "en");

public Language SelectedLanguage
{
	get => selectedLanguage;
```

```
        set => SetProperty(ref selectedLanguage, value);
}

public IList<Language> SupportedLanguages
{
        get => supportedLanguages;
        private set => SetProperty(ref supportedLanguages, value);

}
```

The way these properties are populated is discussed in the next section about the **LocalizationResourceManager** class. For now, let's see how they are bound to the user interface. In the SettingsPage.xaml file, there is the declaration of the Picker view, which looks like the following.

```
<Picker ItemsSource="{Binding SupportedLanguages}"
          ItemDisplayBinding="{Binding Name}"
          SelectedItem="{Binding SelectedLanguage}"/>
```

The **SupportedLanguages** property populates the list in the **Picker**, and the **SelectedLanguage** property is bound to the **SelectedItem** property of the view. The **Picker** will show the name of the language via the **ItemDisplayBinding**, represented by the **Name** property of each individual instance of the **Language** class in the **SupportedLanguages** list.

The next step is understanding how the user interface can handle translated strings and display them as you change the language. This is a relevant topic and of interest outside of the sample app, so it is discussed in the next, dedicated section.

Introducing the LocalizationResourceManager

The **LocalizationResourceManager** is a class that allows the user interface to automatically update the string it shows at runtime, when changing the culture information, and when selecting a different language. It can be considered a bridge between the translation resources and the user interface, constantly listening to changes in the localization. It is a singleton class, so it exposes a property called **Current** that represents the only instance possible. Before this class is used, it must be set up and initialized in the constructor of the **App** class (App.xaml.cs file) as follows.

```
LocalizationResourceManager.Current.PropertyChanged += (sender, e) =>
AppResources.Culture = LocalizationResourceManager.Current.CurrentCulture;

LocalizationResourceManager.Current.Init(AppResources.ResourceManager);

LocalizationResourceManager.Current.CurrentCulture = new CultureInfo("en");
```

AppResources is a class that Visual Studio generated starting from the name of the resource file added to the project (AppResources.resx in this case). It represents a reference to the resource container that stores the translated strings. The first line of the code in the previous snippet makes sure the culture information of the app resources and the current culture are synchronized, which is important when the user interface needs to automatically update.

The second line of code initializes the class with the resources provided, and the final line assigns a default culture, in this case English. Once the **LocalizationResourceManager** has been initialized, it can be used to manage multiple languages across the app. In the sample project, this is done in the SettingsViewModel.cs file that you started to study previously. If you look back at this file, you will find the **LoadLanguages** method, which is implemented as follows.

```
void LoadLanguages()
{
      SupportedLanguages = new List<Language>()
      {
            { new Language(AppResources.English, "en") },
            { new Language(AppResources.Spanish, "es") }
      };
      SelectedLanguage = SupportedLanguages.
      FirstOrDefault(pro => pro.CI == LocalizationResourceManager.
      Current.CurrentCulture.TwoLetterISOLanguageName);
}
```

This method assigns to the **SelectedLanguage** property the first language in the list that matches the current culture in the **LocalizationResourceManager**. It is invoked in the ViewModel's constructor as follows.

```
public SettingViewModel()
{
      LoadLanguages();

      ChangeLanguageCommand = CommandFactory.Create(() =>
      {
            LocalizationResourceManager.Current.CurrentCulture =
            CultureInfo.GetCultureInfo(SelectedLanguage.CI);
            LoadLanguages();
      });
}
```

The code also creates an instance of a command called **ChangeLanguageCommand**, which is bound to the **Picker** view you saw previously. It changes the current culture in the **LocalizationResourceManager** based on the user selection, then reloads the list of languages.

Note: The `CommandFactory` class is discussed in the next chapter.

The command is simply declared as follows.

```
public ICommand ChangeLanguageCommand { get; }
```

The ViewModel also defines a property called **AppVersion**, of type **LocalizedString**. This type deserves a more thorough explanation, which is given in the next section.

Working with LocalizedString and the Translate extension

The Xamarin Community Toolkit exposes the **LocalizedString**, which basically allows for storing strings and sending a change notification when the value changes. Code Listing 11 shows the definition of the class. (Notice that it is not supported on projects based on .NET Standard 1.0.)

Code Listing 11

```
using System;
using Xamarin.CommunityToolkit.ObjectModel;
using Xamarin.Forms.Internals;

namespace Xamarin.CommunityToolkit.Helpers
{
#if !NETSTANDARD1_0
      public class LocalizedString : ObservableObject
      {
            readonly Func<string> generator;

            public LocalizedString(Func<string> generator)
                  : this(LocalizationResourceManager.Current, generator)
            {
            }

            public LocalizedString(LocalizationResourceManager
                   localizationManager, Func<string> generator)
            {
                  this.generator = generator;

