

.NET MAUI COMMUNITY TOOLKIT

SUCCINCTLY

BY **ALESSANDRO
DEL SOLE**

.NET MAUI Community Toolkit Succinctly

Alessandro Del Sole

Foreword by Daniel Jebaraj



Copyright © 2023 by Syncfusion, Inc.

2501 Aerial Center Parkway

Suite 111

Morrisville, NC 27560

USA

All rights reserved.

ISBN: 978-1-64200-231-7

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 claim, damages, or any other liability 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: Graham High, senior content producer, Syncfusion, Inc.

Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.

Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

| | |
|--|-----------|
| The <i>Succinctly</i> Series of Books | 7 |
| About the Author | 8 |
| Chapter 1 Introducing the .NET MAUI Community Toolkit..... | 9 |
| What to expect from this book | 9 |
| .NET MAUI Community Toolkit: solving common problems | 10 |
| Using the .NET MAUI Community Toolkit | 10 |
| Setting up the development environment | 11 |
| Conventions used in this book | 14 |
| Chapter summary | 15 |
| Chapter 2 Working with Views | 16 |
| Arranging visual elements with the UniformItemsLayout..... | 16 |
| New views | 17 |
| Displaying profile pictures with the AvatarView | 18 |
| Displaying certified profile pictures with GravatarImageSource | 21 |
| Freehand line drawing with the DrawingView | 24 |
| Displaying popups | 29 |
| Chapter summary | 36 |
| Chapter 3 Improved Notifications with Alerts | 37 |
| Displaying toast notifications | 37 |
| Displaying snackbars..... | 39 |
| Chapter summary | 43 |
| Chapter 4 Saving Time with Reusable Converters | 44 |
| Converting from Boolean to object..... | 44 |
| Converting from byte arrays to images | 46 |

| | |
|---|-----------|
| Converting dates to Coordinated Universal Time..... | 48 |
| Converting decimal numbers to integers..... | 51 |
| Converting enumerations into Boolean values..... | 53 |
| Converting enumerations into integer values..... | 55 |
| Displaying images from resources..... | 57 |
| Converting an index into an array item | 60 |
| Converting from integer to Boolean | 63 |
| Inverting Boolean bindings | 63 |
| Converting objects into Boolean values and comparing for equality | 65 |
| Comparing objects that implement IComparable..... | 66 |
| Checking for string values | 68 |
| Checking for valid strings..... | 70 |
| Checking data collections for null values | 70 |
| Converting a list of strings into one string | 72 |
| Converting one string into a list..... | 72 |
| Changing string casing | 73 |
| Using multiple converters together | 74 |
| Converting multiple Boolean values into one | 75 |
| Converting strings to mathematical expressions..... | 76 |
| Calculating complex expressions..... | 78 |
| Converting colors | 80 |
| Chapter summary..... | 84 |
| Chapter 5 Data Validation with Reusable Behaviors | 85 |
| Common properties of behaviors..... | 85 |
| Validating an email address | 86 |
| Validating characters in a string..... | 89 |

| | |
|---|------------|
| Detecting when the maximum length of a string is reached | 91 |
| Validating numbers..... | 92 |
| Validating Uniform Resource Identifiers (URIs)..... | 94 |
| Validating strings for equality..... | 95 |
| Hints about multiple validation behaviors..... | 95 |
| Handling focus changes | 96 |
| Mapping events to commands..... | 97 |
| Retrieving event arguments | 99 |
| Setting the tint color of an image | 100 |
| Easily defining input masks | 101 |
| Chapter summary..... | 103 |
| Chapter 6 Enhancing Object Power with Extensions..... | 104 |
| Summarizing extensions | 104 |
| Working with colors | 104 |
| Animating colors | 105 |
| Converting colors..... | 107 |
| Working with MAUI services | 111 |
| Chapter summary..... | 112 |
| Chapter 7 Creating the User Interface with C# Markup | 113 |
| Introducing C# Markup | 113 |
| Creating a sample project | 114 |
| Using C# Markup..... | 116 |
| Working with styles | 119 |
| More C# Markup..... | 120 |
| Chapter summary..... | 120 |

The *Succinctly* Series of Books

Daniel Jebaraj
CEO of Syncfusion, Inc.

When we published our first *Succinctly* series book in 2012, *jQuery Succinctly*, our goal was to produce a series of concise technical books targeted at software developers working primarily on the Microsoft platform. We firmly believed then, as we do now, that most topics of interest can be translated into books that are about 100 pages in length.

We have since published over 200 books that have been downloaded millions of times. Reaching more than 2.7 million readers around the world, we have more than 70 authors who now cover a wider range of topics, such as Blazor, machine learning, and big data.

Each author is carefully chosen from a pool of talented experts who share our vision. The book before you and the others in this series are the result of our authors' tireless work. Within these pages, 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.

We are absolutely thrilled with the enthusiastic reception of our books. We believe the *Succinctly* series is the largest library of free technical books being actively published today. Truly exciting!

Our goal is to keep the information free and easily available so that anyone with a computing device and internet access can obtain concise information and benefit from it. The books will always be free. Any updates we publish will also be free.

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 succinctlyseries@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 social media and 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 ebooks 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. 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](#) and support him on [Patreon](#).

Chapter 1 Introducing the .NET MAUI Community Toolkit

It is very common for developers to replicate elements used in one project in another one, especially when working on several .NET MAUI projects. This is true with value converters, custom views, and behaviors. In addition, sometimes developers need to build their own views that are common in modern mobile app design, but that the .NET MAUI code base does not include. To simplify reusing elements across projects, Microsoft offers a new library called the .NET MAUI Community Toolkit. This chapter introduces the library and will explain some conventions that I will be using across the book. If you have worked with Xamarin.Forms in the past and you have used the Xamarin.Forms Community Toolkit, you will find a lot of similarities and the same approach.

What to expect from this book

By reading this book, you will be able to use all the new elements offered by the .NET MAUI Community Toolkit and this will improve your productivity. However, this assumes you already have quite substantial knowledge of .NET MAUI as a development platform and of Microsoft Visual Studio 2022 as the development environment you use to build apps with MAUI. The main reason for this is to keep the length of this ebook brief enough to fit the *Succinctly* series and to provide you with productivity tools in a faster way, so it's not possible to explain how .NET MAUI works. If you are a beginner, I recommend you read my free ebook *.NET MAUI Succinctly (Coming Soon)* 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 elements unless 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 the time of this writing, the latest stable version of the .NET MAUI Community Toolkit is 3.1.0. 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.

.NET MAUI Community Toolkit: solving common problems

The .NET MAUI Community Toolkit is an open-source collection of reusable elements for cross-platform development with .NET MAUI that works with all the supported OSs. It includes views, value converters, behaviors, animations, color themes, and helper classes that you can reuse across projects and that solve common problems without having to manually share your code or reinvent the wheel every time. Technically speaking, it is a .NET MAUI 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 library's source code, but also for the availability of sample code and the product road map.

Using the .NET MAUI Community Toolkit

The .NET MAUI Community Toolkit is available to the general public as a [NuGet package](#). At the time of writing, the latest version available is 3.1.0 and it can be added to your MAUI solution in the usual way via the NuGet Package Manager. Figure 1 shows how the library appears in the NuGet Package Manager user interface of Visual Studio 2022.

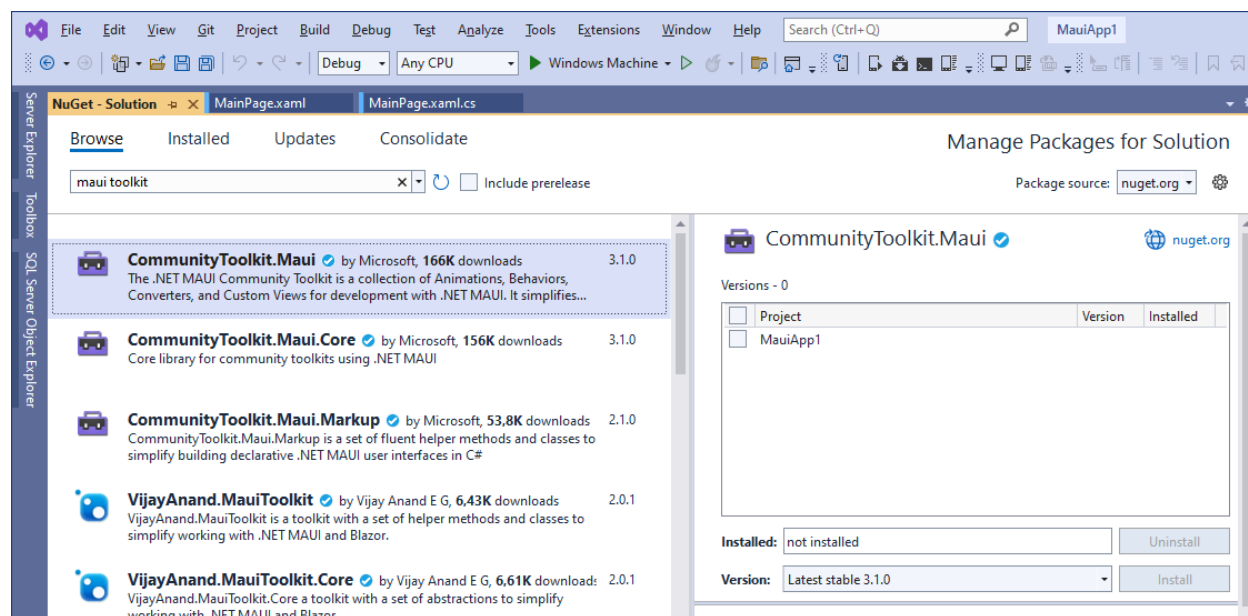


Figure 1: Installing the .NET MAUI Community Toolkit

There is also another NuGet package called `CommunityToolkit.Maui.Markup`, which is only necessary if you want to work with C# markup extensions. This is discussed in [Chapter 7](#). Once you have installed the library, you will be able to use all the objects discussed in this book. The last step is enabling the Community Toolkit in your projects. This is accomplished by changing the invocation of the `MauiApp.CreateBuilder` method in the `MauiProgram.cs` file as follows.

```
var builder = MauiApp.CreateBuilder()  
#if DEBUG  
  
    .UseMauiCommunityToolkit()
```

```

#else

    .UseMauiCommunityToolkit(options =>
    {
        options.SetShouldSuppressExceptionsInConverters(false);
        options.SetShouldSuppressExceptionsInBehaviors(false);
        options.SetShouldSuppressExceptionsInAnimations(false);
    })
#endif

    .UseMauiCommunityToolkitMarkup()
    .UseMauiApp<App>();

```

The **UseMauiCommunityToolkit** extension method enables all the objects in the library, and it disables exceptions for converters, behaviors, and animations when not in debugging mode. **UseMauiCommunityToolkitMarkup** enables C# markup extensions to define the user interface in code. However, it is not necessary to create a new project now. In the next section, I will explain how to use the official sample app.

Setting up the development environment

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

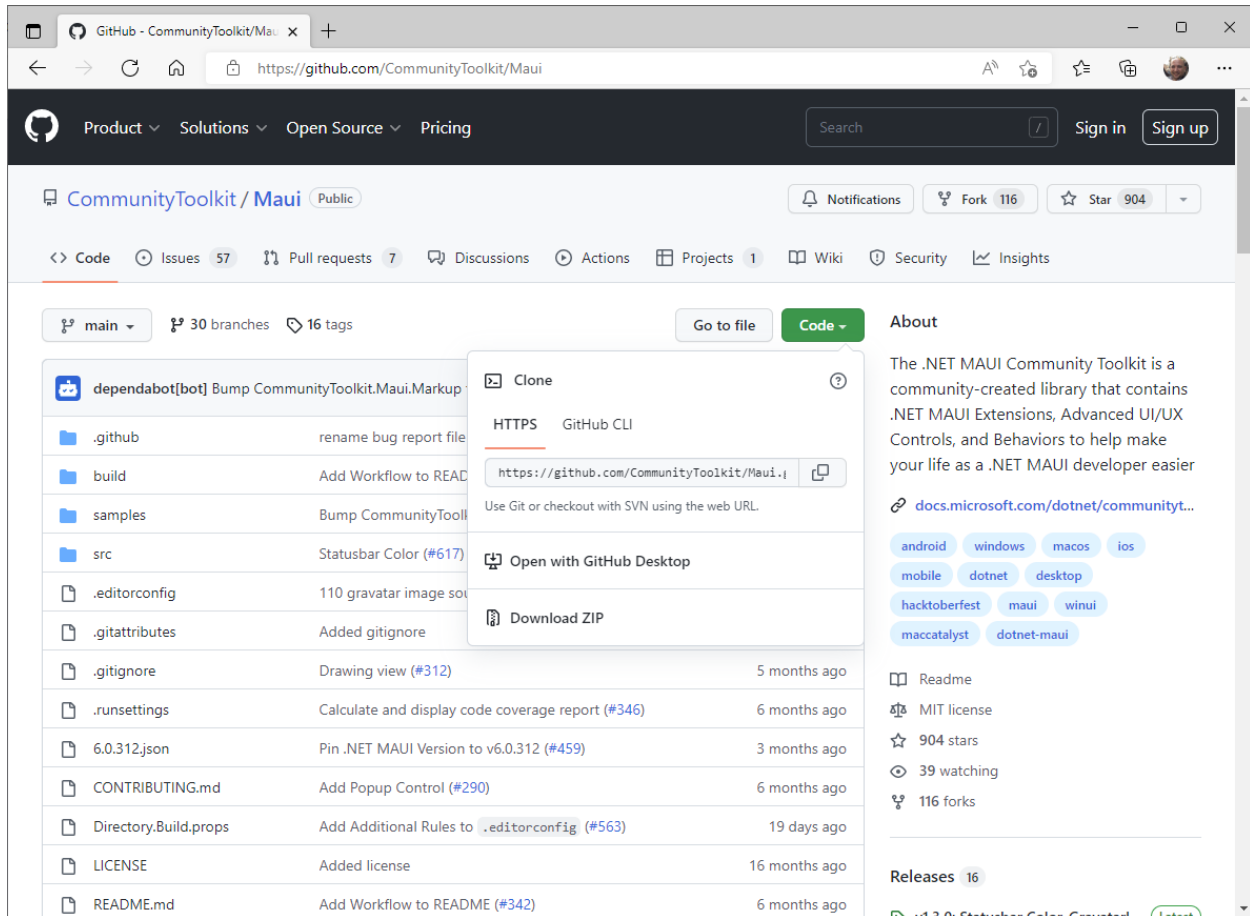


Figure 2: The Official Repository for the .NET MAUI Community Toolkit

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 .NET MAUI Community Toolkit, you can open the `Community.Maui.sln` solution file in Visual Studio. It is located in the `src` subfolder. For the purposes of this book, the solution you need to open is called **CommunityToolkit.Maui.Sample.sln**, which is in the **samples** folder of the repository. Figure 3 shows how the solution appears in Solution Explorer.

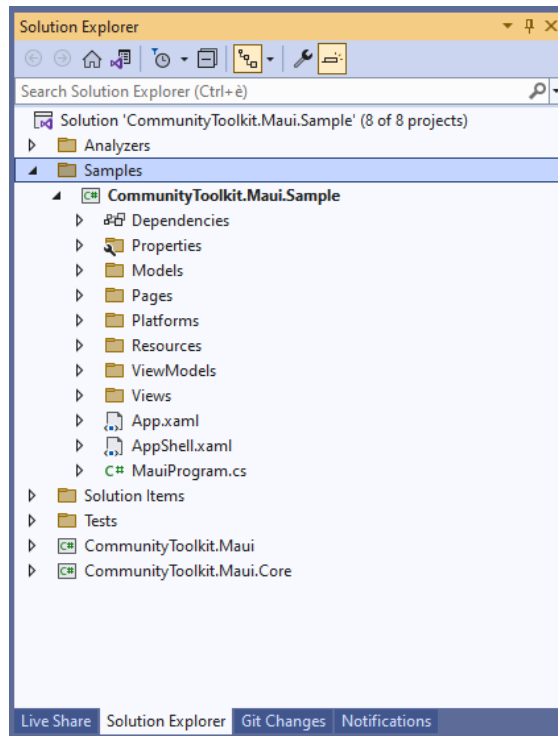


Figure 3: The Official Sample Project in Visual Studio 2022

The project of interest across the book is the shared project called `CommunityToolkit.Maui.Sample` that you can see 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 I told you to download the source code for the full repository instead of only for the sample app is that the `.NET MAUI Community Toolkit` in the sample project is not downloaded from NuGet. Rather, it is referenced via the Visual Studio projects in the repository. If you only downloaded the sample app source code, you would have to manually install the NuGet package and make some changes, wasting your 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. In this book, 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 and the list of examples in the flyout menu. When you click an item, you access a sublist of buttons, and each button opens examples about a specific feature. If you look at Figure 4, starting from left to right, you can see a sample representation of this flow based on the Behaviors feature, but it will work the same for all the features.

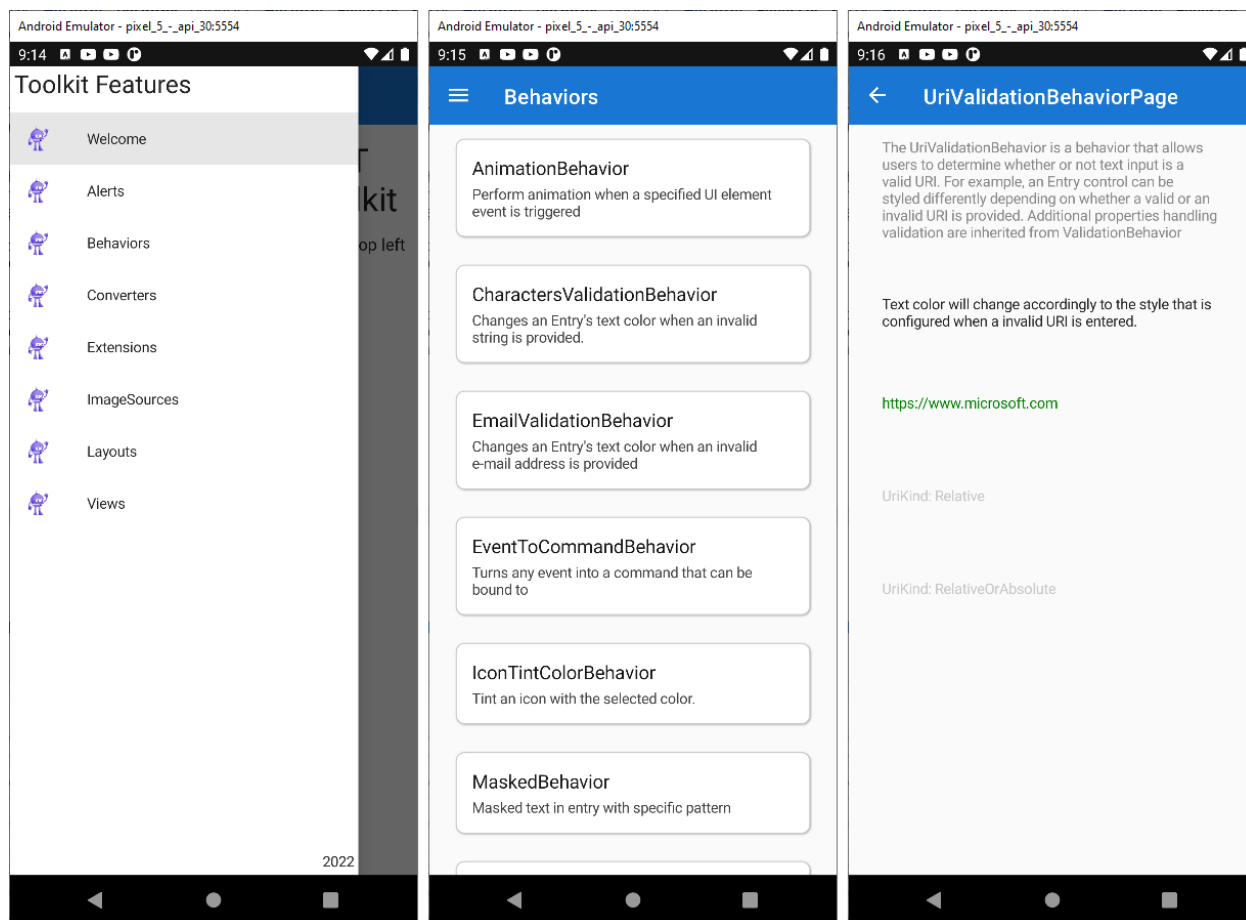


Figure 4: The Official Sample App Running

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

Conventions used in this book

From the next chapter, you will start looking at all the available elements in the .NET MAUI Community Toolkit, so it is useful to provide you with some standards that are shared across the book but 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 also tell you where to find the XAML page files in the project. XAML page files are located in the Pages folder of the 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 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 .NET MAUI Community Toolkit will always include the following namespace declaration.

`xmlns:mct="http://schemas.microsoft.com/dotnet/2022/maui/toolkit"`

In this way, a view like the `UniformItemsLayout` will be accessed as follows.

`<mct:UniformItemsLayout />`

Now that you know to what the `mct` identifier refers, it will not be repeated in the next chapters, nor will the other standards, for the sake of simplicity.

Chapter summary

The .NET MAUI Community Toolkit is an open-source project backed by Microsoft that provides common, reusable elements for .NET MAUI. In this chapter, you received an introduction to the library, you saw how to get the sample source code that will be used in the book, and you saw how to set up the development environment. Then, you learned 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 .NET MAUI Community Toolkit has to offer, and certainly the more natural way to do so is looking at new views.

Chapter 2 Working with Views

The .NET MAUI Community Toolkit includes several views that are commonly used in cross-platform projects but are not available in the MAUI code base, filling the gap with native development. The views also include a new layout, which extends 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.

Arranging visual elements with the `UniformItemsLayout`

The `UniformItemsLayout` is a new layout in the MAUI Community Toolkit that allows creating simplified grids with columns and rows of the same size. The usage, demonstrated in the `Pages\Layouts\UniformItemsLayoutPage.xaml` file, is extremely simple. The default usage is the following.

```
<mct:UniformItemsLayout x:Name="UniformItemsLayout_Default">
    <Button BackgroundColor="Red" />
    <Button BackgroundColor="Green" />
    <Button BackgroundColor="Blue" />
    <Button BackgroundColor="Yellow" />
</mct:UniformItemsLayout>
```

Notice that, unlike the regular `Grid`, it is not possible to specify the `RowDefinitions` and `ColumnDefinitions` collections because the `UniformItemsLayout` automatically arranges its content wrapping and aligning as necessary. You can limit the number of rows and columns by assigning the `MaxRows` and `MaxColumns` properties of type `int` like the following declaration.

```
<mct:UniformItemsLayout x:Name="UniformItemsLayout_MaxRows2MaxColumns2"
    MaxColumns="2" MaxRows="2"/>
```

Other common properties of the `UniformItemsLayout` are `Padding` and `Margin`, both of type `Thickness`. As you would expect, `Padding` represents the space between the content of the layout and the layout borders; `Margin` represents the space between the layout and its parent. Figure 5 shows how the `UniformItemsLayout` appears with different assignments of the `MaxColumns`, `MaxRows`, `Margin`, and `Padding` properties.

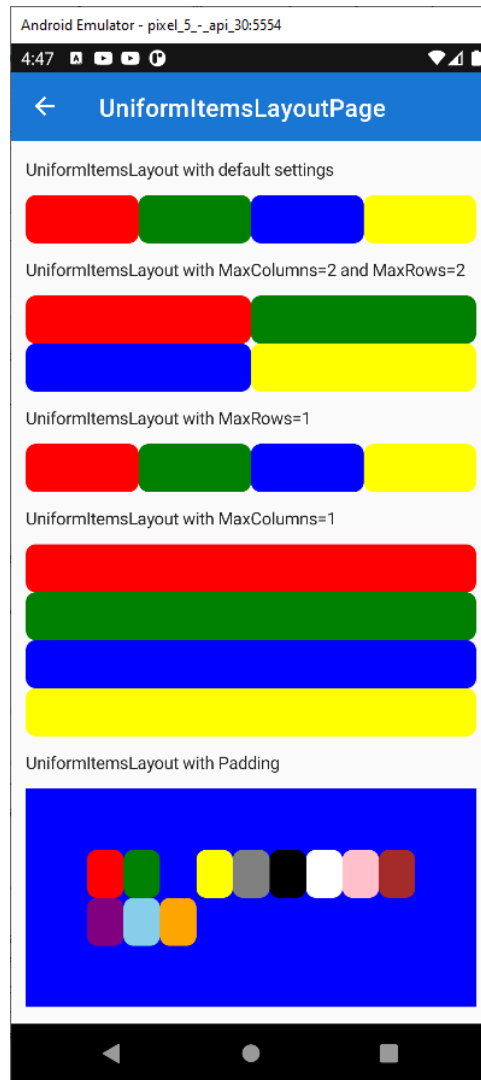


Figure 5: The *UniformItemsLayout* in Action


It's not uncommon to work with grid layouts with rows and columns of the same size, so the **UniformItemsLayout** can save you a lot of time by avoiding the declaration of rows and columns and shortening your code at the same time.

New views

New design standards have debuted in recent years, and consequently new views have entered common mobile app designs. The .NET MAUI Community Toolkit bridges the gap between the MAUI code base and such new designs, introducing views that are now commonly used in mobile apps that will help you be more productive by avoiding time spent creating custom views.

Displaying profile pictures with the AvatarView

An avatar displays the 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 .NET MAUI Community Toolkit makes it easy to use avatars in your apps by offering the **AvatarView** control.

 **Note:** *Displaying the initials of a person's name is a very common use, but it is only one possible use of the AvatarView. You can actually display whatever text you need, as you will discover shortly.*

The sample solution provides 11 different examples located in the Pages\Views\AvatarView subfolder, but for a first understanding of how it works, look at Figure 6. It shows how you can use the AvatarView to display images and text with different shapes and customizations. The code file is Pages\Views\AvatarViewBindablePropertiesPage.xaml.

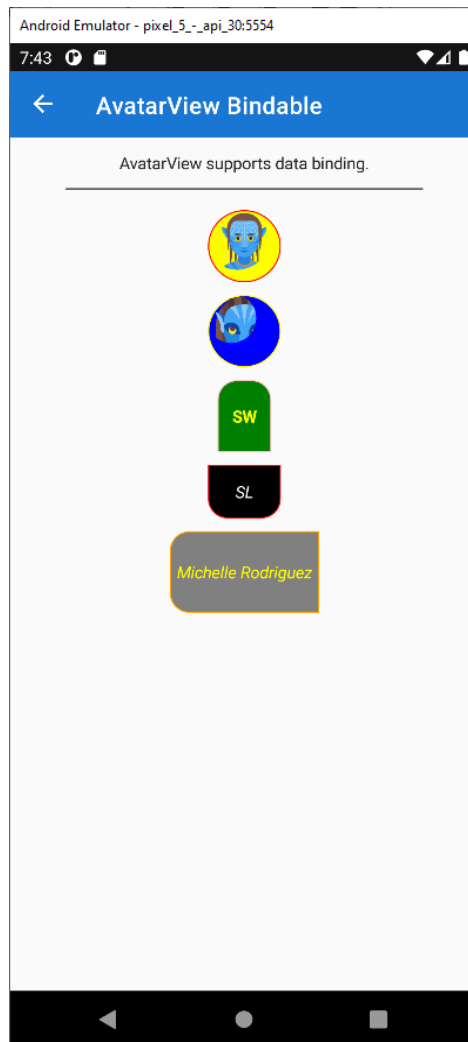


Figure 6: The **AvatarView** in Action

In its simplest form, an **AvatarView** is declared as follows.

```
<mct:AvatarView ImageSource="imagefile.png" Text="FL" />
```

In the previous line, **Text** represents the user initials and **FL** stands for **FirstName LastName**. Obviously, in a real application these will be replaced with relevant text. The **ImageSource** property is assigned with the image you want to display. When the **ImageSource** is not assigned, the value of the **Text** property is displayed.



Tip: The image can also be remote and represented via a *Uri* or in the local and embedded resources.

Table 1 describes the most relevant properties that you will use when working with the **AvatarView**.

Table 1: AvatarView Properties

| Property | Type | Description |
|------------------------|---------------------|---|
| BackgroundColor | Color | The background color of the control. If unset, the background will be the default color (White). |
| BorderColor | Color | The border color of the control. If unset, the border will be the default color (Black). |
| BorderWidth | double | The rendered width of the control border. The default is 1.0. |
| CornerRadius | CornerRadius | The shape of the control. It can be set to a single double uniform corner radius value, or a CornerRadius structure defined by four double values (top left, top right, bottom left, and bottom right). The default is 24 for each corner. |
| ImageSource | ImageSource | The image of the control. If unset, the control will render the Text property. |
| Padding | Thickness | The distance between the control's border and the Text or ImageSource . The default is 1. |
| Text | string | The text for the control, typically the user initials. If unset, the default is '?'. |

| Property | Type | Description |
|------------------|--------------|--------------------------------|
| TextColor | Color | The text color of the control. |

Once you have knowledge of the properties listed in Table 1, understanding how all the examples work will be very easy. They are bindable properties, so they support data binding. For the sake of consistency, a couple more relevant examples are described here. If you look at Figure 7, you can see another sample page that shows how to customize the shape of an **AvatarView**.

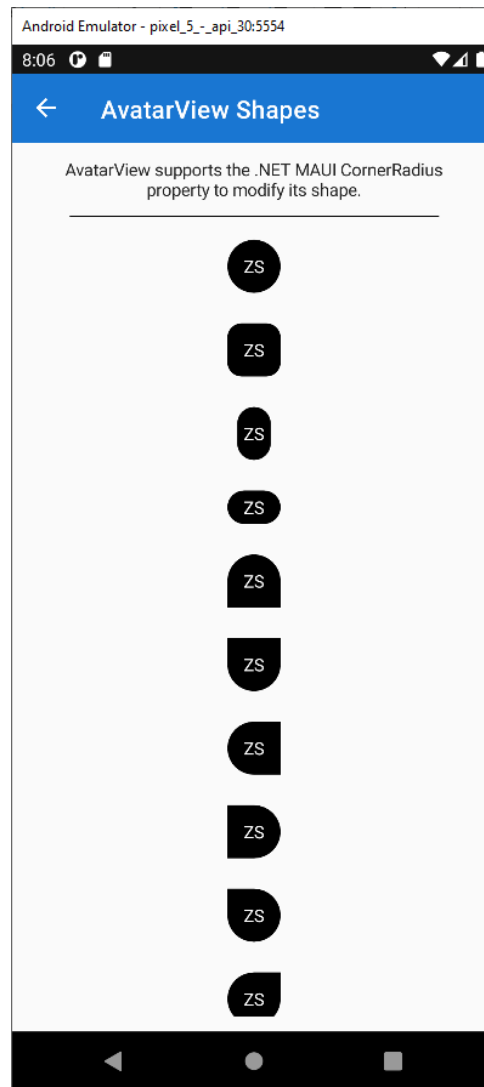


Figure 7: Custom AvatarView Shapes

Customizing the shape is possible by assigning the **CornerRadius** property with specific values for each corner. For example, considering Figure 7, the following assignment produces the sixth shape from the top.

```
<mct:AvatarView CornerRadius="0, 0, 24, 24" Text="ZS" />
```



Tip: In the sample project, the *CornerRadius* is assigned inside *Style* objects. Here the property is assigned to the *AvatarView* directly for clarity.

The following sample code, which is not in the sample solution, generates an **AvatarView** with a yellow background, orange border, dark green text color, some text, and a local image, with a reduced **CornerRadius** to make it appear closer to a square.

```
<mct:AvatarView BackgroundColor="Yellow" BorderColor="Orange"
    BorderWidth="2" Text="AD"
    CornerRadius="12" TextColor="DarkGreen"
    ImageSource="an_image.png">
```

In general, you can control the size of an **AvatarView** via the **WidthRequest** and **HeightRequest** properties in combination with the **CornerRadius** property. Also, **AvatarView** supports gesture recognizers so that you can add user interaction. The following snippet demonstrates how to add a **TapGestureRecognizer** for click support (or tap support on mobile) to an **AvatarView**.

```
<mct:AvatarView.GestureRecognizers>
    <TapGestureRecognizer Tapped="TapGestureRecognizer_Tapped" />
</mct:AvatarView.GestureRecognizers>
```

Then you will need to handle the **Tapped** event in C#. All the other gesture recognizers offered by .NET MAUI can be used. Now that you have knowledge of the way **AvatarView** works and its properties, it will be easier for you to navigate the long list of sample pages about this view.

Displaying certified profile pictures with GravatarImageSource

[Gravatar](#) is an online service for providing globally unique avatars and is widely used to create certified profile pictures. Gravatar is maintained by Automattic, the company that owns WordPress. 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 because you will see shortly in the code how it works. The .NET MAUI Community Toolkit allows for displaying certified profile pictures from Gravatar via the **GravatarImageSource** class, which is demonstrated in the `Pages\ImageSources\GravatarImageSourcePage.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** or to all the other views that expose a property of type **ImageSource**, such as the native **Button** and the **ImageButton**, and even the **AvatarView** discussed in the previous section. A good idea is first having a look at the sample page in action on the device, as shown in Figure 8.

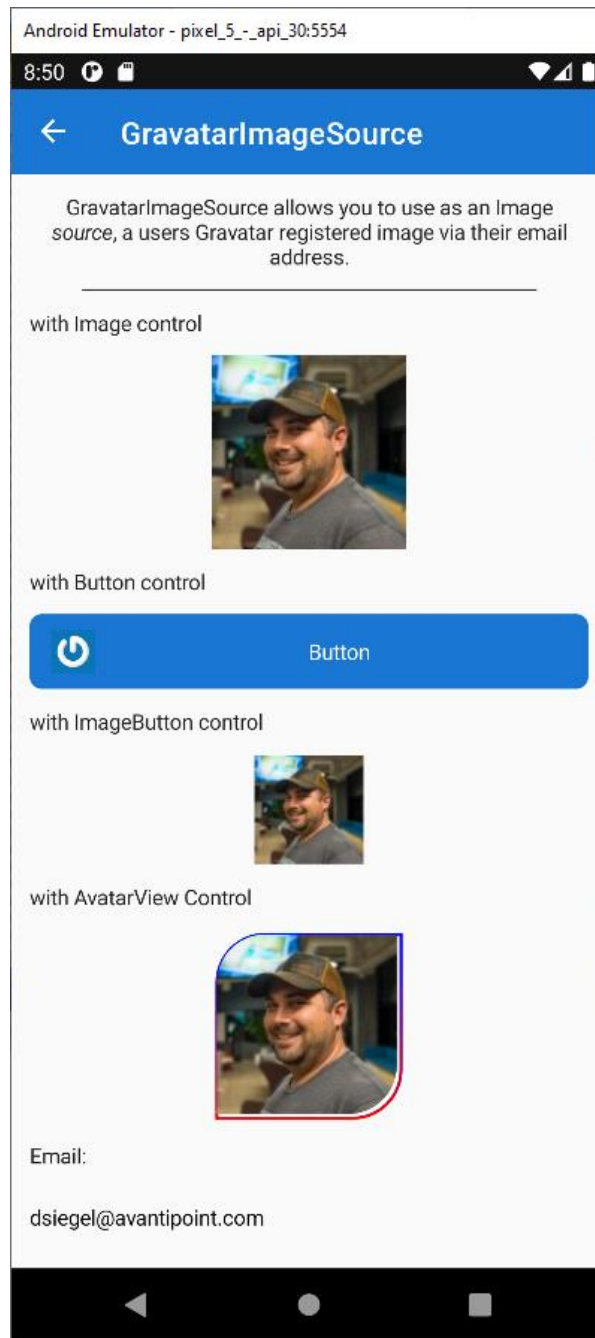



Figure 8: Displaying Profile Pictures from Gravatar

 **Note:** The picture of a real person is used here because the code repository is public and globally available, and it has been intentionally provided by the authors of the .NET MAUI Community Toolkit sample code.

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 several times because it is assigned to different views as specified in the page. You also see a default picture assigned to the **Button**, which you can use in different 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 HeightRequest="128" WidthRequest="128">
    <Image.Source>
        <mct:GravatarImageSource
            CacheValidity="{Binding CacheValidityTimespan}"
            CachingEnabled="{Binding EnableCache}"
            Email="{Binding Email}"
            Image="{Binding DefaultGravatarSelected}"
            SemanticProperties.Description="GravatarImageSource for an Image control,
binding all properties." />
    </Image.Source>
</Image>
```

As you can see, the **GravatarImageSource** object is populating the **Source** property, of type **ImageSource**, 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 to optimize performance. Both the **Size** and **EnableCache** binding properties are exposed by the viewmodel. It is also possible to provide a default picture in case the **GravatarImageSource** is not able to retrieve the proper one. This is possible by assigning the image source with an object of type **DefaultImage**, which is also what you see rendered inside the **Button**. The **DefaultImage** object is an enumeration defined as follows.

```
public enum DefaultImage
{
    MysteryPerson = 0,
    FileNotFound,
    Identicon,
    MonsterId,
    Wavatar,
    Retro,
    Robohash,
    Blank,
}
```

In the **GravatarImageSourceViewModel** class, the **DefaultGravatarSelected** property returns the default image, and it is initialized with the **MysteryPerson** value. However, the user interface of the sample app allows you to select a different one via the following **Picker**.

```
<Picker
    AutomationProperties.LabeledBy="{x:Reference LabelDefaultImage}"
    ItemsSource="{Binding DefaultGravatarItems}"
    SelectedItem="{Binding DefaultGravatarSelected}"
    SemanticProperties.Hint="A default image is displayed if there is no image as
sociated with the requested email hash." />
```

The `Picker.ItemsSource` property is assigned with the following collection of default images.

```
public IReadOnlyList<DefaultImage> DefaultGravatarItems { get; } = new[]
{
    DefaultImage.MysteryPerson,
    DefaultImage.FileNotFound,
    DefaultImage.Identicon,
    DefaultImage.MonsterId,
    DefaultImage.Retro,
    DefaultImage.Robohash,
    DefaultImage.Wavatar,
    DefaultImage.Blank
};
```

Clearly this implementation is for demonstration purposes. In your real-world projects, you will normally use one default image. With the **GravatarImageSource**, you can quickly display a certified picture for a person, and if it's not available, you can warn your users using a default image.

Freehand line drawing with the **DrawingView**

A very useful view introduced by the .NET MAUI Community Toolkit is the **DrawingView**, which provides a surface for drawing lines with the mouse or with touch interaction. There are plenty of scenarios where this view can be useful: certainly, imaging applications that allow for drawing, but also applications where you want to allow users to enter their signature without a mouse, and many others. In the sample solution, usage of the **DrawingView** is demonstrated in the `Pages\Views\DrawingViewPage.xaml` file. Figure 9 shows an example based on text and random lines.

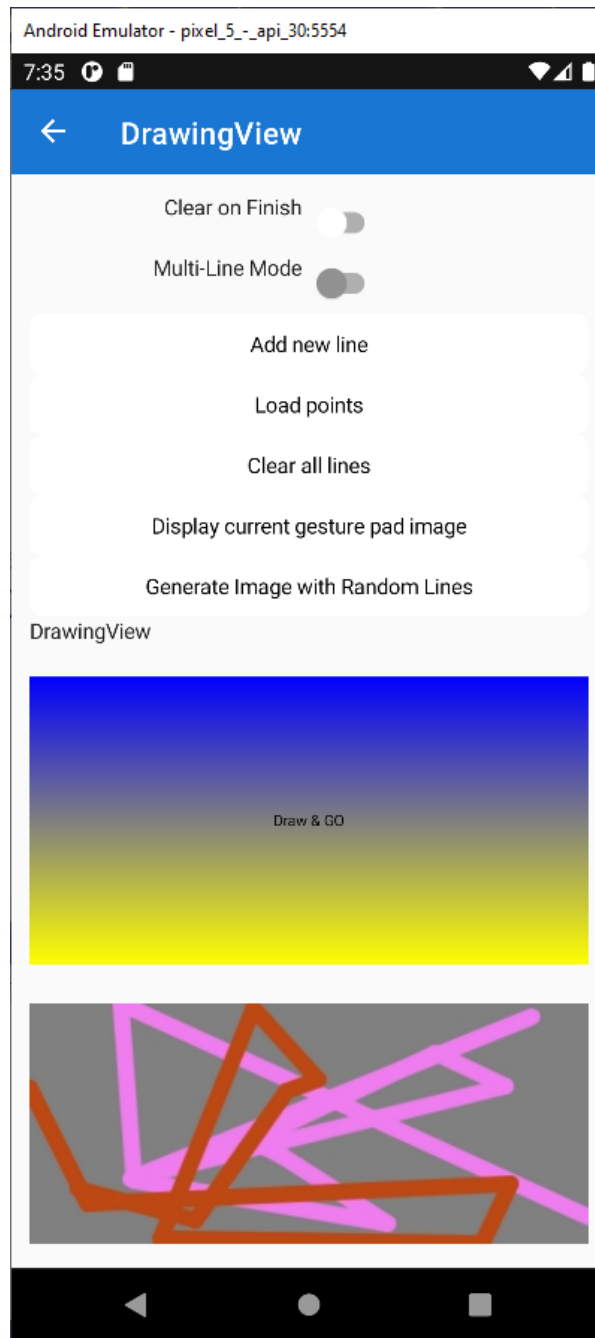


Figure 9: Drawing Lines with the *DrawingView*

In the sample page, there are two drawing surfaces. In the first surface, the app is rendering text. This text is actually drawn with lines, whereas the second surface shows randomly generated lines. In the sample page, the **DrawingView** is declared as follows.

```
<mct:DrawingView x:Name="DrawingViewControl" Margin="0,0,0,10"
    Grid.Row="8" Grid.Column="0" Grid.ColumnSpan="2"
    LineColor="Green" LineWidth="5"
    Lines="{Binding Lines, Mode=TwoWay}"
```

```

DrawingLineCompleted="OnDrawingLineCompleted"
DrawingLineCompletedCommand="{Binding DrawingLineCompletedCommand}"
ShouldClearOnFinish="{Binding Source={x:Reference ClearOnFinish},
Path=IsToggled}"
IsMultiLineModeEnabled="{Binding Source={x:Reference MultiLineMode},
Path=IsToggled}">

```

Some of the properties are data-bound to values from the backing viewmodel, so I've summarized their type and description in Table 2.

Table 2: DrawingView Properties

| Property | Type | Description |
|-------------------------------------|---|---|
| Lines | ObservableCollection<IDrawingLine> | The collection of drawn lines. |
| IsMultiLineModeEnabled | bool | When true, multiple lines can be drawn on the DrawingView when the tap/click is released between drawing lines. If assigned, the DrawingLineCompletedCommand will be invoked after each line that is drawn. |
| ShouldClearOnFinish | bool | Clear the canvas after releasing the tap/click and a line is drawn. |
| DrawingLineCompleted Command | ICommand | Command that is invoked whenever the drawing of a line on the DrawingView has completed. With MultiLineMode enabled, this command is fired multiple times. |
| DrawingLineCompleted | EventHandler<DrawingLine CompletedEventArgs> | An event fired when drawing a line is completed. |
| LineColor | Color | The color of an individual line. |
| LineWidth | float | The width of an individual line. |

Every line is represented by an object of type **DrawingLine**. If the **IsMultiLineModeEnabled** property is **false**, the **Lines** property contains only one **DrawingLine**, otherwise multiple objects. There are a few examples in the project about generating lines. One is represented by the **GenerateLines** method that generates random lines as follows.

```
IEnumerable<DrawingLine> GenerateLines(int count)
{
    var width = GetSide(GestureImage.Width);
    var height = GetSide(GestureImage.Height);
    for (var i = 0; i < count; i++)
    {
        yield return new DrawingLine
        {
            Points = new(DrawingViewViewModel.
                GeneratePoints(10, width, height))
            LineColor = Color.FromRgb(Random.Shared.Next(255),
                Random.Shared.Next(255), Random.Shared.Next(255)),
            LineWidth = 10,
            ShouldSmoothPathWhenDrawn = false,
            Granularity = 5
        };
    }
}
```

This method generates a random number of lines that are then rendered on the **DrawingView** surface. The relevant part of this code is the **DrawingLine** object generation, which highlights the following properties:

- **Points**: a collection of **PointF** objects. The **PointF** class is defined in the **Microsoft.Maui.Graphics** namespace and represents x-, y-, and z-coordinates.
- **LineColor**: an object of type **Color** representing the color of the line.
- **LineWidth**: an object of type **float** representing the line width.
- **ShouldSmoothPathWhenDrawn**: when **true**, enables anti-aliasing when drawing the line.
- **Granularity**: an object of type **int** representing the granularity of the line. The higher the value, the smoother the line, the slower the rendering. The minimum value is 5.

When new lines are generated programmatically, like in the following code, they must be manually added to the **Lines** collection as follows (and as demonstrated in the sample project).

```
DrawingViewControl.Lines.Add(line);
```

When the user draws lines with touch interaction or with the mouse, a collection of **DrawingLine** objects is generated behind the scenes and added to the **Lines** property of the **DrawingView** for rendering. The power of the **DrawingView** is not limited to render lines generated by the user interaction or programmatically. The following sections describe additional useful features.

Retrieving the drawing as an image

The **DrawingView** control displays lines drawn by the user or added in code, but in a lot of situations you will want to create an image from the canvas so that it can be assigned to **Image** views or saved to disk. This is demonstrated in the following event handler of the sample page.

```
async void GetCurrentDrawingViewImageClicked(object sender, EventArgs e)
{
    var stream = await DrawingViewControl.
        GetImageStream(GestureImage.Width, GestureImage.Height);
    GestureImage.Source = ImageSource.FromStream(() => stream);
}
```

The **GetImageStream** method returns a binary stream of type **Stream** that represents the image whose width and height are the same as the target **Image** view. The stream must be converted to an **ImageSource** before it can be assigned to the **Source** property of an **Image** and this is accomplished with the **ImageSource.FromStream** method. Once the image has been assigned, you can work with it as you would any other **Image** view. This includes saving the image to disk for later reuse.



Tip: There are different options to save an image to disk. I personally like the simple way described in this [forum reply on Microsoft Learn](#).

Additional drawing possibilities

You are not limited to drawing lines on the surface of the **DrawingView** control. This exposes the **DrawAction** object, which represents an **Action<ICanvas, RectF>**. The **ICanvas** object is the actual drawing surface of the **DrawingView**. Look at the following code in the constructor of the **DrawingViewPage**.