                  // This instance will be unsubscribed and GCed
                  // if no one references it
                  // since LocalizationResourceManager
                  // uses WeekEventManager
                  localizationManager.PropertyChanged +=
                  (sender, e) => OnPropertyChanged(null);
            }

            [Preserve(Conditional = true)]
            public string Localized => generator();

            [Preserve(Conditional = true)]
            public static implicit operator
```

```
            LocalizedString(Func<string> func) =>
            new LocalizedString(func);
      }
#endif
}
```

Other than the property change notification, the other relevant part of the class is the **Localized** property, which returns the localized version of the resource based on the current culture. This is a very important property, as you will shortly see in the views. In the ViewModel, the **LocalizedString** class is used as follows.

```
public LocalizedString AppVersion { get; } = new LocalizedString(() =>
      string.Format(AppResources.Version, AppInfo.VersionString));
```

The **AppVersion** property concatenates a localized string from the app resources, representing the translation of the *version* word, and the app version number retrieved via the **AppInfo** class, offered by the Xamarin Essentials library. Using a property of type **LocalizedString**, instead of a regular **string**, will cause the bound view to be able to update automatically when the language changes.

Tip: An important benefit of the `LocalizedString` class is also making it possible to translate strings at runtime without using a backing .resx resource file. This is demonstrated in the following code snippets.

The **LocalizedString** class usually works in combination with a new markup extension called **Translate**, which is actually used to display localized strings in the user interface. You find the following two examples in the SettingsPage.xaml file.

```
<Label Text="{xct:Translate ChangeLanguage}"/>

...

<Button Text="{xct:Translate Save}"
        Command="{Binding ChangeLanguageCommand, Mode=OneTime}"/>

<Label HorizontalTextAlignment="Center"
       Text="{Binding AppVersion.Localized}"/>
```

Behind the scenes, the **Translate** markup points to the instance of the **LocalizationResourceManager** class from which it retrieves the active instance of the .resx file that contains the translations. In the previous code snippet, you can see how the **Label** and the **Button** have their **Text** properties bound to the strings defined in the app resources via the **Translate** extension. In this way, when the user changes the language at runtime, the **Text** property is automatically updated with the new string, without any additional efforts.

For strings that are not defined inside a resource file, it is possible to bind the **`Text`** property to the **`Localized`** property of the **`LocalizedString`** instance, as is happening in the last **`Label`** of the previous code snippet. This will also cause the view to auto-update when the string value changes based on a different culture.

If you now run the sample app again, following the steps described in the “Getting started” section of this chapter, it will be completely clear how the translation process works, and why the user interface can auto-update at runtime without any further efforts on the development side once the user changes the language.

Chapter summary

Localizing applications is a crucial part of development when you build apps for multiple markets. There are certainly several techniques available, and the final choice depends on your architecture. The Xamarin Community Toolkit offers one that works on the front-end side.

With the **`LocalizationResourceManager`**, you can quickly retrieve translated strings from .resx resource files and with the **`LocalizedString`** class. With the **`Translate`** markup extension, you can use data binding to assign strings to views that support text and that will auto-refresh when the user selects a different language at runtime. This approach requires a new app update every time you want to add a new language, but on the other hand, it is extremely simple and effective.

Chapter 8 Creating the User Interface with C# Markup

In every development platform based on XAML, the declarative markup is not the only way to create the user interface. Everything you do in XAML can be also done in C#. Although this might not be very common, there are situations in which you might need to add views to the visual tree at runtime depending on some conditions.

Building the user interface with C# can be tricky, especially when you need to set up data-bindings, and especially because of the way the C# syntax works. In order to simplify creating the user interface with C#, Microsoft has been working on a new API called C# Markup, which provides a new syntax you can use to define views, their properties, and their interaction behaviors more easily and elegantly. This new API is now offered by the Xamarin Community Toolkit and is the topic of this chapter.

Introducing C# Markup

C# Markup is a new set of fluent APIs that simplify the way developers can create the user interface in C# with the help of new extension methods, and using a syntax based on lambda expressions. You will need to install the Xamarin.CommunityToolkit.Markup NuGet package before using this feature. It is already included in the official sample project, so you do not need any further steps.

It is worth mentioning that before this feature was released to production, it was part of the Xamarin.Forms code base in a preview state. So if you had a chance to try this feature in its early days, you need to remember to upgrade your code by installing the aforementioned NuGet package and removing the preview flags from the App.xaml.cs file.

In the sample app, you can click the **C# Markup** card to access a page completely built with this new feature, shown in Figure 48.

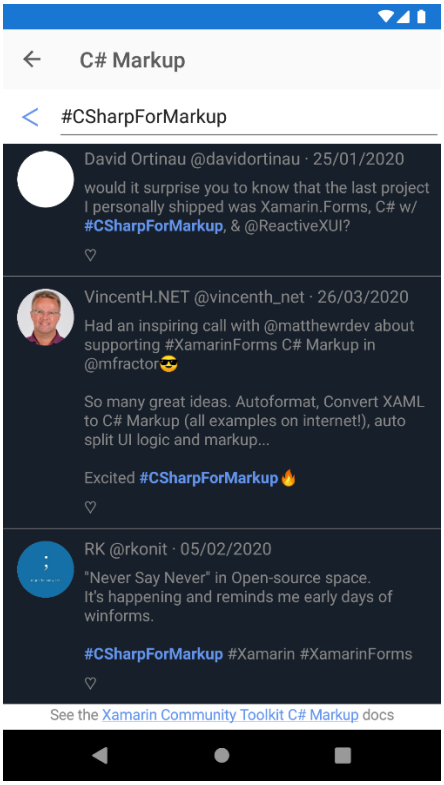


Figure 48: An example of a page created with C# Markup

As you can see, the page represents a Twitter-like user interface, with a list of tweets, each with its own set of visual elements, including images and a search bar. The C# code for the page is in the Pages\Markup\SearchPage.cs file, whereas the logic for the page is in the SearchPage.logic.cs file.

You will also notice a file called Styles.cs, where some styles are also defined in C#, as you will see shortly. Let's now focus on the definition of the user interface with C# Markup.

Working with fluent APIs

> ***Note: This section describes the most relevant and common methods of the Xamarin C# Markup. For a full list and reference, you can read the official documentation.***

The SearchPage.cs file contains the definition of the page and its child views. The first part of the page is the header, which includes a back button and the entry where the user can search. The code is the following.