```
DrawingViewControl.DrawAction = (canvas, rect) =>
{
    canvas.DrawString("Draw & GO", 0, 0, (int)DrawingViewControl.Width,
        (int)DrawingViewControl.Height, HorizontalAlignment.Center,
        VerticalAlignment.Center);
};
```

Objects of type **ICanvas** expose several methods for adding different types of drawings. In the previous code, the **DrawString** method draws the specified string as lines on the drawing surface with specified dimensions. Table 3 summarizes the other methods you can use to draw on the **DrawingView**.

Table 3: *ICanvas Methods*

| Method | Description |
|-----------------|---|
| DrawLine | Draw a line given its X1, Y1, X2, and Y2 coordinates. |

| | |
|-----------------------------|--|
| DrawEllipse | Draw an ellipse given its X1, Y1, X2, and Y2 coordinates. |
| DrawRectangle | Draw a rectangle given its X1, Y1, X2, and Y2 coordinates. |
| DrawRoundedRectangle | Draw a rectangle with rounded corners given its X1, Y1, X2, and Y2 coordinates. |
| DrawArc | Draw an arc given its X , Y , width , height , startAngle , endAngle , clockwise , and closed parameters. |
| DrawImage | Draw an object of type Image given its X , Y , width , and height parameters. |
| DrawString | Draw a string as lines given its X , Y , width , and height parameters, plus optional font type and colors. |
| DrawText | Draw a string with attributes (italic, bold, underline). This requires the Microsoft.Maui.Graphics.Text.Markdig NuGet package, which also allows drawing text from Markdown content. |

As usual, IntelliSense will help you add the appropriate arguments to these methods. For further information, you can read the **ICanvas** [documentation](#).

Displaying popups

The .NET MAUI Community Toolkit exposes the **Popup** view, which allows rendering native popups on each supported platform. Popups can also be customized in several ways. The sample project provides many examples under the Pages\Views\Popup folder.



Note: *The sample project provides many examples on popups, but most of them are built by adding property assignments or views to the popup content. For this reason, I will start discussing popups from the simplest popup possible and then explain only the most relevant features.*

The best way to start is with the simplest popup and then walk through customization possibilities. The Simple Popup example looks like Figure 10.

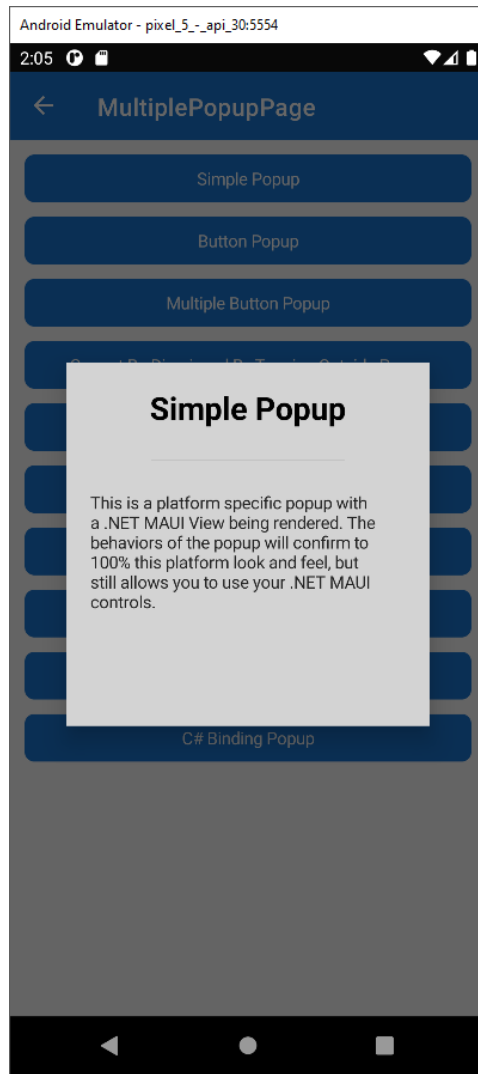


Figure 10: Displaying a Simple Popup

The content of a popup is generally a layout so that you can create a complex visual hierarchy, with a title, text, and other views. The **Popup** class is the starting point for creating and displaying popups. It exposes several public members that allow you to control and customize popups very easily, and these are summarized in Table 4.

Table 4: Most Relevant Popup Members

| Member | Type | Description |
|----------------|---------------------------------|--|
| Content | Property of type View . | The visual hierarchy that represents the popup user interface. |
| Color | Property of type Color . | The background color of a popup. |

| Member | Type | Description |
|--|---|--|
| HorizontalOptions | Property of type LayoutAlignment . | Determine the horizontal alignment for the popup (None , Start , Center , or End). |
| VerticalOptions | Property of type LayoutAlignment . | Determine the vertical alignment for the popup (None , Start , Center , End). |
| Size | Property of type Size . | The popup size expressed in width and height. |
| CanBeDismissedByTappingOutsideOfPopup | Property of type bool . | Determine if the popup can be dismissed by tapping outside of it. |
| Close | Method | Close the popup in C# code. |
| Closed | Event | Fired when the popup is closed. |
| Opened | Event | Fired when the popup is opened. |

In the sample solution, there is a class called **PopupSizeConstants** that defines named size constants assigned to the **Popup.Size** property in the various examples (see Code Listing 1).

Code Listing 1

```
namespace CommunityToolkit.Maui.Sample.Models;

public class PopupSizeConstants
{
    public PopupSizeConstants(IDeviceDisplay deviceDisplay)
    {
        Tiny = new(100, 100);
    }
}
```

```

        Small = new(300, 300);
        Medium = new(0.7 * (deviceDisplay.MainDisplayInfo.Width /
            deviceDisplay.MainDisplayInfo.Density), 0.6 *
            (deviceDisplay.MainDisplayInfo.Height /
            deviceDisplay.MainDisplayInfo.Density));
        Large = new(0.9 * (deviceDisplay.MainDisplayInfo.Width /
            deviceDisplay.MainDisplayInfo.Density), 0.8 *
            (deviceDisplay.MainDisplayInfo.Height /
            deviceDisplay.MainDisplayInfo.Density));
    }

    // examples for fixed sizes.
    public Size Tiny { get; }

    public Size Small { get; }

    public Size Medium { get; }

    public Size Large { get; }
}

```

A popup is declared inside an individual XAML file and is exposed via the **Popup** class. Code Listing 2 shows the full code for the simplest popup.

Code Listing 2

```

<?xml version="1.0" encoding="utf-8" ?>
<mct:Popup
    x:Class="CommunityToolkit.Maui.Sample.SimplePopup"
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:mct="http://schemas.microsoft.com/dotnet/2022/maui/toolkit">
    <VerticalStackLayout Style="{StaticResource PopupLayout}">
        <VerticalStackLayout.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, MacCatalyst=5}" />
    </Style>
</ResourceDictionary>
</VerticalStackLayout.Resources>

<Label Style="{StaticResource Title}" Text="Simple Popup" />
<BoxView Style="{StaticResource Divider}" />
<Label Style="{StaticResource Content}" Text="This is a platform
specific popup with a .NET MAUI View being rendered. The behaviors of the
popup will confirm to 100% this platform look and feel, but still allows
you to use your .NET MAUI controls." />
</VerticalStackLayout>
</mct:Popup>

```

The **VerticalStackLayout** represents the content of the popup, which means you can show whatever you need. The styles defined in the layout resources are applied to the child views and the result you get is shown in Figure 10. In order to display popups, you need to invoke a new extension method called **ShowPopupAsync**, which extends the **Navigation** class and takes the popup instance as an argument. The usage of this method is demonstrated in the `MultiplePopupPage.xaml.cs` file and works like this:

```

async void HandleSimplePopupButtonClicked(object sender, EventArgs e)
{
    var simplePopup = new SimplePopup(popupSizeConstants);
    await this.ShowPopupAsync(simplePopup);
}

```

The **SimplePopup** object is a reference that points to the popup defined in Code Listing 2.



Tip: All the other examples work similarly: a popup visual hierarchy is defined in XAML, an instance of the popup is declared in C#, and *ShowPopupAsync* is invoked.

Optionally, **ShowPopupAsync** can return a result in the form of an **object** if the popup is enabled to do so. By default, popups can be dismissed by tapping outside of them (a 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 **CanBeDismissedByTappingOutsideOfPopup** property in the XAML code as follows.

CanBeDismissedByTappingOutsideOfPopup="False"

You can implement actions inside popups. This can be quickly accomplished by adding **Button** views to the content of the popup. Two examples, called **Button Popup** and **Multiple Button Popup** respectively, demonstrate this. The following XAML from the second example (**MultipleButtonPopup.xaml** file) shows how to implement multiple buttons.

```
<HorizontalStackLayout Style="{StaticResource ButtonGroup}">
    <Button
        Clicked="Cancel_Clicked"
        Style="{StaticResource CancelButton}"
        Text="Cancel" />
    <Button Clicked="Okay_Clicked" Text="OKAY" />
</HorizontalStackLayout>
```

You can simply handle the **Clicked** event or bind buttons to commands to execute actions. Figure 11 shows how the popup with buttons from the sample project appears.

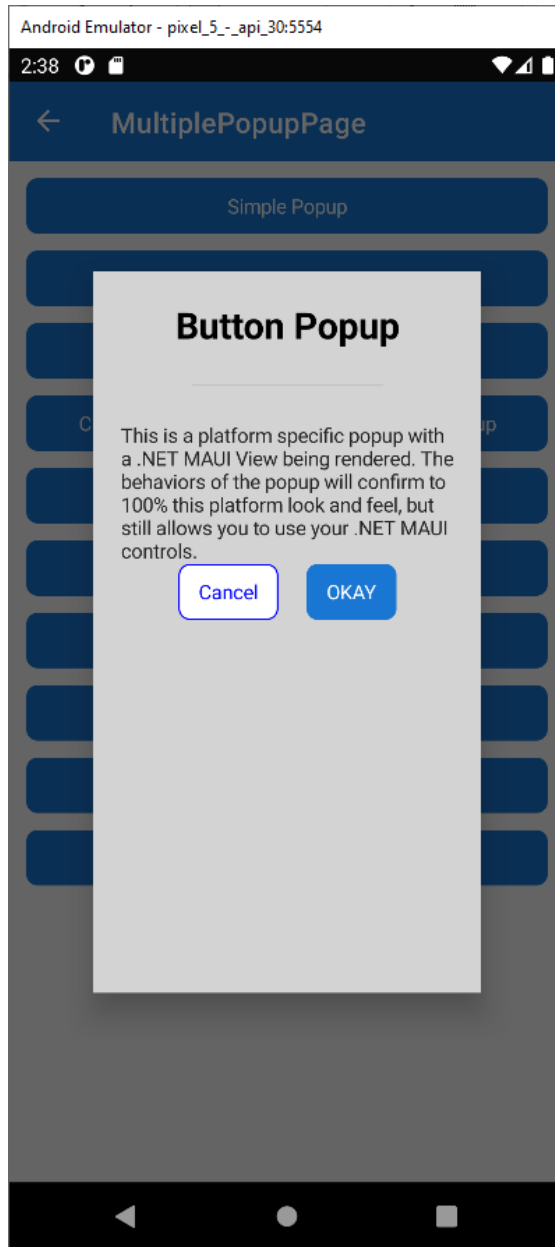



Figure 11: Implementing Actions Inside Popups

 **Tip:** [Table 4](#) listed the *Opened* and *Closed* events. They can be subscribed like any other regular event in C# and handled consequently. The *OpenedEventSimplePopup.xaml* page provides an example.

Data binding and change notifications

The properties listed in Table 4 support data binding, so you can assign popup properties via viewmodels that are also populated at runtime. As a consequence, these properties also implement change notification. For example, if you reassign the **Size** property of the popup, this will be resized accordingly. The `ToggleSizePopup.xaml.cs` file demonstrates this. Finally, you can anchor a popup to another view by simply assigning the **Popup.View** property with the instance of the target view.

Chapter summary

Native popups can be displayed very quickly and without the need to build 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 popup content. This is all possible because of the **Popup** class exposed by the .NET MAUI Community Toolkit, which covers a very common requirement in most applications and is not part of the base MAUI codebase.

Chapter 3 Improved Notifications with Alerts

The .NET MAUI Community Toolkit offers great shortcuts to quickly display toast notifications and snackbars. From a layout perspective, they are similar because they are both represented by a box with some information inside that typically auto dismisses after a certain amount of time. However, toast notifications are normally used to display system messages. Snackbars, instead, typically show a message related to the latest operation done in the app and they should contain a single-line message and no icon. This chapter describes both notification types, using the official sample project as the base. All the objects described in this chapter belong to the `CommunityToolkit.Maui.Alerts` namespace.

Displaying toast notifications

A toast notification is represented by the `Toast` class. It exposes a static method called `Make`, which returns an instance of the `Toast` class and displays the notification via the `Show` method. The sample project first demonstrates how to create a default notification as follows.

```
var toast = Toast.Make("This is a default Toast.");  
await toast.Show();
```

In this case, you simply pass the text of the notification as an argument of the `Make` method. Then, the sample code provides an example of a custom notification as follows.

```
var toast = Toast.Make("This is a big Toast.", ToastDuration.Long, 30d);  
await toast.Show();
```

The `Make` method takes a value of the `ToastDuration` enumeration as its second argument, between `Short` and `Long`. `Short` sets 2 seconds as the notification duration, whereas `Long` sets 3.5 seconds. The third argument of the `Make` method is a `double`, indicated by the `d` suffix, which represents the font size for the notification text. Figure 12 shows what the long toast notification looks like.

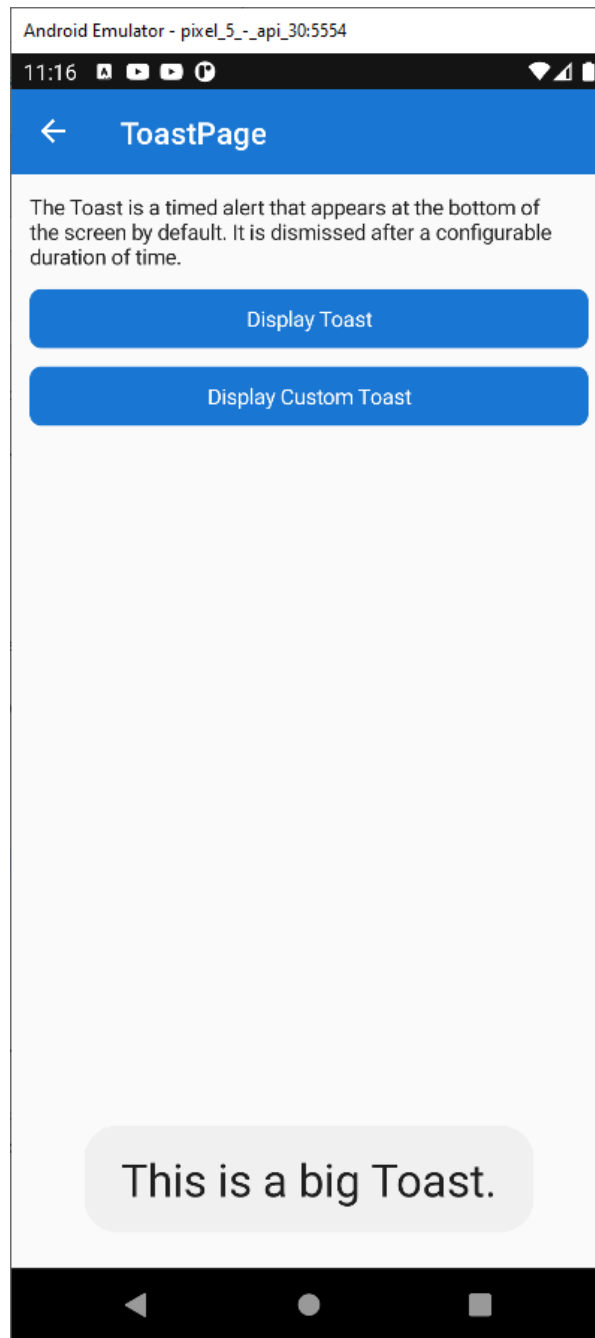


Figure 12: Displaying a Toast Notification

As you can see, the toast notification does not allow performing an additional action. That is a feature that belongs to the snackbar.

Displaying snackbars

Snackbars typically notify the user of an application event, and they also allow for user interaction. In the sample project, you can look at the `Pages\Alerts\SnackbarPage.xaml` file and its code-behind. There are two ways of displaying snackbars:

- Invoking a default snackbar via the **DisplaySnackBar** extension method that is added by the Toolkit to any object that implements the **IView** interface, such as pages.
- Getting an instance of the **Snackbar** class via its static **Make** method, customizing the notification appearance, and then invoking the **Show** instance method.

To display a default snackbar, you can use code like the following.

```
await this.DisplaySnackBar("This is a Snackbar.\nIt will disappear in 3 seconds.\nOr click OK to dismiss immediately.");
```

The **this** qualifier is referring to a page object in this case. You basically pass the text of the notification to the **DisplaySnackBar** method, and the notification will disappear after 3 seconds. Notice how you can use multiple lines by adding the `\n` escape sequence. Figure 13 shows how the default snackbar appears.

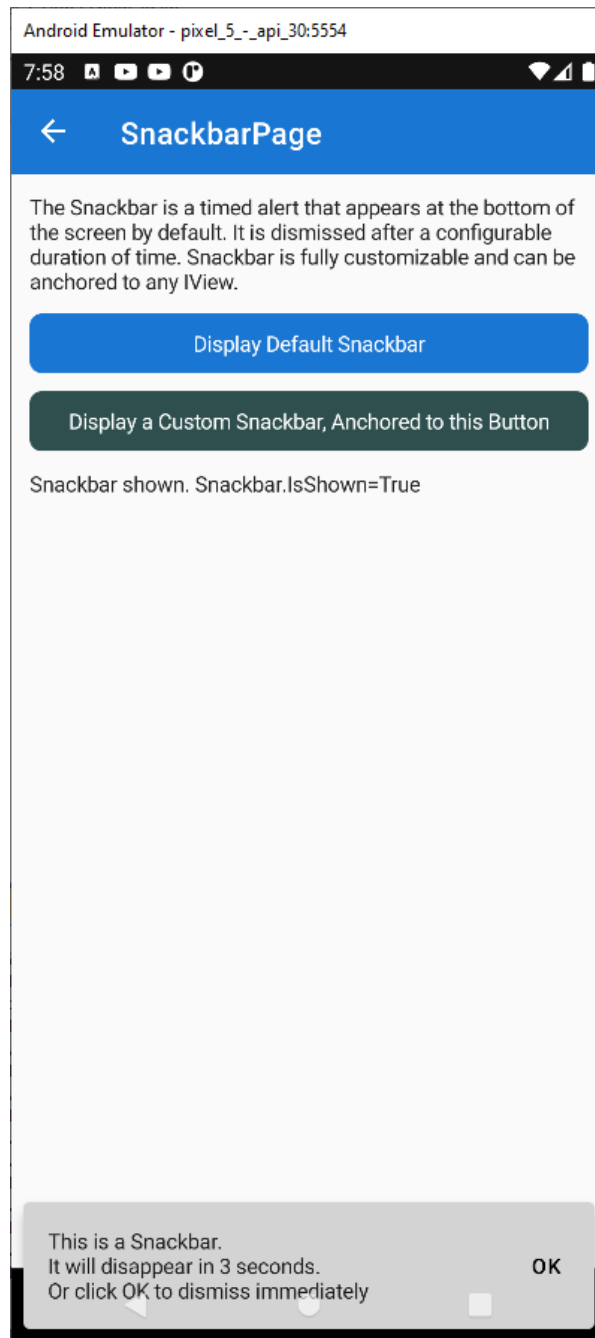


Figure 13: Displaying a Default Snackbar

You can further customize snackbars by adding a user action and by changing the appearance of the notification box. This can be accomplished by first creating an instance of the **SnackbarOptions** class. The sample project does this via the following code.

```
var options = new SnackbarOptions
{
    BackgroundColor = Colors.Red,
    TextColor = Colors.Green,
```



```

        ActionButtonTextColor = Colors.Yellow,
        CornerRadius = new CornerRadius(10),
        Font = Font.SystemFontOfSize(14),
    };

```

In summary, you can change the background and text colors in the snackbar, and you can also assign a different color to the text in the action button. Just so you know, another property called **ActionButtonFont** is available that allows for setting a different font to the action button, but it is not used in the example. Then, you can customize the corner radius of the snackbar via an object of type **CornerRadius**, which can take one **double** as a parameter to set a uniform radius for all the corners, or four **double** values to specify a different radius for each corner. Finally, you can assign a global font for the notification content. The next step is getting an instance of the **Snackbar** class. The sample project provides the following code.