```
StackLayout Header => new StackLayout { Children = {
    new Button { Text = "\u1438" } .Style (Styles.HeaderButton)
                .Width (50)
                .Bind (nameof(vm.BackCommand)),
```

```
    new Entry { Placeholder = "Search" }
              .FillExpandHorizontal ()
              .Invoke (entry =>
               {
                   entry.Focused   += Search_FocusChanged;
                   entry.Unfocused += Search_FocusChanged;
               })
              .Bind (nameof(vm.SearchText))

}}.Horizontal ();
```

The syntax of C# Markup is a combination of the object initializers feature and new extension methods, together with the shorter syntax of lambda expressions (**=>**) that simplify property getters. If you look at the **Button** declaration, the **Style** extension method makes it possible to assign an object of type **Style** to the current visual element. (Styles used in the example are also defined in C# inside the Styles.cs file and will be discussed shortly.)

Notice how the **Bind** extension method quickly allows for binding a property to another object, in this case a command. If you know how setting up data-binding in C# code normally works, you can immediately understand the benefit of having this method available. In this particular example, the bound property **BackCommand** is defined in the ViewModel and simply allows for going back to the previous navigation state. You can look at its definition yourself, since it's not relevant for the explanation here.

It is also possible to use the **BindTapGesture** method to quickly implement gesture recognizers. This method allows you to specify the target **Command** object that is executed on tap, the source object for the command, and command parameters.

If you now look at the **Entry** definition, you can see how the **FillExpandHorizontal** extension method does with one invocation that you would get assigning the **HorizontalOptions** property with a value of **FillAndExpand**. Then, notice how it is possible to implement interaction via the **Invoke** extension method, which allows for assigning event handlers to the events exposed by the current view instance.

In the end, the **Horizontal** extension method invoked over the **StackLayout** provides horizontal alignment of its child elements, again with a simple method invocation instead of nested property assignments with a more verbose syntax.

The next relevant piece of the user interface definition that contains interesting extension methods is the declaration of the **CollectionView** used to display the list of tweets, and whose code is shown in Code Listing 12.

Code Listing 12

```
enum TweetRow { Separator, Title, Body, Actions }
enum TweetColumn { AuthorImage, Content }

CollectionView SearchResults => new CollectionView { ItemTemplate =
    new DataTemplate(() =>
```

```
        RowDefinitions = Rows.Define (
            (TweetRow.Separator, 2   ),
            (TweetRow.Title    , Auto),
            (TweetRow.Body     , Auto),
            (TweetRow.Actions  , 32  )
        ),

        ColumnDefinitions = Columns.Define (
            (TweetColumn.AuthorImage, 70  ),
            (TweetColumn.Content    , Star)),

        Children = {
            new BoxView { BackgroundColor = Color.Gray }
                        .Row (TweetRow.Separator) .ColumnSpan
                        (All<TweetColumn>()) .Top() .Height (0.5),