```

customSnackbar = Snackbar.Make(
    "This is a customized Snackbar",
    async () =>
    {
        await
        DisplayCustomSnackbarButton.BackgroundColorTo(colors[Random.Shared.Next(colors.Count)], length: 500);
        DisplayCustomSnackbarButton.Text = displayCustomSnackbarText;
    },
    "Change Button Color",
    TimeSpan.FromSeconds(30),
    options,
    DisplayCustomSnackbarButton);

```



Note: *customSnackbar is a variable of type ISnackbar and is declared at the beginning of the code file.*

The **Make** method takes some arguments, though actually only the first one, representing the notification content that maps to the **Text** property, is mandatory. All the other arguments are optional. After the notification text, you specify the target action for the snackbar button. In this particular example, the action (of type **Action**) changes the background color of a button in the page. The third argument is a string that you can supply to customize the action button text and that maps to the **ActionButtonText** property; the default value is **Ok**. The fourth argument is the notification duration that maps to the **Duration** property and is an object of type **TimeSpan**. In this case, the **TimeSpan.FromSeconds(30)** invocation makes the snackbar visible for 30 seconds. The fifth argument is the instance of the **SnackbarOptions** class created previously. The sixth and last argument allows you to anchor the snackbar to the specified view and maps to the **Anchor** property, of type **IView** of the **Snackbar** class instance. Figure 14 shows how the current snackbar looks.

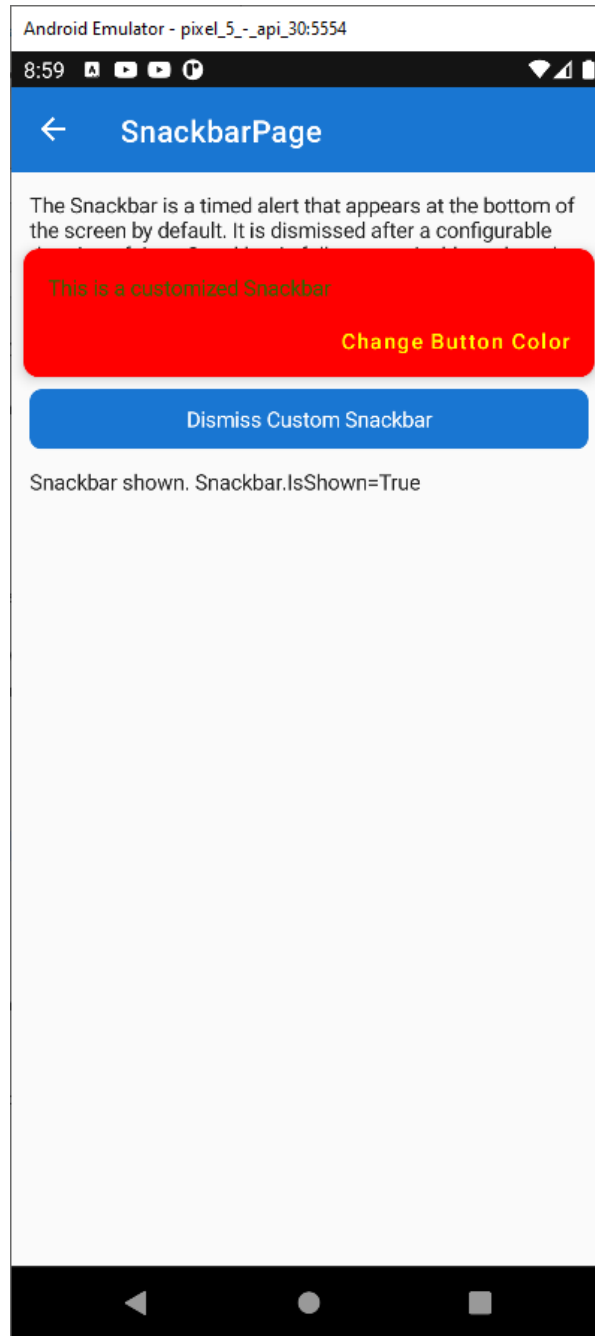


Figure 14: Displaying a Customized Snackbar

Snackbars are a very popular and convenient way to discretely notify the user of an application event, especially in mobile apps, so it is an extremely useful addition to the MAUI Community Toolkit.

Chapter summary

In-app notifications are important ways to inform the user about system and application events while maintaining the user experience. With the .NET MAUI Community Toolkit, you can leverage the most common and popular notification objects: toast notifications and snackbars. Toast notifications are represented by the **Toast** class and notify the user about an event, with no user interaction and limited customization options. Snackbars are represented by the **Snackbar** class and allow for notifying the user about an application event, providing user interaction opportunities, and giving developers more customization options. Alerts fit into the larger area of view enhancements, which also includes an extended collection of data converters, described in the next chapter.

Chapter 4 Saving Time with Reusable Converters

In .NET MAUI 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 with the other chapters, the assumption here is that you already know how value converters work. If not, I recommend you read the [official documentation](#) before continuing. The .NET MAUI 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 Pages\Converters folder of the CommunityToolkit.Maui.Sample project.



Note: *To be consistent with the other ebooks in 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 it needs without going into the details of the architecture. Also, converter classes are very easy to understand, so you can always look at their definition by analyzing the .shared.cs files located under the Converters folder of the CommunityToolkit.Maui project. In addition, note that the `ItemTappedEventArgsConverter` and `SelectedItemEventArgsConverter` will be discussed in Chapter 5 because they work in combination with the `EventToCommandBehavior` class, which is discussed with behaviors.*

Converting from Boolean 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 .NET MAUI 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 15, where you can see that the color of an ellipse changes via data binding depending on the true or false value set via the checkbox (the two buttons change the color by assigning properties directly, instead of using data binding).

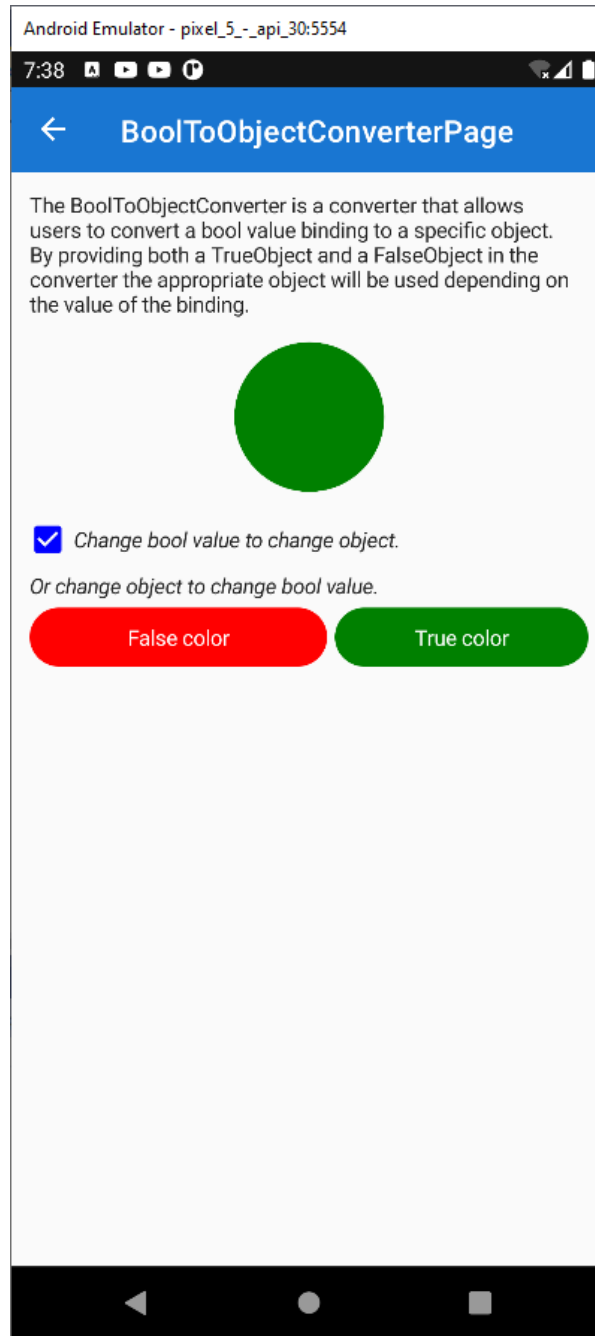



Figure 15: Binding and Converting a Value of Type `bool` to `Color`

 **Tip:** If you do not see the checkbox, in the source code assign the `Color` property of the `CheckBox` view with an explicit color, such as `Blue`.

In this example, the **Ellipse** color is a bound value coming from the **CheckBox** view and is converted into a **Color**. However, conversion is possible to any other type for maximum flexibility. Let's see how this works in code. The **Ellipse** that displays the color is declared as follows.

```
<Ellipse x:Name="Ellipse"
    Grid.Row="1"
    Grid.ColumnSpan="3"
    Margin="10"
    HorizontalOptions="Center"
    HeightRequest="100"
    WidthRequest="100"
    Fill="{Binding Source={x:Reference CheckBox},
    Path=IsChecked, Mode=TwoWay,
    Converter={StaticResource BoolToColorBrushConverter}}"/>
```

Notice how the **Fill** property is bound to the **IsChecked** property of the **CheckBox** and how the **BoolToObject** converter is referenced via a static resource called **BoolToColorBrushConverter**. The converter declaration in the page resources looks like the following.

```
<mct:BoolToObjectConverter x:Key="BoolToColorBrushConverter"
    TrueObject="{StaticResource TrueColorBrush}"
    FalseObject="{StaticResource FalseColorBrush}"/>
```

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. The way they are stored in a data source is typically a Base64 string or a byte array if the image is retrieved by a **Stream** object. The **Image** view in .NET MAUI cannot display a byte array as an image directly, so a converter is required. This is something developers do in almost every project. The .NET MAUI Community Toolkit simplifies your code base by offering the **ByteArrayToImageSourceConverter**, which is demonstrated in `Pages\Converters\ByteArrayToImageSourceConverterPage.xaml`. In this page, you can see that the converter is declared in the simplest way possible.

```
<pages:BasePage.Resources>
    <ResourceDictionary>
        <mct:ByteArrayToImageSourceConverter
            x:Key="ByteArrayToImageSourceConverter"/>
    </ResourceDictionary>
</pages:BasePage.Resources>
```

It can be used when binding a property of type **ImageSource**, such as the **Source** property of the **Image** view or of the **AvatarView**. The current example is based on the **Image** as follows.

```
<Image HorizontalOptions="Center"
    HeightRequest="75"
    Source="{Binding DotNetBotImageByteArray, Mode=OneWay,
    Converter={StaticResource ByteArrayToImageSourceConverter}}"/>
```

The purpose of the sample code is to display an image downloaded from the internet as a byte array, which is converted into an **ImageSource** object.

In the sample project, the **ByteArrayToImageSourceConverterViewModel** class (which is the data source of the page) exposes a property called **DotNetBotImageByteArray** of type **byte?[]**. This property will contain the downloaded image, which is retrieved from the GitHub page of .NET MAUI. The viewmodel contains all the code that downloads the image and assigns it to the property, but the relevant part is the following snippet from the **DownloadDotNetBotImage** method (see line 53 of the **ByteArrayToImageSourceConverterViewModel.cs** file).

```
DotNetBotImageByteArray =  
await client.GetByteArrayAsync("https://user-  
images.githubusercontent.com/13558917/137551073-ac8958bf-83e3-4ae3-8623-  
4db6dce49d02.png",  
cancellationTokenSource.Token).ConfigureAwait(false);
```

The **DotNetBotImageByteArray** 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 **Image**. If you run the app, you will see how the image will appear after the few seconds required to download it from GitHub, as shown in Figure 16.

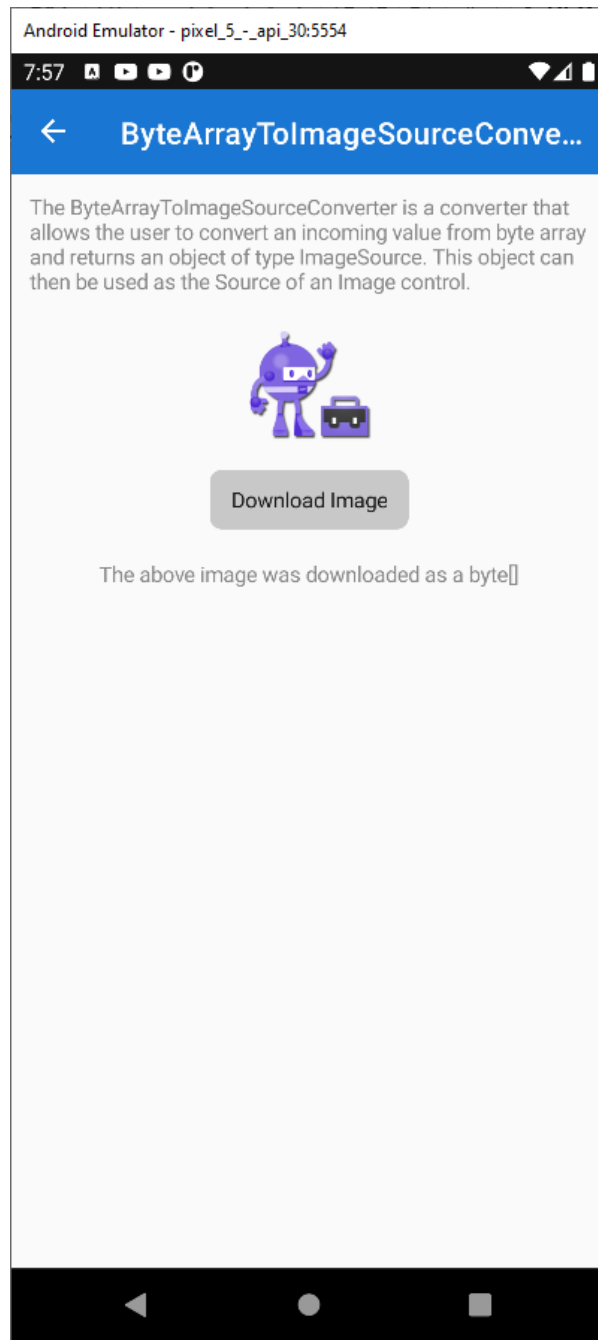


Figure 16: Converting a Byte Array into an Image

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 [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 .NET MAUI 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>
    <mct:DateTimeOffsetConverter x:Key="DateTimeOffsetConverter" />
</ResourceDictionary>
```

The usage is then very simple because 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 partial class DateTimeOffsetConverterViewModel : BaseViewModel
{
    [ObservableProperty]
    DateTimeOffset theDate = DateTimeOffset.Now;
}
```

 **Tip:** The `ObservableProperty` attribute is implemented in a library called [.NET Community Toolkit](#), which includes objects that simplify model-view-viewmodel development. This library is also referenced by the sample project. The `ObservableProperty` attribute basically supplies change notification features to a plain CLR property in a very simple and clean way.

If you run the sample project, you will see the converter in action in the appropriate page, shown in Figure 17.

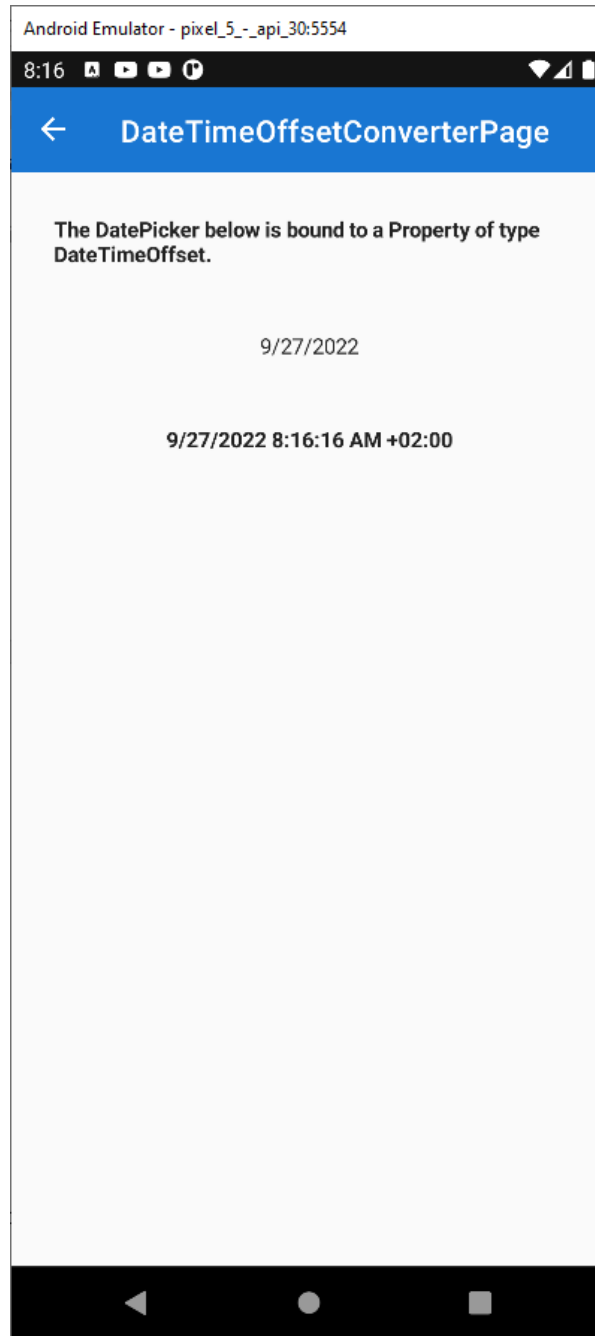


Figure 17: Converting a Date to UTC

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

Converting decimal numbers to integers

In the case that you need to convert decimal numbers of type **double** into integers of type **int**, you can leverage the **DoubleToIntConverter** class without implementing conversion logic on your own. This is demonstrated in the Pages\Converters\DoubleToIntConverterPage.xaml file. For an understanding of the result, look at Figure 18, where you can see an input decimal number converted (and rounded when appropriate) into an integer.

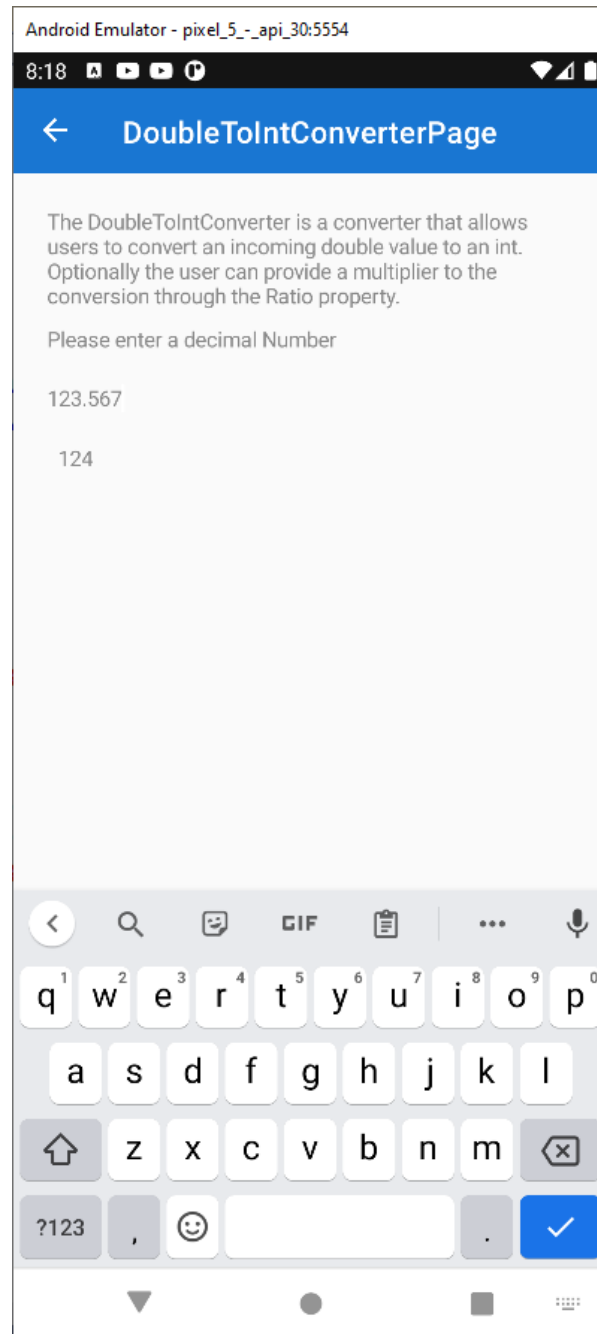


Figure 18: Converting Decimal Numbers into Integers

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

```
<ResourceDictionary>
    <mct: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 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 **ConvertFrom** method of the converter.

```
public override int ConvertFrom(double value, object? parameter = null,
    CultureInfo? culture = null) =>
    (int)Math.Round(value * GetParameter(parameter));
```

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 switch
{
    null => Ratio,
    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)),
    _ => throw new ArgumentException(
        "Parameter must be a valid number.")
};
```

This ratio is then multiplied by the input **double** number. This allows you to control 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. It allows you to specify which values from the enumeration must be treated as **true**, and then you can use the resulting **bool** as you like. The sample page lets you select an operating system to see if some text is visible for it, and it looks like Figure 19.

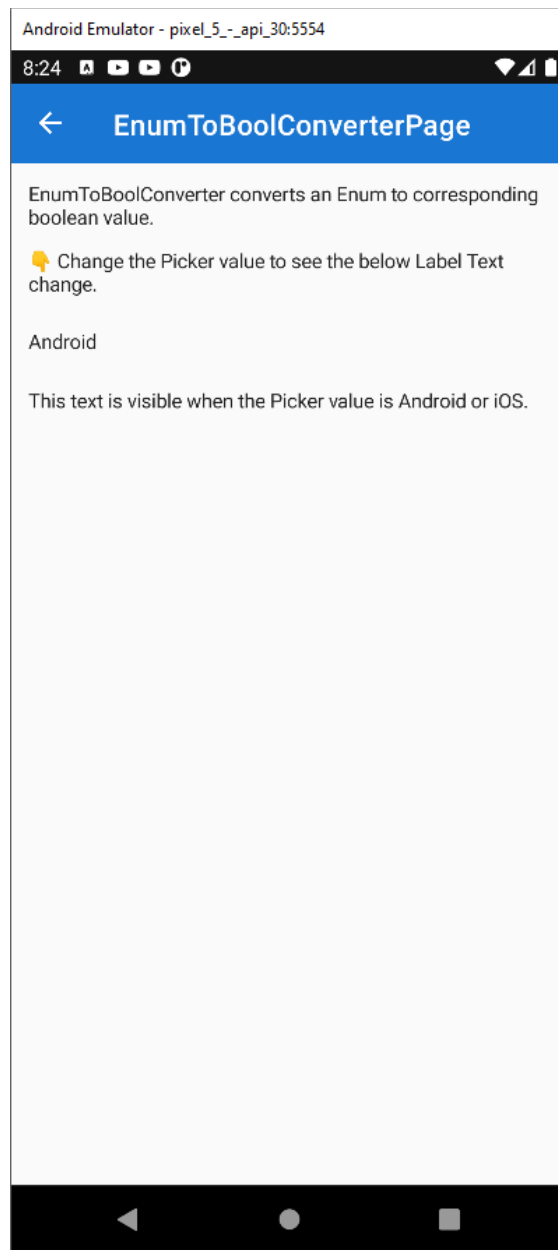


Figure 19: Converting Enumeration Values into Boolean Values

In the example, the **Picker** that you use to select a platform is bound to a collection of **MyDevicePlatform** enumerations, defined as follows.

```
public enum MyDevicePlatform
{
    Android,
    iOS,
    MacCatalyst,
    Tizen,
    WinUI,
}
```

Label views that display the text depending on the selected platform are defined as follows.

```
<!-- Converter with TRUE list -->
<Label IsVisible="{Binding SelectedPlatform,
    Converter={StaticResource MobileConverter}}"
    Text="This text is visible when the Picker value is Android or iOS."/>
<Label IsVisible="{Binding SelectedPlatform,
    Converter={StaticResource DesktopConverter}}"
    Text="This text is visible when the Picker value is MacCatalyst or WinUI."/>
<!-- Converter, that uses parameter -->
<Label IsVisible="{Binding SelectedPlatform,
    Converter={StaticResource PlatformConverter},
    ConverterParameter={x:Static vm:MyDevicePlatform.Tizen}}"
    Text="This text is visible when the Picker value is Platform is Tizen."/>
```

The **IsVisible** property of each **Label** is bound to an instance of the **EnumToBoolConverter**, as you can see in the page resources represented in Code Listing 3.

Code Listing 3

```
<pages:BasePage.Resources>
    <ResourceDictionary>
        <!-- Converter with TRUE list -->
        <mct:EnumToBoolConverter x:Key="MobileConverter">
            <mct:EnumToBoolConverter.TrueValues>
                <vm:MyDevicePlatform>Android</vm:MyDevicePlatform>
                <vm:MyDevicePlatform>iOS</vm:MyDevicePlatform>
            </mct:EnumToBoolConverter.TrueValues>
        </mct:EnumToBoolConverter>
        <mct:EnumToBoolConverter x:Key="DesktopConverter">
            <mct:EnumToBoolConverter.TrueValues>
                <vm:MyDevicePlatform>MacCatalyst</vm:MyDevicePlatform>
                <vm:MyDevicePlatform>WinUI</vm:MyDevicePlatform>
            </mct:EnumToBoolConverter.TrueValues>
        </mct:EnumToBoolConverter>
        <!-- Converter that uses parameter -->
```

```
<mct:EnumToBoolConverter x:Key="PlatformConverter" />
</ResourceDictionary>
</pages:BasePage.Resources>
```

The **MobileConverter** and **DesktopConverter** definitions contain a list of values from the **MyDevicePlatform** enumeration that must be treated as **true**. The **PlatformConverter** definition does not contain any true values because one is provided as a converter parameter in the last **Label** (refer to the code shown previously).

With all this code, when the user selects a platform from the **Picker**, each **Label** becomes visible if the specified converter returns true for the selected value in the enumeration. And this is how the **EnumToBoolConverter** works: it returns **true** for the specified values inside an enumeration.

Converting enumerations into integer values

There is another, similar class called **EnumToIntConverter** that allows converting values inside an enumeration into their primitive, underlying value. This is demonstrated in the `Pages\Converters\EnumToIntConverterPage.xaml` file. The sample page allows you to select a state for a hypothetical issue on GitHub, and it looks like Figure 20.

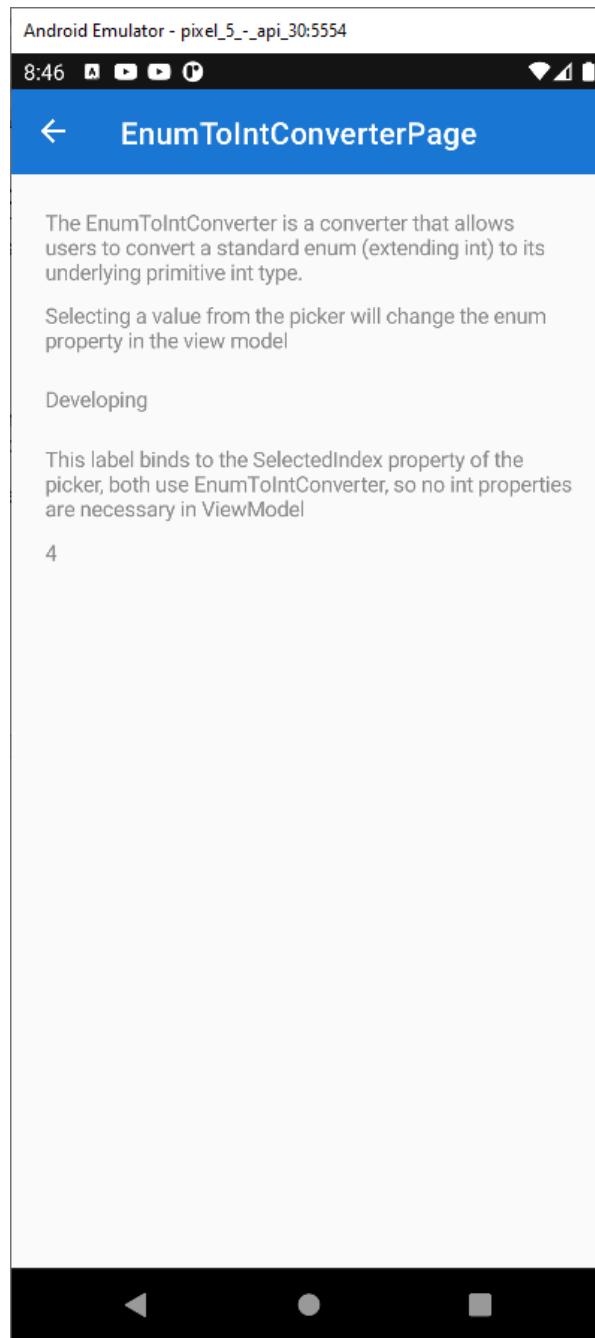


Figure 20: Converting Enumerations into Integers

The **Picker** is bound to a collection called **AllStates**, of type **ReadOnlyList<string>**. This collection is populated with the string representation of the values of an enumeration called **IssueState**, defined as follows.

```
public enum IssueState
{
    None = 0,
    New = 1,
```



```

    Open = 2,
    Waiting = 3,
    Developing = 4,
    WantFix = 5,
    Rejected = 6,
    Resolved = 7
}

```



Tip: The *AllStates* collection receives the string representation of the enumeration values via the *Enum.GetNames* method. This is visible in the *EnumToIntConverterViewModel.cs* file.

The viewmodel also defines a property called **SelectedState** of type **IssueState**, used to bind the current state to the user interface. A converter instance is declared in the page as follows.

```

<ResourceDictionary>
    <mct:EnumToIntConverter x:Key="EnumToIntConverter" />
</ResourceDictionary>

```

About data binding, the following **Label** displays the underlying integer value from the enumeration by invoking the **EnumToIntConverter**.

```

<Label Text="{Binding Path=SelectedState,
    Converter={StaticResource EnumToIntConverter}}"
    TextColor="{StaticResource NormalLabelTextColor}" />

```

Converting enumeration values into their underlying integers is another very common scenario, so this converter will be very helpful in many situations.

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 and then, for each image, in the **Properties** tool window, you set the **Build Action** property as **EmbeddedResource** and the **Copy To Output Directory** property 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 the assembly name, folder name, and image file name.

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. For this reason, you need a converter, and the .NET MAUI 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, you will see the result of the conversion as shown in Figure 21.

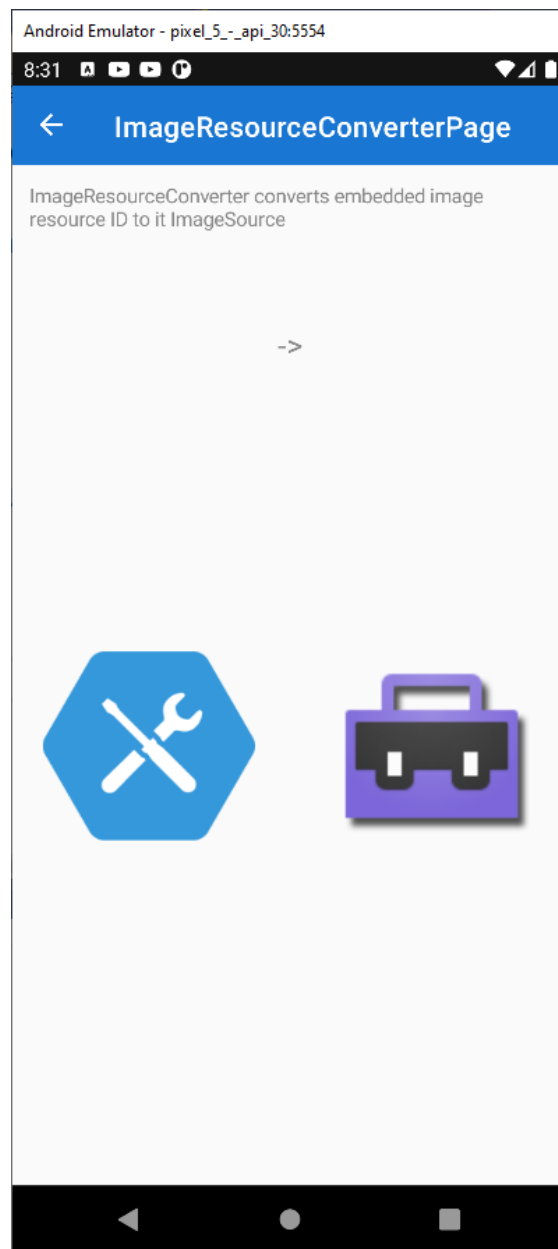


Figure 21: Displaying Images from App Resources

The sample project uses two local images under the Resources\Embedded subfolder, both called mct.png. These images simply represent the Xamarin and .NET MAUI logos. Code Listing 4 shows the code for the viewmodel, located at ViewModels\Converters\ImageResourceConverterViewModel.cs, where there is code that builds the image ID the proper way and the properties that expose the image to the UI.

Code Listing 4

```
public class ImageResourceConverterViewModel : BaseViewModel
{
    public string XamarinImageResource { get; } =
        BuildImageResource("MCT.png");
    public string MauiImageResource { get; } =
        BuildImageResource("MCT.png");

    static string BuildImageResource(in string resourceName) =>
        $"{System.Reflection.Assembly.GetExecutingAssembly().
            GetName().Name}.Resources.Embedded.{resourceName}";
}
```

As you can see, the **BuildImageResource** method retrieves the assembly name, then it concatenates the subfolder name and finally the image name. The resulting image ID is returned by the **XamarinImageResource** and **MauiImageResource** properties of type **string**. In the sample page, a reference to the **ImageResourceConverter** class is added as follows.

```
<ResourceDictionary>
    <mct:ImageResourceConverter x:Key="ImageResourceConverter"/>
</ResourceDictionary>
```

The **Image** views are bound to the properties of the viewmodel and invoke the **ImageResourceConverter** as follows.

```
<Image Grid.Row="1" Grid.Column="0"
    Source="{Binding XamarinImageResource,
        Converter={StaticResource ImageResourceConverter}}"
    WidthRequest="150" HeightRequest="150" HorizontalOptions="End"/>

<Image Grid.Row="1" Grid.Column="2"
    Source="{Binding MauiImageResource,
        Converter={StaticResource ImageResourceConverter}}"
    WidthRequest="150" HeightRequest="150" HorizontalOptions="End"/>
```

In this way, you can quickly display an image from the app resources, taking advantage of the data-binding automation without the need to write C# code that handles 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.

```
<ContentPage.Resources>
    <mct:IndexToArrayItemConverter x:Key="IndexToArrayItemConverter" />
    <x:Array x:Key="StringArray" Type="{x:Type x:String}">
        <x:String>Value0</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:String>Value6</x:String>
        <x:String>Value7</x:String>
    </x:Array>
</ContentPage.Resources>
```

The declared array contains a series of **string** values, and the converter is declared the usual way. The sample app displays the list of the array items inside a **CollectionView**, and then via a **Stepper**, it allows selecting an index of the array. The appearance of the current index will change. Figure 22 shows the result of this example.

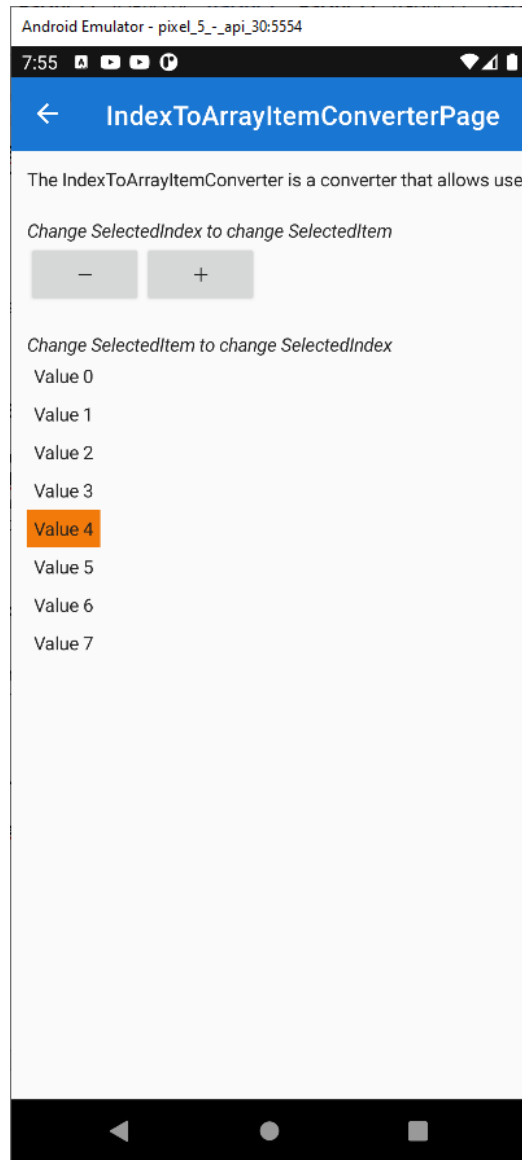


Figure 22: Converting an Array Index to an Object Item

The code for the **Stepper** is the following.

```
<Stepper x:Name="Stepper" Minimum="0" Maximum="7"
    Value="{Binding SelectedIndex}" Increment="1" Grid.Column="0"
    Grid.Row="2"/>
```

The **Stepper** is bound to the **SelectedIndex** property of the viewmodel, which represents the index of the selected item from the array. When the value of the **Stepper** changes and the **SelectedIndex** property is updated accordingly, the following **CollectionView** changes the selected item to reflect the value in the **Stepper**, as demonstrated in its definition.

```

<CollectionView SelectionMode="Single"
    SelectedItem="{Binding SelectedIndex,
        Converter={StaticResource IndexToArrayItemConverter},
        ConverterParameter={StaticResource StringArray}}"
    ItemsSource="{StaticResource StringArray}"
    Grid.Row="4" Grid.ColumnSpan="2">
    <CollectionView.ItemTemplate>
        <DataTemplate>
            <ContentView>
                <Label Padding="5" Text="{Binding}"/>
            </ContentView>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>

```



Tip: The color for the selected item in the *CollectionView* is the default one. This is why there is no code to handle the selection color.

The data source for the **CollectionView** is the array of strings defined previously, and the item selection is handled by binding the **SelectedItem** property, of type **object**, to the **SelectedIndex** property of the viewmodel. However, the latter is of type **int**, so the **IndexToArrayItemConverter** class converts the input integer to an **object**. Notice how you also need to supply the input array via the **ConverterParameter** property of the binding. The conversion is performed inside the **ConvertFrom** method of the **IndexToArrayItemConverter** class and looks like this:

```

public override object? ConvertFrom(int value, Array parameter,
    CultureInfo? culture = null)
{
    ArgumentNullException.ThrowIfNull(parameter);

    if (value < 0 || value >= parameter.Length)
    {
        throw new ArgumentOutOfRangeException(nameof(value),
            "Index was out of range.");
    }

    return parameter.GetValue(value);
}

```

Basically, the **GetValue** method from the **Array** class returns an **object** that can be bound to the caller view.

Converting from integer to Boolean

If you need to bind integer values to **false** and **true** Boolean values, the .NET MAUI 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, as usual.

```
<ResourceDictionary>
    <mct: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 .NET MAUI Community Toolkit provides the **InvertedBoolConverter** class, demonstrated in the `Pages\Converters\InvertedBoolConverter.xaml` file. The converter is declared as follows.

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

In the sample code, the second **Switch** is assigned with the opposite state of the first **Switch** view, as follows.

```
<Switch IsToggled="{Binding IsToggled, Mode=OneWay,
    Converter={StaticResource InvertedBoolConverter}}" IsEnabled="false"/>
```

The data binding is not direct between views. It happens via the **IsToggled** property, of type **bool**, defined in the backing viewmodel. The result of the code in the sample app is shown in Figure 23.

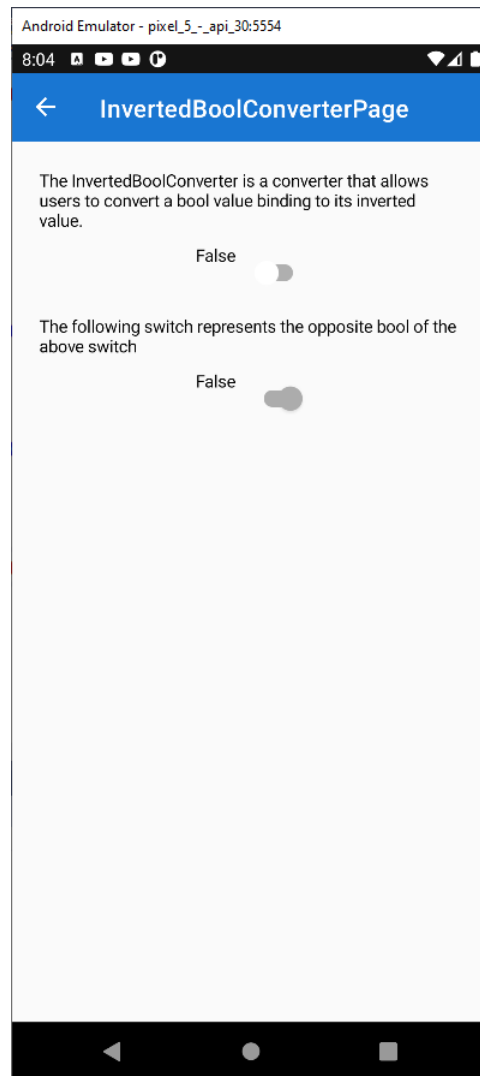



Figure 23: Inverting the Binding of a **bool** Value

 **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.

Converting objects into Boolean values and comparing for equality

The .NET MAUI Community Toolkit exposes the **IsEqualConverter** class, 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 compare it to. In the sample project, this is demonstrated in `Pages\Converters\IsEqualConverterPage.xaml`. More specifically, it demonstrates how to return **true** or **false** if the input value equals the string **MAUI**, which is the value specified in the **ConverterParameter**. Figure 24 shows what the example looks like.

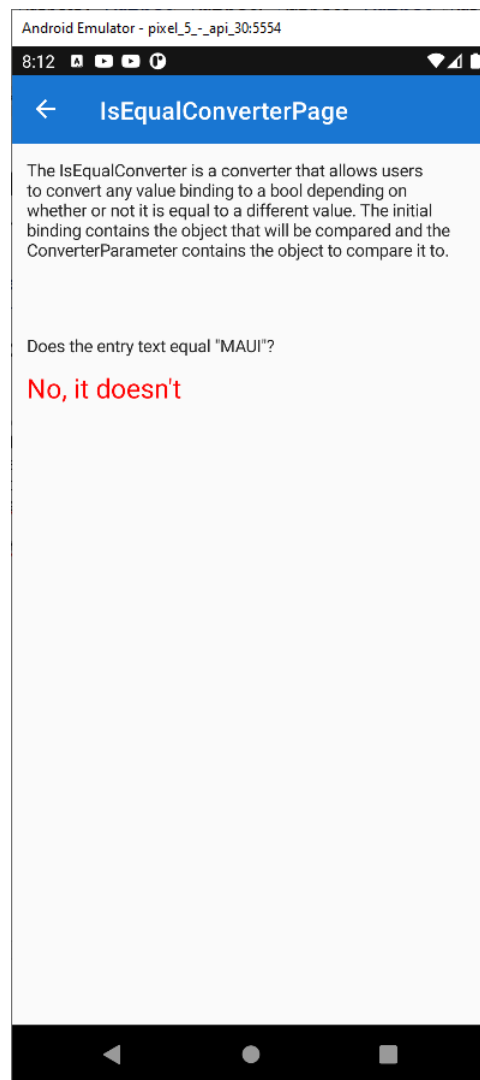


Figure 24: Converting Equality Comparisons to **bool**

You can type a string in the **Entry** and the app will check whether it equals the **MAUI** string declared in the page resources. If it does, a green confirmation message will appear. The last **Label** from the top shows how the converter is actually used.

```

<Label FontSize="Large" HorizontalOptions="FillAndExpand"
    Text="Yes, it does!" TextColor="Green">
    <Label.Triggers>
        <DataTrigger TargetType="Label"
            Binding="{Binding InputValue, Converter={StaticResource
                IsEqualConverter},
            ConverterParameter={StaticResource MAUI}}"
            Value="False">
            <Setter Property="Text" Value="No, it doesn't" />
            <Setter Property="TextColor" Value="Red" />
        </DataTrigger>
    </Label.Triggers>
</Label>

```

The **IsEqualConverter** 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. Notice how, in this particular case, the converter is invoked in the **Label**'s data triggers, which means that it is invoked only when the bound value changes. As an opposite tool, the **IsNotEqualConverter** works exactly like **IsEqualConverter**, except that it returns **true** if two objects are not equal and **false** if they are equal.

Comparing objects that implement **Comparable**

The Community Toolkit provides another converter called **CompareConverter**, which allows for comparing two objects that implement the **Comparable** interface given a condition. Depending on the result of the comparison (**true** or **false**), you can decide if the converter should return **true**, **false**, or a specific object. This is demonstrated in the `Pages\Converters\CompareConverterPage.xaml` file. Before you run the example, in case you are running the Android emulator, a good idea is adding the following colors to the **Slider**.

```

ThumbColor="Black"
MinimumTrackColor="Black"
MaximumTrackColor="Black"

```

This will make the **Slider** more visible, otherwise its default color would be too light on screen. The following example shows how to compare the value of the **Slider** to a **double** of 0.5 declared in the page resources as **ComparingValue**, and the condition is **GreaterOrEqual**.

```

<Label
    Text="{Binding SliderValue, Mode=OneWay,
    Converter={mct:CompareConverter ComparingValue={StaticResource
    ComparingValue}, ComparisonOperator=GreaterOrEqual}}" />

```

When the value of the **Slider** is greater than or equal to 0.5, the **CompareConverter** returns **true**, otherwise **false**, and the Boolean result is displayed as the text of the **Label**. Figure 25 shows an example.

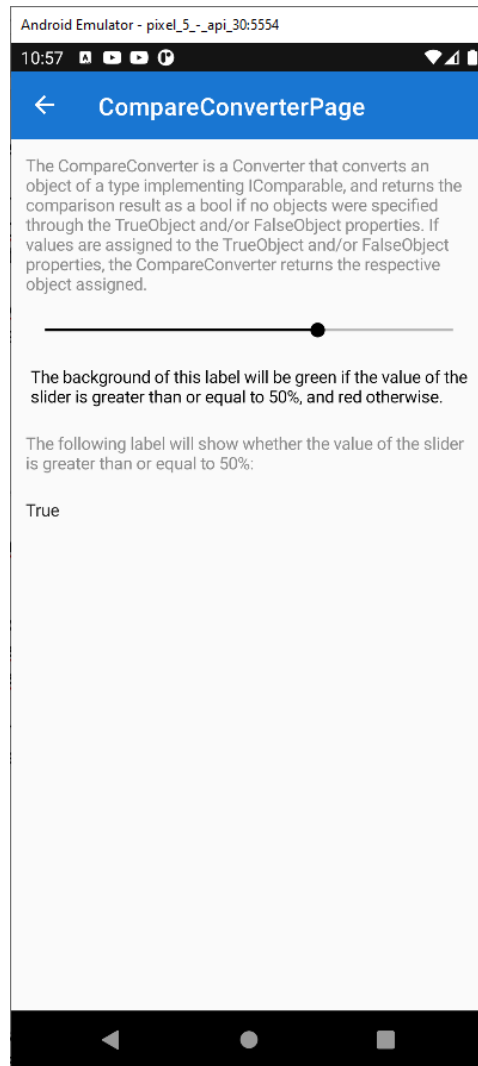


Figure 25: Comparisons by *IComparable*

The condition for the comparison can be one of the self-explanatory values from the **OperatorType** enumeration, also defined in the Community Toolkit (see the CompareConverter.shared.cs file).

```
[Flags]
public enum OperatorType
{
    NotEqual = 0,
    Smaller = 1 << 0,
    SmallerOrEqual = 1 << 1,
    Equal = 1 << 2,
    Greater = 1 << 3,
    GreaterOrEqual = 1 << 4,
}
```

In the previous example, the converter returns a Boolean value. However, if you look at the following sample code in the same page, you can see how an object can be returned instead.

```
<Label Text="The background of this label will be green if the value of the
    slider is greater than or equal to 50%, and red otherwise."
    BackgroundColor="{Binding SliderValue, Mode=OneWay,
    Converter={mct:CompareConverter ComparingValue={StaticResource
    ComparingValue}, ComparisonOperator=GreaterOrEqual,
    TrueObject=LightGreen, FalseObject=PaleVioletRed}}"
    TextColor="Black" Padding="4, 0"/>
```

In this example, if the comparison is successful, the object assigned to the **TrueObject** property of the converter is returned, otherwise the object assigned to the **FalseObject** property is returned. Both objects are of type **Color** and affect the background of the **Label**. This is certainly a very useful converter, but it requires that both parts of the comparison implement the **IComparable** interface. In all the other cases, you can use the **IsEqualConverter** described at the beginning of this section.