            RoundImage ( 53, nameof(Tweet.AuthorImage) )
                       .Row (TweetRow.Title, TweetRow.Actions)
                       .Column (TweetColumn.AuthorImage)
                       .CenterHorizontal () .Top ()
                       .Margins (left: 10, top: 4),

            new Label { LineBreakMode =
                        LineBreakMode.MiddleTruncation } .FontSize (16)
                      .Row (TweetRow.Title)
                      .Column (TweetColumn.Content) .Margins (right: 10)
                      .Bind (nameof(Tweet.Header)),

            new Label { } .FontSize (15)
                      .Row (TweetRow.Body) .Column (TweetColumn.Content
                      .Margins (right: 10)
                      .Bind (Label.FormattedTextProperty,
                             nameof(Tweet.Body),
                      convert: (List<TextFragment> fragments) =>
                      Format(fragments)),

            LikeButton ( )
                        .Row (TweetRow.Actions) .Column
            (TweetColumn.Content) .Left () .Top () .Siz
        }
    })}.Background (Color.FromHex("171F2A"))
       .Bind (nameof(vm.SearchResults));
```

nsion methods in action, most of them related to the ataTemplate of the **CollectionView**. For he **Define** method that simplifies how you ca es an array of objects where each represe n the previous example, the row or column posi r the **TweetRow** and **TweetColumn** enumerations.

The next group of metho v visual elements are positioned within their parent layout. If you consider the vis s defined inside the **Children** property of the **Grid**, you can see that their position is s a the **Row** and **Column** methods. Other methods like **Top**, **Left**, **Right**, **Bottom**, **TopExpand**, **LeftExpand**, **RightExpand**, and **BottomExpand** allow for specifying the element position with or without expansion to fit the available space.

You can also see how the **Margins** method allows you to specify a margin with a verbose syntax (for example, **.Margins(left: 10, top: 4)**) instead of explicitly assigning an object of type **Thickness**. Views that support working with text also expose the **Font** and **FontSize** methods. The first method allows you to assign the font family and font attributes to the view, whereas the second method allows for assigning the font size with the same fluent approach. Some more specialized objects for working with text, like **Span**, expose additional methods to change the text appearance, such as **Bold** and **Italic**.

Working with styles

At the beginning of the discussion about working with the fluent APIs, you saw that views expose the **Style** extension method, which allows for assigning a style. If you look at the Styles.cs file, you will see how to declare styles with the C# Markup syntax. For example, consider the following styles.

```
static Style<Span> link;
static Style<Button> headerButton;

public static Style<Span> Link => link ??= new Style<Span>(
    (Span.FontSizeProperty, 14),
    (Span.TextColorProperty, Color.CornflowerBlue),
    (Span.TextDecorationsProperty, TextDecorations.Underline)
);

public static Style<Button> HeaderButton => headerButton ??= new
Style<Button>(
    (Button.TextColorProperty, Color.CornflowerBlue),
    (Button.FontSizeProperty, 24)

)   .BasedOn (Implicit.Buttons);
```

Thickness

Focus

C# Markup allows for declaring styles by defining properties of type **`Style<T>`**, where **`T`** is the target type for the style. In the previous code snippet, the first style is applied to **`Span`** objects, and it simplifies the way properties are assigned with the desired value. The fluent APIs also allow for using style inheritance via the **`BasedOn`** extension method, which you see in action in the second style, **`HeaderButton`**, that targets the **`Button`** type. The base style is called **`Buttons`**, and is defined in the **`Implicit`** static class located in the same code file, which is declared as follows.

```
public static Style<Button> Buttons => buttons ??= new Style<Button>(
    (Button.BackgroundColorProperty, Color.Transparent)

);
```

In this example, the base style sets a default background color for the button, and then derived styles can add further customizations.

Finding more extension methods

Including the full reference to C# Markup in a book of the *Succinctly* series is not possible; as I mentioned at the beginning of this chapter, you can read the official documentation to get more information. However, as a developer, you can also look into the source code of the Xamarin.CommunityToolkit.Markup library, which is included in the full project repository. File names help identify the target of the extension methods.

For instance, the BindableObjectExtensions.cs file contains extension methods that extend objects of type **`BindableObject`**; VisualElementExtensions.cs contains extension methods that extend objects of type **`VisualElement`**, and so on. This work will not only give you the full list of fluent APIs available, but it will also give you a different point of view in writing C# code with the most modern syntax.

Chapter summary

C# Markup is a new set of flu an create the user interface in C#. New extensio pressions provide a more convenient and elegant way s, as well as implement their interaction behaviors v vailable via the Xamarin.CommunityToolkit.Markup NuGe y you create the UI in C#—especially when you need to add v