Checking for string values

The .NET MAUI Community Toolkit provides two converters that detect whether a string is not null and return a **bool** value if this is the case: **IsStringNotNullOrEmptyConverter** returns **false** if a string is null or if it is of type **string.Empty**, and **IsStringNotNullOrWhiteSpaceConverter** returns **false** if a string is null or if it contains white spaces. They both return **true** if a string is not null. In the sample project, they are demonstrated in the `Pages\Converters\IsStringNotNullOrEmptyConverterPage.xaml` and `Pages\Converters\IsStringNotNullOrWhiteSpaceConverterPage.xaml` files respectively. Both sample pages provide an **Entry** where you can enter a string. When the string is null or empty for the first converter, or null or white spaces for the second one, a bound **Label** displays **False**. When you enter some valid text, the **Label** displays **True**. Figure 26 shows how this appears.

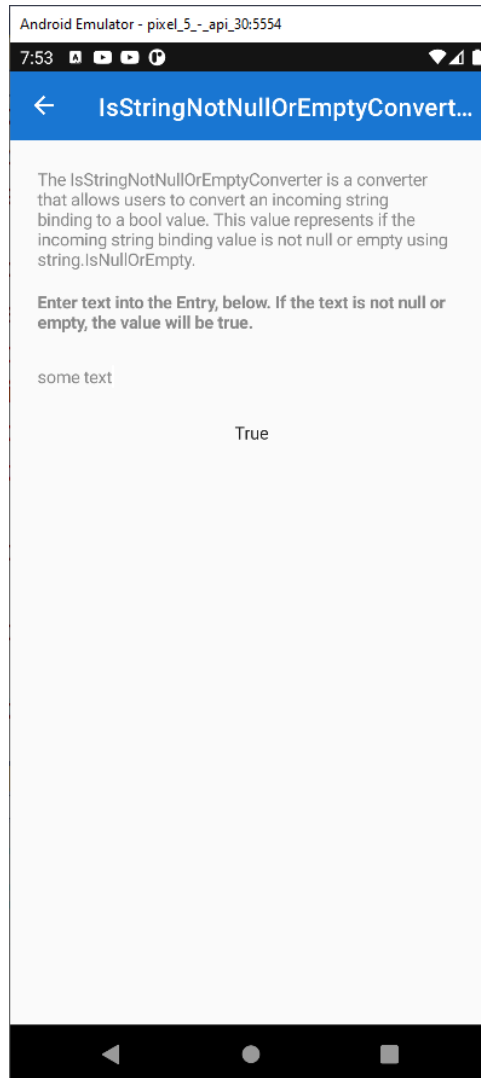


Figure 26: Checking for Null Values

The declaration of the converters is very simple, as usual.

```
<ResourceDictionary>
    <mct:IsStringNotNullOrEmptyConverter
        x:Key="IsStringNotNullOrEmptyConverter" />
</ResourceDictionary>

...

<ResourceDictionary>
    <mct:IsStringNotNullOrWhiteSpaceConverter
        x:Key="IsStringNotNullOrWhiteSpaceConverter" />
</ResourceDictionary>
```

The **Label** that displays the result is bound to the **Entry** as follows.

```

<Entry VerticalOptions="CenterAndExpand"
    HorizontalOptions="Fill" Text="{Binding Path=LabelText,
    Mode=OneWayToSource}"
    TextColor="{StaticResource NormalLabelTextColor}" />

<Label VerticalOptions="CenterAndExpand" HorizontalOptions="CenterAndExpand"
    Text="{Binding Path=LabelText, Mode=OneWay,
    Converter={StaticResource IsStringNotNullOrEmptyConverter}}" />

```

Only the referenced converter changes between the two examples, otherwise the binding expression is exactly the same.

Checking for valid strings

The .NET MAUI Community Toolkit also provides two complementary converters: **IsStringNullOrEmptyConverter** and **IsStringNotNullOrWhiteSpaceConverter**. They return **true** when a string is null or empty for the first converter, and when a string is null or contains white spaces for the second converter. You use them like the converters described previously, but with the opposite **bool** result.

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 been around 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 make this simple, the .NET MAUI Community Toolkit exposes the **IsListNotNullOrEmptyConverter** and **IsListNullOrEmptyConverter** classes, which respectively return **true** if a collection is not null or if it is null. They are demonstrated in the **IsListNotNullOrEmptyConverterPage.xaml** and **IsListNullOrEmptyConverterPage.xaml** files, both located under **Page\Converters**. Both sample pages work similarly, so I'm taking into consideration **IsListNullOrEmptyConverterPage.xaml**. If you run the sample app and open this sample page, you will see a list of items and a **Label** that displays **False** because the list is not empty. If you click the Clear List button, the data-bound collection will be cleared, and the **Label** will display **True**. Figure 27 demonstrates this.

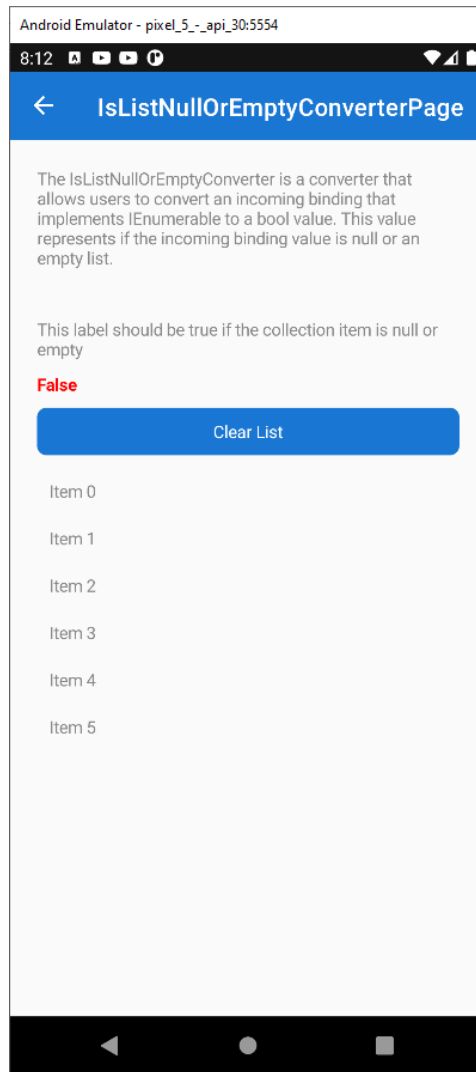


Figure 27: Checking Lists for Values

The bound data collection is called **StringItemSource** and is exposed from the **IsListNullOrEmptyConverterViewModel** class. The **Text** property of the **Label** that displays the status is assigned with the data collection, passing the converter in the binding expression. So, to retrieve the status of the data (empty or available), we do this:

```
<Label
    Text="{Binding StringItemSource,
    Converter={StaticResource IsListNullOrEmptyConverter}}"
    FontAttributes="Bold" TextColor="Red" />
```

With these simple converters, you can quickly implement empty states and error states against any item 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 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 IReadOnlyList<string> ItemSource { get; } = new[]
{
    "This",
    "Is",
    "The",
    "ListToStringConverter"
};
```

The converter is declared as follows.

```
<ResourceDictionary>
    <mct: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 ItemSource,
    Converter={StaticResource ListToStringConverter}}"
    TextColor="{StaticResource NormalLabelTextColor}"/>
```

The resulting string will be the following.

This, Is, The, ListToStringConverter

Converting one string into a list

The MAUI Community Toolkit also provides an opposite converter, called `StringToListConverter`, which quickly creates a `List<string>` from words in a string, given a separator. This is demonstrated in the `StringToListConverterPage.xaml` file with the following relevant code.

```
<Entry
    FontSize="Medium"
    Placeholder="Enter some text separated by ',' or '.' or ';' "
    Text="{Binding LabelText, Mode=OneWayToSource}" />

<CollectionView
    ItemsSource="{Binding LabelText, Mode=OneWay,
    Converter={StaticResource StringToListConverter}}">
```



```

<CollectionView.ItemTemplate>
    <DataTemplate x:DataType="x:String">
        <Label FontSize="Medium" Text="{Binding .}"
            TextColor="{StaticResource NormalLabelTextColor}" />
    </DataTemplate>
</CollectionView.ItemTemplate>
</CollectionView>

```

If you added the following string in the **Entry**:

```
This;is;a;string
```

The converter would return a new **List<string>** with four strings: **This**, **is**, **a**, and **string**.

Changing string casing

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

```

<ResourceDictionary>
    <mct: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 provided 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 .NET MAUI 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 how powerful it is, let's consider the code of the sample page. The purpose of the sample page is to convert an input string to uppercase and, by default, check whether it is different from the MAUI string. This is possible by combining the **TextCaseConverter** class with the **IsNotEqualConverter** class, both of which you saw previously. The **MultiConverter** class can contain multiple converters, as follows.

```
<mct:MultiConverter x:Key="MyMultiConverter">
    <mct:TextCaseConverter />
    <mct:IsNotEqualConverter />
</mct: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 mct:MultiConverterParameter}">
    <mct:MultiConverterParameter
        ConverterType="{x:Type mct:TextCaseConverter}"
        Value="{x:Static mct:TextCaseType.Upper}" />
    <mct:MultiConverterParameter
        ConverterType="{x:Type mct:IsNotEqualConverter}"
        Value="MAUI" />
</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 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},
    Mode=OneWay}" 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 how it works in practice, but the biggest benefit to remember is that you can use multiple converters simultaneously, including converters defined outside of the .NET MAUI Community Toolkit.

Converting multiple Boolean values into one

The .NET MAUI Community Toolkit provides the **VariableMultiValueConverter** class, which allows for converting multiple **bool** values into one based on the specified conditions. This converter is demonstrated in the `Pages\Converters\VariableMultiValueConverterPage.xaml` file. For a better understanding, consider Figure 28, which shows the converter in action.

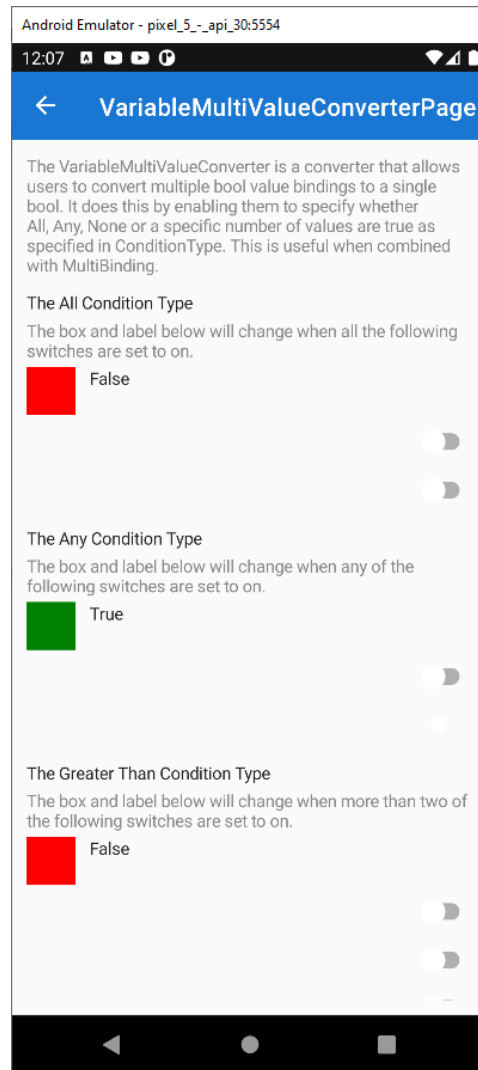


Figure 28: Converting Multiple Booleans into One

If you play with the various checkboxes, you will see the colored boxes change their color from red to green when the corresponding checkboxes 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 for as many conditions you need, in this case three.

```

<pages:BasePage.Resources>
    <mct:VariableMultiValueConverter x:Key="AllTrueConverter"
                                    ConditionType="All" />
    <mct:VariableMultiValueConverter x:Key="AnyTrueConverter"
                                    ConditionType="Any" />
    <mct:VariableMultiValueConverter x:Key="GreaterThanConverter"
                                    ConditionType="GreaterThan"
                                    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 they are applied to, 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 changes the converter to which they refer.

```

<BoxView BackgroundColor="Red" HeightRequest="40" WidthRequest="40"
    Grid.Row="3">
    <BoxView.Triggers>
        <DataTrigger TargetType="BoxView" Value="true">
            <DataTrigger.Binding>
                <MultiBinding
                    Converter="{StaticResource AllTrueConverter}">
                    <Binding Path="IsAllGroupSwitch1On" />
                    <Binding Path="IsAllGroupSwitch2On" />
                </MultiBinding>
            </DataTrigger.Binding>
            <Setter Property="BackgroundColor" Value="Green" />
        </DataTrigger>
    </BoxView.Triggers>
</BoxView>

```

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

Converting strings to mathematical expressions

The .NET MAUI Community Toolkit introduces the **MathExpressionConverter** class for simplifying the generation of mathematical expressions from user input. For a better understanding of it, consider Figure 29, where a **Stepper** allows for selecting a value and two mathematical operations are calculated and displayed below it, a division by 2 and a multiplication by 5.

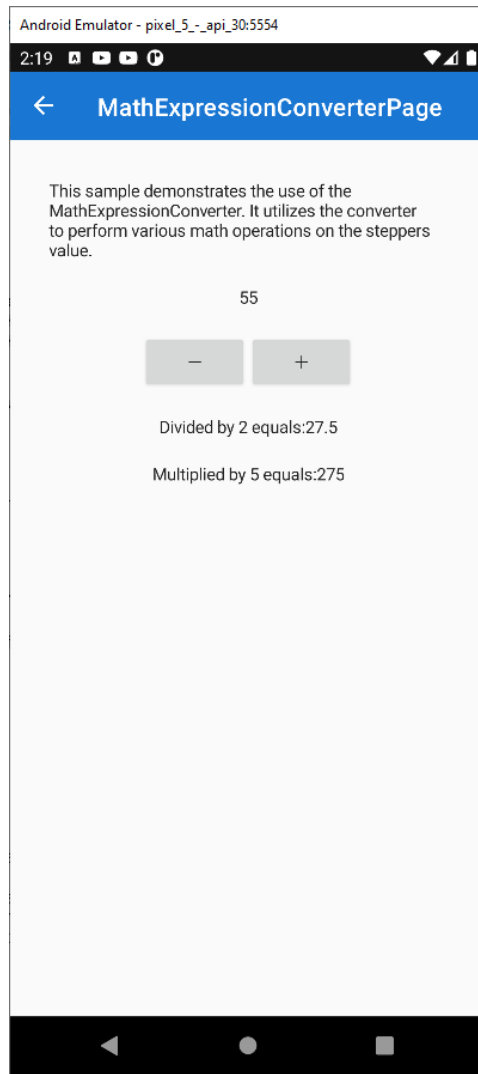


Figure 29: Converting the User Input to Mathematical Expressions

The first expression, the division, is displayed via the following **Label**.

```
<Label HorizontalOptions="Center">
    <Label.FormattedText>
        <FormattedString>
            <Span Text="Divided by 2 equals:"/>
            <Span Text="{Binding SliderValue,
                Converter={StaticResource MathExpressionConverter},
                ConverterParameter='x/2'}"/>
        </FormattedString>
    </Label.FormattedText>
</Label>
```

The **SliderValue**-bound object is a property of type **double** defined in the viewmodel, and it stores the value selected in the **Stepper**. The division expression is calculated by invoking the **MathExpressionConverter** object, specifying the operation type through the **ConverterParameter** binding property. In this case, the parameter represents a division by 2 and **x** is the dividend. Similarly, the result of the multiplication is displayed via the following **Label**.

```
<Label HorizontalOptions="Center" >
    <Label.FormattedText>
        <FormattedString>
            <Span Text="Multiplied by 5 equals:"/>
            <Span Text="{Binding SliderValue,
                Converter={StaticResource MathExpressionConverter},
                ConverterParameter='x*5'}"/>
        </FormattedString>
    </Label.FormattedText>
</Label>
```

In this case, **x** represents the first operand. Behind the scenes, the converter creates an instance of the **MathExpression** class, which is new in the .NET MAUI Community Toolkit, passing the first value of the expression (**SliderValue** in the previous examples) and the value of the **ConverterParameter**. You can see this in the **ConvertFrom** method definition, located inside the **MathExpressionConverter.shared.cs** file.

```
public override double ConvertFrom(double value, string parameter,
    CultureInfo? culture = null)
{
    ArgumentNullException.ThrowIfNull(parameter);
    var mathExpression = new MathExpression(parameter, new[] { value });
    return mathExpression.Calculate();
}
```

Notice that the sample application is passing only one value along with the converter parameter, but as you can see from the code, it is possible to pass an array of **double** or an **IEnumerable<double>**. The **Calculate** method then returns the result of the mathematical expression.



Tip: The list of supported operations and expressions is visible inside the constructor of the **MathExpression** class, located in the **MathExpression.Shared.cs** file. You can right-click **MathExpression** in the **ConvertFrom** method and then select **Go To Definition**, so you can quickly go to the class's constructor mentioned before.

Calculating complex expressions

When you need to generate complex expressions, you can use the **MultiMathExpressionConverter** class, which is declared the usual way, as follows.

```

<ResourceDictionary>
    <mct:MultiMathExpressionConverter x:Key="MultiMathExpressionConverter"/>
</ResourceDictionary>

```

The sample code for this converter is available in the Pages\Converters\MultiMathExpressionConverterPage.xaml file. If you look at Figure 30, you can see how multiple values are entered as strings and how the result of the operation is displayed in one **Label**.

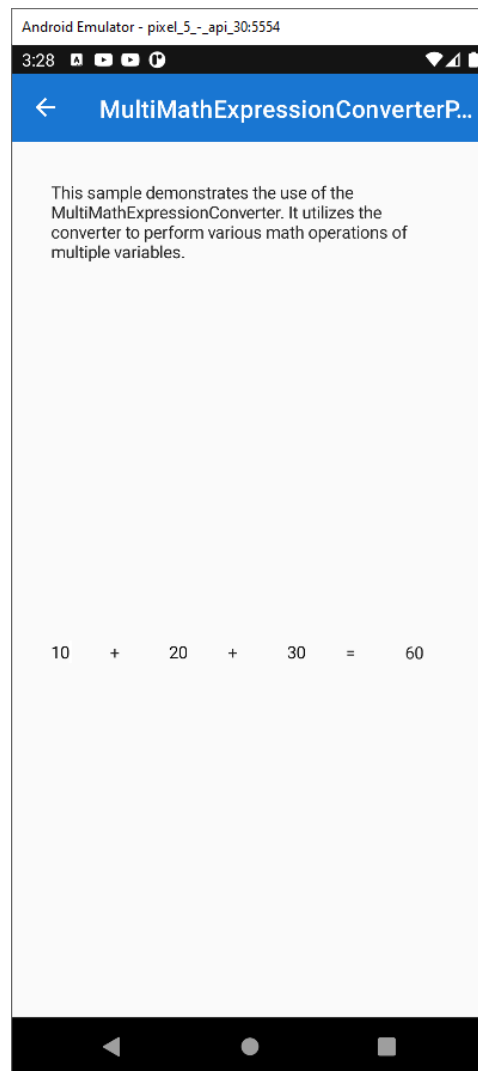


Figure 30: Converting the User Input to Complex Expressions

The code that displays the result is the following.

```

<Label Grid.Column="6"
      Grid.Row="1"
      VerticalTextAlignment="Center"
      VerticalOptions="Center">
    <Label.Text>

```

```

        <MultiBinding Converter="{StaticResource MultiMathExpressionConverter}"
            ConverterParameter="x0 + x1 + x2">
            <Binding Path="X0" Mode="OneWay"/>
            <Binding Path="X1" Mode="OneWay"/>
            <Binding Path="X2" Mode="OneWay"/>
        </MultiBinding>
    </Label.Text>
</Label>

```

The **MultiBinding** expression attaches multiple bindings to one target. In this case, the three **Binding** expressions point to the values that represent the operands of the **addition**. The **ConverterParameter** of the **MultiMathExpressionConverter**, invoked in the **MultiBinding** expression, passes the values to the constructor of the **MathExpression** class, which has been discussed previously. Inside the **ConverterParameter**, values are referenced via the **x** literal plus the index (with zero base) of each **Binding** in the **MultiBinding** expression. So, **x0** refers to the **Binding Path="X0"** expression (index zero), **x1** refers to the **Binding Path="X1"** expression (index 1), and **x2** refers to the **Binding Path="X2"** expression (index 2). Obviously, you are not limited to one operator. In fact, the power of the **MultiMathExpressionConverter** is that you can have different operators in one expression (for example, you might have both the addition and multiplication operators).

Converting colors

The .NET MAUI Community Toolkit exposes 21 classes for converting **Color** instances into different types. Table 5 summarizes these converters.



Note: In Table 5, you will find several acronyms related to colors. If you are not familiar with these, it is good to know that the **K** in **CMYK** stands for key, but it represents black. On the other hand, the **A** in **RGBA** and **CMYKA** stands for alpha and it represents alpha-blending, a value used to calculate the amount of transparency in a color.

Table 5: Color Converters

| Color converter | Description |
|-----------------------------------|--|
| ColorToRgbStringConverter | Returns a string that contains the RGB (red, green, blue) values of the specified color separated by commas. |
| ColorToRgbaStringConverter | Returns a string that contains the RGBA (red, green, blue, alpha) values of the specified color separated by a comma. |

| Color converter | Description |
|--|--|
| ColorToHexRgbStringConverter | Returns a string that contains the hexadecimal representation of the specified color generated from its RGB values. |
| ColorToHexRgbaStringConverter | Returns a string that contains the hexadecimal representation of the specified color generated from its RGBA values. |
| ColorToCmykStringConverter | Returns a string that contains the CMYK (cyan, magenta, yellow, key) values of the specified color separated by commas. |
| ColorToCmykaStringConverter | Returns a string that contains the CMYKA (cyan, magenta, yellow, key, alpha) values of the specified color separated by commas. |
| ColorToHslStringConverter | Returns a string that contains the HSL (hue, saturation, luminosity) values of the specified color separated by commas. |
| ColorToHslaStringConverter | Returns a string that contains the HSLA (hue, saturation, luminosity, alpha) values of the specified color separated by commas. |
| ColorToPercentBlackKeyConverter | Returns a string that contains the percentage of black key from the specified color. |
| ColorToByteAlphaConverter | Returns the byte representation of the specified color's alpha-blending. |
| ColorToByteRedConverter | Returns the byte representation of the specified color's red component. |
| ColorToByteGreenConverter | Returns the byte representation of the specified color's green component. |

| Color converter | Description |
|---------------------------------------|--|
| ColorToByteBlueConverter | Returns the byte representation of the specified color's blue component. |
| ColorToPercentCyanConverter | Returns a float that represents the percentage of cyan in the specified color. |
| ColorToPercentMagentaConverter | Returns a float that represents the percentage of magenta in the specified color. |
| ColorToPercentYellowConverter | Returns a float that represents the percentage of yellow in the specified color. |
| ColorToDegreeHueConverter | Returns a float representing the degree of hue in the specified color. |
| ColorToBlackOrWhiteConverter | Returns the closest monochrome color (Color.Black or Color.White) for the specified color. If the supplied color is dark, the converter returns Black . If it is light, it returns White . |
| ColorToColorForTextConverter | Returns Color.Black or Color.White if the supplied color is determined to be dark for the human eye or light. |
| ColorToGrayScaleColorConverter | Converts the supplied color into grayscale and returns a new Color . |
| ColorToInverseColorConverter | Returns the inverse of the supplied color. |

Figure 31 shows the sample page in action, where you can see the results of most converters.

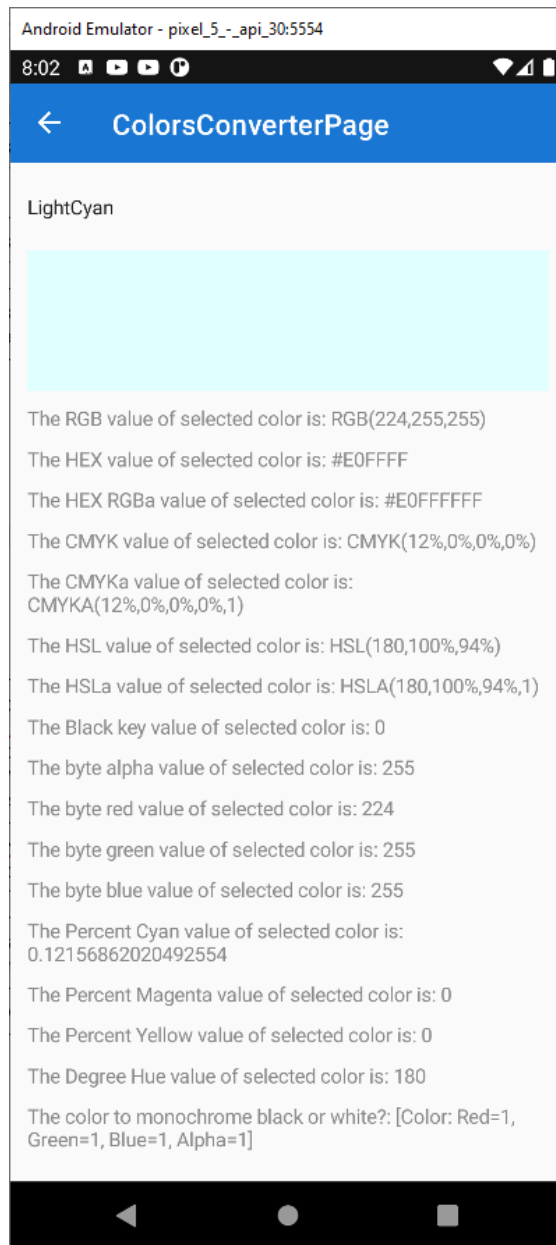


Figure 31: Demonstration of Color Converters

Because of the large number of converters, and because the way you use such converters is always the same except for the return type and data, only two examples will be discussed here. You can refer to the official sample code and [documentation](#) for more information. In real-world development, it is common to retrieve the RGB and hexadecimal representations of a color. This is accomplished with the following converters.

```
<mct:ColorToRgbStringConverter x:Key="ColorToRgbStringConverter" />
...
<mct:ColorToHexRgbStringConverter x:Key="ColorToHexRgbStringConverter" />
```

In the sample page (Pages\Converters\ColorConverters.xaml), the two converters are used as follows.

```
<Label TextColor="{StaticResource NormalLabelTextColor}"
      Text="{Binding Source={x:Reference BoxView}, Path=BackgroundColor,
      Converter={StaticResource ColorToRgbStringConverter},
      StringFormat='The RGB value of selected color is: {0}'}"/>

<Label TextColor="{StaticResource NormalLabelTextColor}"
      Text="{Binding Source={x:Reference BoxView}, Path=BackgroundColor,
      Converter={StaticResource ColorToHexRgbStringConverter},
      StringFormat='The HEX value of selected color is: {0}'}"/>
```

The input color, defined by the **NormalLabelTextColor** resource, is a dark gray represented by the **#888888** code. As you can see in Figure 31, the **ColorToRgbStringConverter** returns the following string.

RGB(224,255,255)

The **ColorToHexRgbStringConverter** returns the following string.

#E0FFFF

Behind the scenes, converters are defined in three different files of the .NET MAUI Community Toolkit solution: **ColorToStringConverter.shared.cs**, which contains converters that return **string** values; **ColorToColor.shared.cs**, which contains converters that return objects of type **Color**; and **ColorToComponent.shared.cs**, which contains converters that return objects of type **byte** and **float**. For each of the converters listed in Table 5, you will find a **ConvertFrom** method, which invokes other methods that perform the required conversion. These methods are defined in the **ColorConversionExtensions** class, which is analyzed in [Chapter 6 Enhancing Object Power with Extensions](#), because they can also be used outside of value converters.

Chapter summary

Data binding is one of the most powerful features in development platforms based on XAML, and in many situations you will need to bind views that support specific data types to objects of different types—this is where value converters come in. The .NET MAUI Community Toolkit provides a lot of commonly used converters so that you do not need to reinvent the wheel every time, simplifying your code base. Value converters deal with data, and so do the validation behaviors described in the next chapters.

Chapter 5 Data Validation with Reusable Behaviors

In .NET MAUI (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 .NET MAUI 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 .NET MAUI 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.

Common properties of behaviors

All the behaviors related to data validation exposed by the .NET MAUI Community Toolkit derive from the `CommunityToolkit.Maui.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

| Property | Description |
|------------------------|---|
| IsValid | Returns true when the bound data is considered valid. |
| IsValidNotValid | 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. |

| Property | Description |
|------------------|--|
| Value | The data to be validated. |
| IsRunning | Returns true when the validation process is running. |
| Flags | Of type ValidationFlags , an enumeration 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 sections, so it's important for you to know what they are about.

Validating an email address

Validating an email address that the user enters via an **Entry** view is a very common task and the .NET MAUI 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 32, where an invalid email address has been entered.

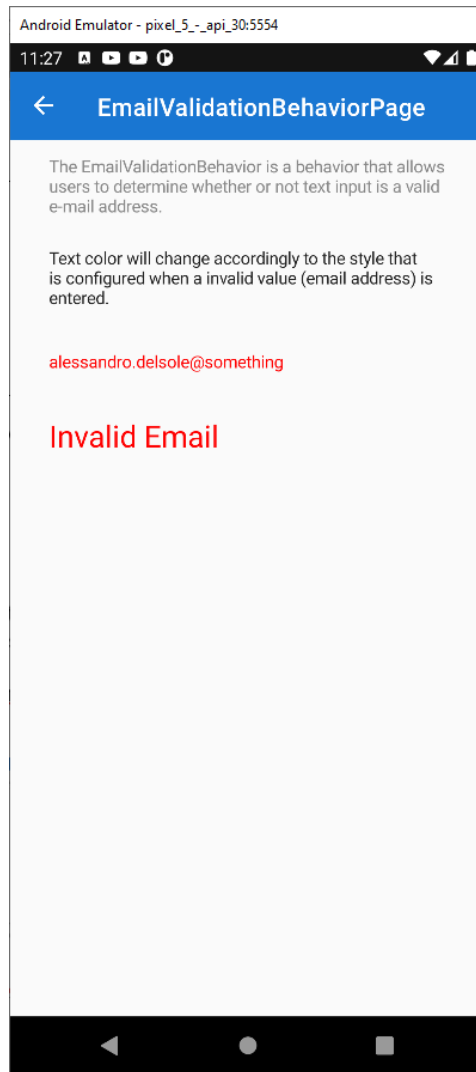


Figure 32: Email Address Validation

The provided string has been recognized as invalid by the **EmailValidationBehavior** object, which automatically performs formal validation. This behavior exposes two properties, **IsValid** and **IsValid**, that return **true** if the email address is valid or invalid, respectively. Based on these property values, you can style other UI elements. For example, with an invalid email address, the label you see in Figure 32 is set as visible and the color for the invalid text is red. When the behavior recognizes the supplied email address as valid, the views' styles return to their original state. Code Listing 5 shows how this is implemented.

Code Listing 5

```
<?xml version="1.0" encoding="UTF-8"?>
<pages:BasePage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:mct="http://schemas.microsoft.com/dotnet/2022/maui/
        toolkit"
```

```

        xmlns:pages="clr-namespace:CommunityToolkit.Maui.
            Sample.Pages"
        x:Class="CommunityToolkit.Maui.Sample.Pages.Behaviors.
            EmailValidationBehaviorPage"
        xmlns:vm="clr-namespace:CommunityToolkit.Maui.Sample.
            ViewModels.Behaviors"
        x:TypeArguments="vm:EmailValidationBehaviorViewModel"
        x:DataType="vm:EmailValidationBehaviorViewModel">

    <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>

    <VerticalStackLayout Padding="{StaticResource ContentPadding}"
        Spacing="24"
        VerticalOptions="StartAndExpand">
        <Label Text="The EmailValidationBehavior is a behavior that allows
            users to determine whether or not text input is a valid e-
            mail address. " TextColor="{StaticResource NormalLabelTextColor}" />
        <Label Text="Text color will change accordingly to the style that i
            s configured when a invalid value (email address) is entered." />
        <Entry Placeholder="Email"
            HorizontalOptions="FillAndExpand"
            VerticalOptions="Center">
            <Entry.Behaviors>
                <mct:EmailValidationBehavior
                    x:Name="EmailValidator"
                    Flags="ValidateOnValueChanged"
                    DecorationFlags="Trim"
                    ValidStyle="{StaticResource ValidEntryStyle}"
                    InvalidStyle="{StaticResource InvalidEntryStyle}"/>
            </Entry.Behaviors>
        </Entry>
        <Label Text="Invalid Email"
            VerticalTextAlignment="Start"
            HorizontalTextAlignment="Start"
            HorizontalOptions="Start"
            VerticalOptions="Start"
            TextColor="Red"
            FontSize="25"
            IsVisible="{Binding IsNotValid,
                Source={x:Reference EmailValidator}}"/>
    </VerticalStackLayout>

```



```
</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, all self-explanatory except for **Trim**, are **None**, **NormalizeWhiteSpace**, **NullToEmpty**, **Trim**, **TrimEnd**, and **TrimStart**. Finally, notice how the **Label** becomes visible only when the **IsValid** property of the behavior becomes **true**. This is a common approach to displaying an error message close to the input box.

Validating characters in a string

The .NET MAUI Community Toolkit makes it easy to validate characters in a string. This is possible via the **CharactersValidationBehavior** class that you can attach to **Entry** or **Editor** views. In the sample project, it is demonstrated in the `Pages\Behaviors\Characters\ValidationBehavior.xaml` file and its code-behind. At the time of this writing, there is a bug in the source code for this example that must be fixed manually. First, at the page definition level, you need to add a page identifier as follows.

```
x:Name="CharValidationPage"
```

Obviously, the identifier is up to you, but you will then need to use it consistently across edits. The next fix is changing the source reference in the **Picker** declaration, as follows.

```
<Picker Grid.Column="1" Title="CharacterType"
        ItemsSource="{Binding CharacterTypes,
        Source={x:Reference CharValidationPage}, Mode=OneTime}"
        SelectedIndex="1"/>
```

Now that the **Picker** is pointing to the correct source, it is possible to do the following validations. The **CharactersValidationBehavior** requires you to specify values for the following properties:

- **CharacterType**: This is of type **CharacterType** and establishes the validation criterion. It 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 33. Notice that the sample code offers two examples and the second one is based on visual states, but I will use the first one.

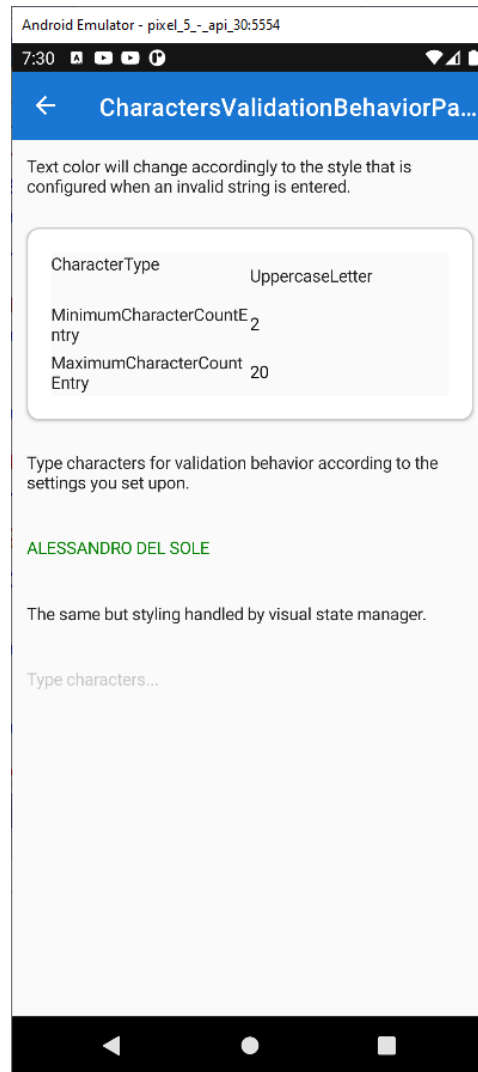


Figure 33: Validating Characters in a String

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 all uppercase and within the length limits, it is considered valid, and it is displayed in green. The XAML code for this example looks like the following.

```
<Entry Placeholder="Type characters...">
  <Entry.Behaviors>
    <mct: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>
```

```

    </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 the code-behind with the following code.

```

public IReadOnlyList<CharacterType> CharacterTypes { get; } =
    Enum.GetValues(typeof(CharacterType)).Cast<CharacterType>().ToList();

```

The values of the **CharacterType** enumeration are 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="IndianRed"/>
    </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 **Pages\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>
        <mct:MaxLengthReachedBehavior MaxLengthReached=

```

```

        "MaxLengthReachedBehavior_MaxLengthReached"
        ShouldDismissKeyboardAutomatically="{Binding Path=IsToggled,
        Source={x:Reference AutoDismissKeyboardSetting}}" />
    </Entry.Behaviors>
</Entry>

```

In the example, when the maximum length has been reached, the focus is moved to another **Entry**. All the relevant properties are data-bound to other visual elements, but the following is 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 input should be ignored 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 .NET MAUI 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 `Pages\Behaviors\NumericValidationBehaviorPage.xaml` file. Consider Figure 34, where a number is highlighted in red and considered invalid.

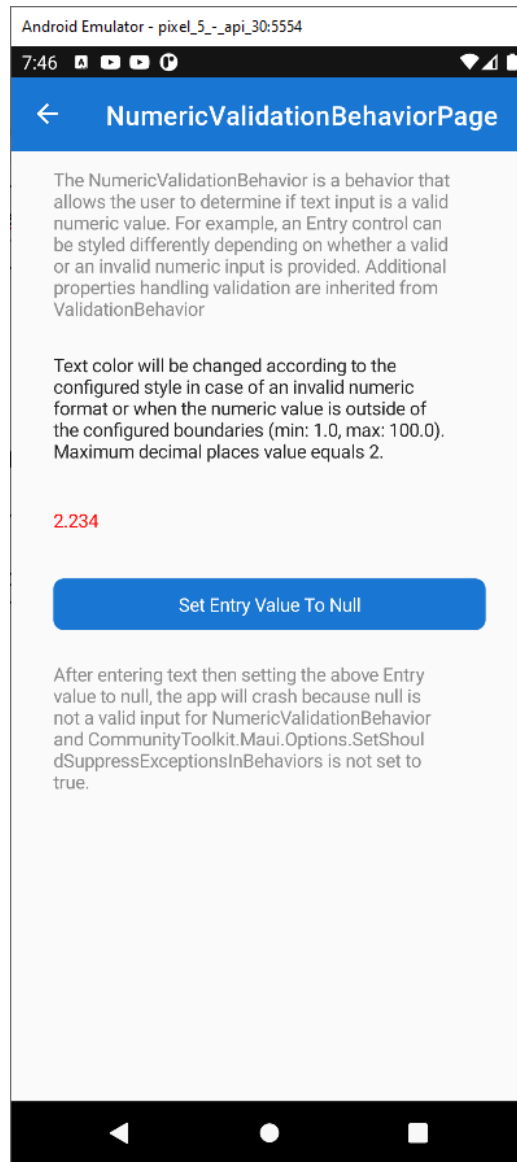


Figure 34: Validating Numbers

The following XAML puts the number validation in place.

```
<Entry Placeholder="Number">
  <Entry.Behaviors>
    <mct: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 these 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 (URIs)

Uniform resource identifiers (also referred to as URIs) are typically used to represent web addresses. A URI is made of the protocol (e.g., `https://`) and the address (e.g., `www.microsoft.com`). Validating URIs is another extremely common requirement in many apps, and the .NET MAUI 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 `Pages\Behaviors\UriValidationBehaviorPage.xaml` file. The following is the XAML sample code.

```
<Entry Placeholder="UriKind: Absolute">
    <Entry.Behaviors>
        <mct:UriValidationBehavior UriKind="Absolute"
            Flags="ValidateOnValueChanged"
            ValidStyle="{StaticResource ValidEntryStyle}"
            InvalidStyle="{StaticResource InvalidEntryStyle}"/>
    </Entry.Behaviors>
</Entry>
<Entry Placeholder="UriKind: Relative">
    <Entry.Behaviors>
        <mct:UriValidationBehavior UriKind="Relative"
            Flags="ValidateOnValueChanged"
            ValidStyle="{StaticResource ValidEntryStyle}"
            InvalidStyle="{StaticResource InvalidEntryStyle}"/>
    </Entry.Behaviors>
</Entry>
<Entry Placeholder="UriKind: RelativeOrAbsolute">
    <Entry.Behaviors>
        <mct:UriValidationBehavior UriKind="RelativeOrAbsolute"
            Flags="ValidateOnValueChanged"
            ValidStyle="{StaticResource ValidEntryStyle}"
            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 `Pages\Behaviors\RequiredStringValidationBehavior.xaml` file. The following XAML code demonstrates how easy it is to use.

```
<Entry Placeholder="Confirm Password">
    <Entry.Behaviors>
        <mct:RequiredStringValidationBehavior
            InvalidStyle="{StaticResource InvalidEntryStyle}"
            Flags="ValidateOnValueChanged"
            ValidStyle="{StaticResource ValidEntryStyle}"
            RequiredString="123" />
    </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 assign the **RequiredString** property with the string you want to match and set the **InvalidStyle** with the style you want to use for invalid strings. If you run the code, you will see the text highlighted in red if it does not match the text in the first **Entry** and in green if it does.

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>
        <mct:MultiValidationBehavior
            InvalidStyle="{StaticResource InvalidEntryStyle}" >
            <mct:CharactersValidationBehavior x:Name="digit"
                CharacterType="Digit" MinimumCharacterCount="1"
                mct:MultiValidationBehavior.Error="1 number" RegexPattern="" />
            <mct:CharactersValidationBehavior x:Name="upper"
                CharacterType="UppercaseLetter" MinimumCharacterCount="1"
                mct:MultiValidationBehavior.Error="1 upper case" RegexPattern="" />
            <mct:CharactersValidationBehavior x:Name="symbol"
                CharacterType="NonAlphanumericSymbol" MinimumCharacterCount="1"
                mct:MultiValidationBehavior.Error="1 symbol" RegexPattern="" />
        </mct:MultiValidationBehavior>
    </Entry.Behaviors>
</Entry>
```

```

        <mct:CharactersValidationBehavior x:Name="any"
CharacterType="Any" MinimumCharacterCount="8"
mct:MultiValidationBehavior.Error="8 characters" RegexPattern="" />
        </mct:MultiValidationBehavior>
    </Entry.Behaviors>
</Entry>

```



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

The Pages\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 Pages\Behaviors\SetFocusOnEntryCompleteBehaviorPage.xaml file. The following code snippet demonstrates how to move to the next **Entry** when the user finishes typing in the first one.

```

<Entry
    x:Name="Entry1"
    mct:SetFocusOnEntryCompletedBehavior.NextElement="{x:Reference Entry2}"
    Placeholder="Entry 1 (Tap `Next` or `Enter` or `Return` when finished)"
    ReturnType="Next" />

```

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. Different from other examples, this behavior in the sample project works with a **SearchBar** and its XAML code is as follows.

```

<SearchBar Placeholder="Start searching..."
    Margin="{StaticResource ContentPadding}">
    <SearchBar.Behaviors>
        <mct: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 decides that the user has really stopped typing, combined with a minimum number of characters the user must type (**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.

Mapping events to commands

With programming patterns like model-view-viewmodel (MVVM), views do not handle events for the user interaction directly (e.g., button clicks). Rather, they are data-bound to objects of type **Command**, defined inside a backing viewmodel. However, there are views that do not expose properties of type **Command** that can be bound to a viewmodel, so you need an alternative to map the view's event to a command defined inside a viewmodel. This is accomplished via the **EventToCommandBehavior**, demonstrated in the `Pages\Behaviors\EventToCommandBehaviorPage.xaml` file of the sample project. Code Listing 6 shows the full XAML code for the page.

Code Listing 6

```

<?xml version="1.0" encoding="utf-8" ?>
<pages:BasePage
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:pages="clr-namespace:CommunityToolkit.Maui.Sample.Pages"
    xmlns:mct="http://schemas.microsoft.com/dotnet/2022/maui/toolkit"
    xmlns:vm="clr-
namespace:CommunityToolkit.Maui.Sample.ViewModels.Behaviors"
    x:Class="CommunityToolkit.Maui.Sample.Pages.Behaviors.EventToCommandBeh
aviorPage"
    x:TypeArguments="vm:EventToCommandBehaviorViewModel"
    x:DataType="vm:EventToCommandBehaviorViewModel">

    <VerticalStackLayout Spacing="20">
        <Label Text="The EventToCommandBehavior is a behavior that allows t
he user to invoke an ICommand through an event. It is designed to associate
Commands to events exposed by controls that were not designed to support C
ommands. It allows you to map any arbitrary event on a control to a Command
." TextColor="{StaticResource NormalLabelTextColor}" />

        <Label Text="This sample demonstrates how to use EventToCommandBeha
vior. Here we observe Clicked event of the button and trigger IncrementComm
and from ViewModel." />
    
```

```

<Label Text="{Binding ClickCount, StringFormat='{0} clicks'}" />

<Button Text="Click Me"
        TextColor="White"
        BackgroundColor="{StaticResource NormalButtonBackground
Color}">
    <Button.Behaviors>
        <mct:EventToCommandBehavior
            EventName="Clicked"
            Command="{Binding IncrementCommand}" />
    </Button.Behaviors>
</Button>
</VerticalStackLayout>
</pages:BasePage>

```

The backing viewmodel 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 6, 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 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 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>
        <mct: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 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 .NET MAUI Community Toolkit provides two converters that work in combination with the **EventToCommandBehavior** class and that allow for retrieving the appropriate event arguments. They are demonstrated in the `Pages\Converters\ItemTappedEventArgsPage.xaml` and `Pages\Converters\ItemSelectedEventArgsPage.xaml` files, respectively, and they work very similarly. Both assign their result to the **EventArgsConverter** property of the **EventToCommandBehavior** instance, and they work as follows.

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

...

<ListView.Behaviors>
  <mct: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 it does, 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. That way, the **ItemSelectedEventArgsConverter** is also the same, and only the returned object is different.

Setting the tint color of an image

The MAUI Community Toolkit introduces a behavior called **IconTintColorBehavior**, which allows for changing the tint color of an image.



Note: *At the time this ebook was written, this behavior does not work on Windows.*

Images should represent monochrome icons like those you would use on the navigation bar of your app, for example. In the sample code, you will be able to see that the behavior is applied to both **Image** and **ImageButton** views, with red and green colors. Figure 35 shows the result.

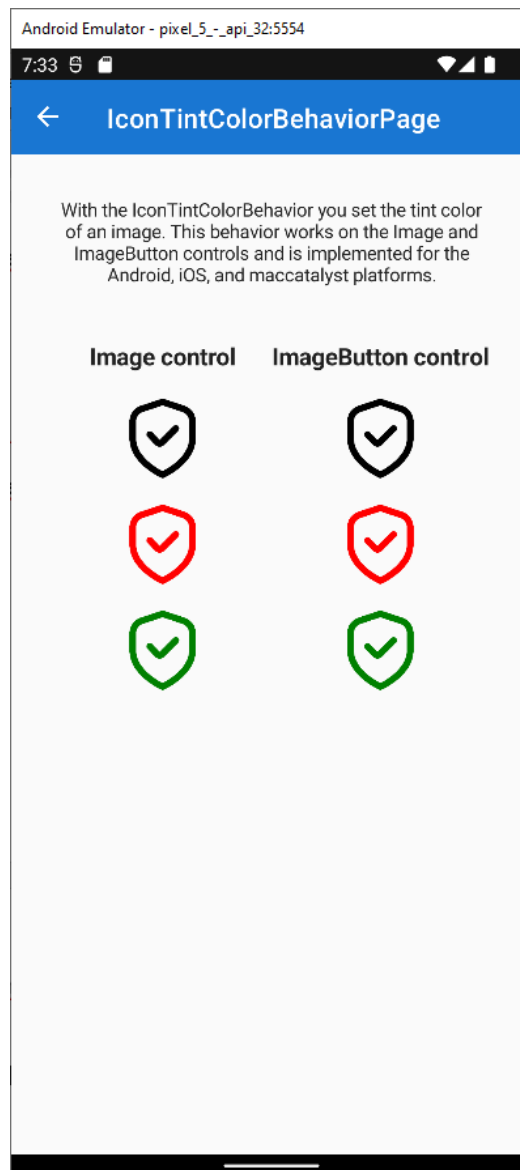


Figure 35: Setting the Tint Color of an Image

The following is an excerpt that applies the **Green** color.

```
<Image Grid.Row="4" Source="shield.png">
  <Image.Behaviors>
    <mct:IconTintColorBehavior TintColor="Green" />
  </Image.Behaviors>
</Image>

...

<ImageButton Grid.Row="4" Grid.Column="1" Source="shield.png">
  <ImageButton.Behaviors>
    <mct:IconTintColorBehavior TintColor="Green" />
  </ImageButton.Behaviors>
</ImageButton>
```

This behavior is very useful when you have icons representing states that change during the application lifecycle.

Easily defining input masks

Sometimes you might need to force the user to input only values matching a specific pattern. The MAUI Community Toolkit simplifies this via the **MaskedBehavior**. If you look at Figure 36, you can see how the user input always matches the pattern described in the various labels.

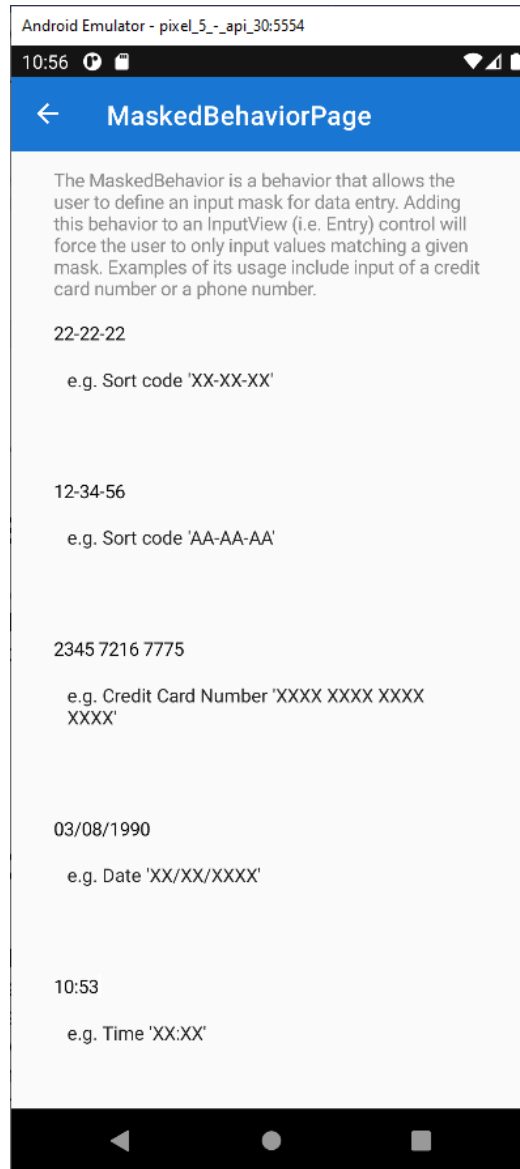


Figure 36: Forcing User Input to Match Patterns

You can try this yourself by running the sample application and seeing how, as you type, your input is formatted based on the pattern specified in each instance of the **MaskedBehavior**. The way you use it is very simple. For example, consider the code for the time pattern shown in Figure 36.

```
<Entry Keyboard="Numeric" Placeholder="hh:mm">
  <Entry.Behaviors>
    <mct:MaskedBehavior Mask="XX:XX" />
  </Entry.Behaviors>
</Entry>
```

You assign an instance of the **MaskedBehavior** object to a view derived from **InputView**, such as the **Entry**. Then you assign the **Mask** property with the pattern you want the user input to match. You can define more complex patterns for things like credit card numbers with the following code.

```
<Entry Keyboard="Numeric">
  <Entry.Behaviors>
    <mct:MaskedBehavior Mask="XXXX XXXX XXXX XXXX" />
  </Entry.Behaviors>
</Entry>
```

By default, the **MaskedBehavior** recognizes the **X** letter as the masked character. If you want to use a different one, you can specify the **UnmaskedCharacter** property as follows.

```
<Entry Keyboard="Numeric">
  <Entry.Behaviors>
    <mct:MaskedBehavior Mask="AA-AA-AA" UnmaskedCharacter="A" />
  </Entry.Behaviors>
</Entry>
```

This behavior is extremely useful and certainly time-saving in all those scenarios where the user input should have a fixed format and a specific number of characters.

Chapter summary

Data validation is required in any kind of application, and mobile apps are no exception. The .NET MAUI Community Toolkit exposes several reusable behaviors that make it quick and easy to validate strings without the need to implement custom logic. In fact, you can quickly validate email addresses, URIs, characters in a string, numbers, and string equality, and you can even perform multiple validations concurrently. Data is often related to an important programming pattern such as MVVM, and the objects discussed in this chapter extend the development possibilities when using MVVM.

Chapter 6 Enhancing Object Power with Extensions

Some of the features discussed in the previous chapters are empowered by new extension methods implemented by the .NET MAUI Community Toolkit. Such extension methods can also be used outside of the objects provided by the Toolkit and leveraged in your own scenarios. More specifically, you can animate and convert colors, and you can make the MAUI engine properly handle your views. These possibilities are all covered in this chapter.

Summarizing extensions

Among others, the .NET MAUI Community Toolkit exposes three classes that define specific extension methods. These classes are referred to as *extensions*, and Table 7 summarizes them.

Table 7: Reusable Extensions

| Extension | Description |
|------------------------------------|---|
| ColorAnimationExtensions | Expose extension methods that allow for animating properties of type Color . |
| ColorConversionExtensions | Expose extension methods that convert an object of type Color into another Color or another type. |
| ServiceCollectionExtensions | Expose extension methods that make it simpler to add views and their associated viewmodels to the app's service collection. |

The next sections describe all the extension methods with suggestions about their use cases.

Working with colors

Two classes allow for working with colors: **ColorAnimationExtensions** and **ColorConversionExtensions**. These are described in the next sections.

Animating colors

The `ColorAnimationExtensions` class exposes two extension methods:

- **BackgroundColorTo**: This method animates the **BackgroundColor** property of those views where it is available, such as (but not limited to) **Grid** and **Button**.
- **TextColorTo**: This method animates the **TextColor** property of those views where it is available, such as **Entry** and **Label**.

Both methods require at least one argument, which is the target color for the animation. However, they share a full list of parameters described in Table 8.

Table 8: Common Method Parameters

| Parameter | Type | Description |
|---------------|---------------|--|
| color | Color | The target color for the animation. |
| rate | uint | The time between frames of the animation in milliseconds. If omitted, the default is 16. |
| length | uint | The duration of the animation in milliseconds. If omitted, the default is 250. |
| easing | Easing | The easing function used for the animation. The default is null . |

Before using these methods, you need the following **using** directive.

```
using CommunityToolkit.Maui.Extensions;
```

For example, you could animate the background of a **Label** as follows.

```
var label = new Label { BackgroundColor = Colors.White };  
await label.BackgroundColorTo(Colors.Red);
```

This code will animate the background color with the default values described in Table 8. You could then further customize the animation like in the following example.

```
await label.BackgroundColorTo(Colors.Red, 10, 180, Easing.Linear);
```

With this code, the time between frames is 10 milliseconds, the animation duration is 180 milliseconds, and the type of animation is the **Linear** one from the **Easing** collection. The sample project provides the `ColorAnimationExtensionsPage.xaml` file under the `Pages\Extensions` folder. If you run the example and start the animations, you will get a result similar to Figure 37.

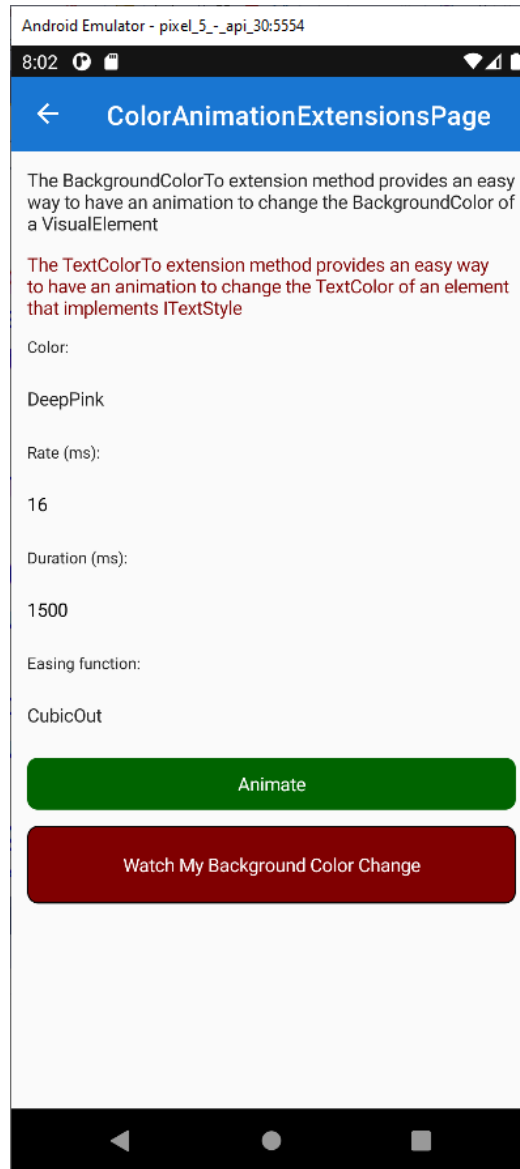


Figure 37: Animating Colors

These methods are very useful when you need to capture the user's attention, or when you simply want to provide a nice behavior to your user interface.

Converting colors

In [Chapter 4, “Saving Time with Reusable Converters,”](#) you saw how to convert one color to another color or one color to another type via value converters with data-binding scenarios. However, converting colors can also be useful outside of data binding. The value converters you saw in Chapter 4 all rely on separate methods that perform the actual conversions and are all defined inside the static **ColorConversionExtensions** class, located inside the `ColorConversionExtensions.shared.cs` file of the Toolkit. For example, if you want to convert a **Color** into a string that contains the RGB representation of the color, you can invoke an extension method called **ToRgbString**, as follows.

```
string resultString = Colors.Green.ToRgbString();
```

The extension method definition is:

```
public static string ToRgbString(this Color color)
{
    ArgumentNullException.ThrowIfNull(color);
    return $"RGB({color.GetByteRed()}, {color.GetByteGreen()}, {color.
        GetByteBlue()})";
}
```



Tip: The *this* keyword in the parameter list identifies the definition of an extension method, together with the *static* modifier.

The `GetByteRed`, `GetByteGreen`, and `GetByteBlue` methods are defined in the same file and their purpose is to retrieve the color's components. Their definition is:

```
public static byte GetByteRed(this Color color)
{
    ArgumentNullException.ThrowIfNull(color);
    return ToByte(color.Red * 255);
}


public static byte GetByteGreen(this Color color)
{
    ArgumentNullException.ThrowIfNull(color);
    return ToByte(color.Green * 255);
}

public static byte GetByteBlue(this Color color)
{
    ArgumentNullException.ThrowIfNull(color);
    return ToByte(color.Blue * 255);
}
```

The **ToByte** method, invoked by several conversion methods, is defined as follows.

```
static byte ToByte(float input)
{
    var clampedInput = Math.Clamp(input, 0, 255);
    return (byte)Math.Round(clampedInput);
}
```

The **Math.Clamp** method returns a value clamped to the original value including the specified minimum and maximum ranges. The result is rounded via **Math.Round** and then converted into a **byte** value.

 **Tip:** In order to use conversion methods, you need the following directive: using *CommunityToolkit.Maui.Core.Extensions*;

The list of available methods is very long, so a good idea is grouping methods by return type and purpose. Table 9 summarizes methods that return another **Color** as the result of the conversion or **bool** as the result of a color check.

Table 9: Methods for Conversion to Color

| Method name | Return type | Description |
|------------------------------|--------------|---|
| ToBlackOrWhite | Color | Convert the color to the closest monochrome color between Color.Black and Color.White . |
| ToBlackOrWhiteForText | Color | If the starting color is determined to be dark for the human eye, convert the color to Color.Black , otherwise to Color.White . |
| ToGrayScale | Color | Convert the color to a grayscale color. |
| ToInverseColor | Color | Return the inverse color. |
| IsDark | bool | Return true if the specified color is determined to be dark. |
| IsDarkForTheEye | bool | Return true if the specified color is determined to be dark for the human eye. |

All the methods in Table 9 (except for **IsDark** and **IsDarkForTheEye**) work in the same way: they are invoked over a color instance, and they return another **Color** object. For example, you can easily retrieve the inverse color for the specified **Color** object as follows.

```
Color newColor = Colors.Red.ToInverseColor();
```

In this example, **newColor** receives zero for red, 1 for green, 1 for blue, and 1 for alpha. Another set of methods is used to retrieve individual components from a color. These methods are summarized in Table 10.

Table 10: Methods that Return Components of a Color

| Method name | Return type | Description |
|---------------------------|---------------|---|
| GetByteRed | byte | Returns the red component of a color. |
| GetByteGreen | byte | Returns the green component of a color. |
| GetByteBlue | byte | Returns the blue component of a color. |
| GetDegreeHue | double | Returns the hue component of a color, with a value between 0 and 360. |
| GetPercentCyan | float | Returns the percentage of cyan in a color, with a value between 0 and 1. |
| GetPercentMagenta | float | Returns the percentage of magenta in a color, with a value between 0 and 1. |
| GetPercentYellow | float | Returns the percentage of yellow in a color, with a value between 0 and 1. |
| GetPercentBlackKey | float | Returns the percentage of the black key in a color, with a value between 0 and 1. |
| GetByteAlpha | byte | Returns the alpha-blending component in a color, with a value between 0 and 1. |



Tip: For methods that return *byte*, the minimum and maximum values are between 0 and 255, which is the range for the *byte* type.

In the following line of code, you can see how to return a component from a color.

```
byte blue = Colors.Maroon.GetByteBlue();
```

The other methods work the same, obviously with their specific return type. Another group of methods returns formatted **string** objects with color components, as represented in Table 11.

Table 11: Methods for Retrieving Formatted Strings

| Method name | Description |
|----------------------|---|
| ToCmykaString | Returns a string containing the percentage of cyan, magenta, yellow, black key, and alpha from a color. The alpha value is between 0 and 1. The other values are between 0 and 100. |
| ToCmykString | Returns a string containing the percentage of cyan, magenta, yellow, and black key from a color. Values are between 0 and 100. |
| ToHslaString | Returns a string containing the amount of hue, saturation, lightness, and alpha from a color. Hue is between 0 and 360, alpha is between 0 and 1, and the others are between 0 and 100. |
| ToHslString | Returns a string containing the amount of hue, saturation, and lightness from a color. Hue is between 0 and 360, and the other values are between 0 and 100. |
| ToRgbaString | Returns a string containing the red, green, blue, and alpha values from a color. Alpha is between 0 and 1, and the other values are between 0 and 255. |
| ToRgbString | Returns a string containing the red, green, and blue values from a color. Values are between 0 and 255. |

Methods in this group also work in the same way. For example, you can retrieve the RGB with alpha components as follows.

```
Console.WriteLine(Colors.Azure.ToRgbaString());
```

The result of the invocation is the following.

```
> RGBA(240,255,255,1)
```



Tip: Remember that invoking `Console.WriteLine` in a .NET MAUI project will print the output in the Output window.

The last group of extension methods allow for generating a new **Color** from an existing one, to which a different component is applied. Table 12 summarizes these methods.

Table 12: Methods for Color Conversion

| Method name | Return type | Description |
|---------------------|--------------|---|
| WithRed | Color | Replaces the red component of a color with a value between 0 and 255. |
| WithGreen | Color | Replaces the green component of a color with a value between 0 and 255. |
| WithBlue | Color | Replaces the blue component of a color with a value between 0 and 255. |
| WithCyan | Color | Replaces the cyan component of a color with a value between 0 and 1. |
| WithMagenta | Color | Replaces the magenta component of a color with a value between 0 and 1. |
| WithYellow | Color | Replaces the yellow component of a color with a value between 0 and 1. |
| WithBlackKey | Color | Replaces the black key component of a color with a value between 0 and 1. |

For example, the following code adds the maximum value for the yellow component and creates a derived color.

```
Color newColor = Colors.PaleVioletRed.WithYellow(1);
```

Methods for color conversion can be very useful in a variety of scenarios, and the library of methods provided by the .NET MAUI Community Toolkit will cover most of your needs.

Working with MAUI services

The .NET MAUI Community Toolkit also provides interesting extensions to methods that you can use if you work with the dependency injection and inversion of control patterns. Such methods simplify the way you register views and their associated viewmodels within the .NET MAUI [IServiceCollection](#), a type handled by the **MauiApp** class that registers the services that will be used across the app.

For a better understanding, consider the **AddScoped<TView>**, **AddSingleton<TView>**, and **AddTransient<TView>** methods. These are already part of the MAUI codebase, and they respectively allow for adding a scoped view, a singleton view, and a transient view to the collection of services. However, with the Toolkit extension, you now have the option to also register a viewmodel along with its view, as follows.

```
builder.Services.AddScoped<HomePage, HomePageViewModel>();
```

```
builder.Services.AddSingleton<HomePage, HomePageViewModel>();
```

```
builder.Services.AddTransient<HomePage, HomePageViewModel>();
```

Similarly, there are methods in MAUI that allow for registering a view so that the **Shell** can navigate to the view via the specified route. These methods are **AddScopedWithShellRoute<TView>**, **AddSingletonWithShellRoute<TView>**, and **AddTransientWithShellRoute<TView>**, and they have the same purpose as the previously mentioned methods, but with the possibility to specify a shell route. With the Community Toolkit, you can now associate a viewmodel as follows.

```
builder.Services.AddScopedWithShellRoute<HomePage, HomePageViewModel>();
```

```
builder.Services.AddSingletonWithShellRoute<HomePage, HomePageViewModel>();
```

```
builder.Services.AddTransientWithShellRoute<HomePage, HomePageViewModel>();
```

Though these extension methods become very useful only with dependency injection and inversion of control scenarios, it is worth mentioning that many MVVM third-party frameworks rely on both patterns. Syncfusion has a very interesting [article](#) about dependency injection in MAUI that is a nice supplement to this section.

Chapter summary

The .NET MAUI Community Toolkit enhances your development experience with a library of extension methods defined in three public classes. Methods exposed by the **ColorAnimationExtensions** class allow for animating objects of type **Color**. Methods exposed by the **ColorConversionExtensions** class allow for converting **Color** objects into other colors or other types, depending on the value they retrieve. Methods exposed by the **ServiceCollectionExtensions** class enhance the way you associate viewmodels with views in a dependency injection scenario. As you have seen, this chapter is mostly based on C# code rather than XAML and this is what you will also see in the next chapter, where you will learn how to leverage fluent C# APIs to define the user interface of your apps.

Chapter 7 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 also be 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 an API called C# Markup, which provides a new, fluent syntax you can use to define views, their properties, and their interaction behaviors more easily and in a more elegant way. This API is offered by the .NET MAUI Community Toolkit and is the topic of this chapter.



Note: C# Markup for mobile development was first introduced with the *Xamarin Community Toolkit*. If you had a chance to work with it, you will be immediately familiar with C# Markup in the .NET MAUI Community Toolkit.

Introducing C# Markup

[C# Markup](#) is a set of fluent APIs that simplifies 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. In the ecosystem of the .NET MAUI Community Toolkit, C# Markup is provided by a separate project available on [GitHub](#) that produces the `CommunityToolkit.Maui.Markup` NuGet package. You will need to install this package in your projects before using this feature. It is also worth mentioning that C# Markup is an independent feature, which means that you do not need to install the .NET MAUI Community Toolkit NuGet package also.



Note: C# Markup is the only feature from the .NET MAUI Community Toolkit that is not included in the official sample project, so a new project will be created in the next sections. In addition, the focus of this chapter is using C# Markup, not walking through the library design and implementation.

Before creating a project with more detailed explanations about C# Markup, it is a good idea to discuss some short examples. In the following code, the first `Label` is defined in C# using the regular syntax, whereas the second `Label` is defined using C# Markup.

```
var label1 = new Label { FontAttributes = FontAttributes.Bold,
                        HorizontalOptions = LayoutOptions.Center,
                        HorizontalTextAlignment = TextAlignment.Center,
                        };

var label2 = new Label().Bold().Center().TextCenter();
```

As a general rule, C# Markup is based on extension methods that represent property assignments. As you can see, with this approach the code you write is shorter and cleaner, yet easy to understand. In the case of properties that support multiple values from an enumeration, such as **HorizontalAlignment** of type **TextAlignment**, there are extension methods that match the enumeration, such as **TextStart** and **TextEnd**. In the next example, you can see how a **CollectionView** is defined at runtime and how its **ItemsSource** property is bound to some data.

```
var dataView = new CollectionView();  
dataView.SetBinding(CollectionView.ItemsSourceProperty,  
    new Binding(nameof(ViewModel.ObjectCollection)));
```

Data binding is assigned via the **SetBinding** method, which takes the target dependency property as the first argument, and an object of type **Binding** that points to the data source as the second argument. With C# Markup, the code could be rewritten as follows.

```
var dataView = new CollectionView().Bind(CollectionView.ItemsSourceProperty,  
    nameof(ViewModel.RegistrationCode));
```

As you can see, the expression is simplified by the **Bind** extension method, which also avoids the need to explicitly declare an instance of the **Binding** type as the second argument. The benefits of using C# Markup to define some user interface in code-behind should be clearer now, but they will be even clearer with a sample project.

Creating a sample project



Note: *This ebook requires you to already have a basic knowledge of .NET MAUI, including how to create MAUI projects, so this will not be explained here.*



Note: *The complete code for the sample is available on [GitHub](#).*

In Visual Studio 2022, create a new MAUI app project with a name of your choice. If you want to be consistent with the [companion project](#), you can use MarkupExample. When the project is ready, the first thing you need to do is install the CommunityToolkit.Maui.Markup package from NuGet (see Figure 38).

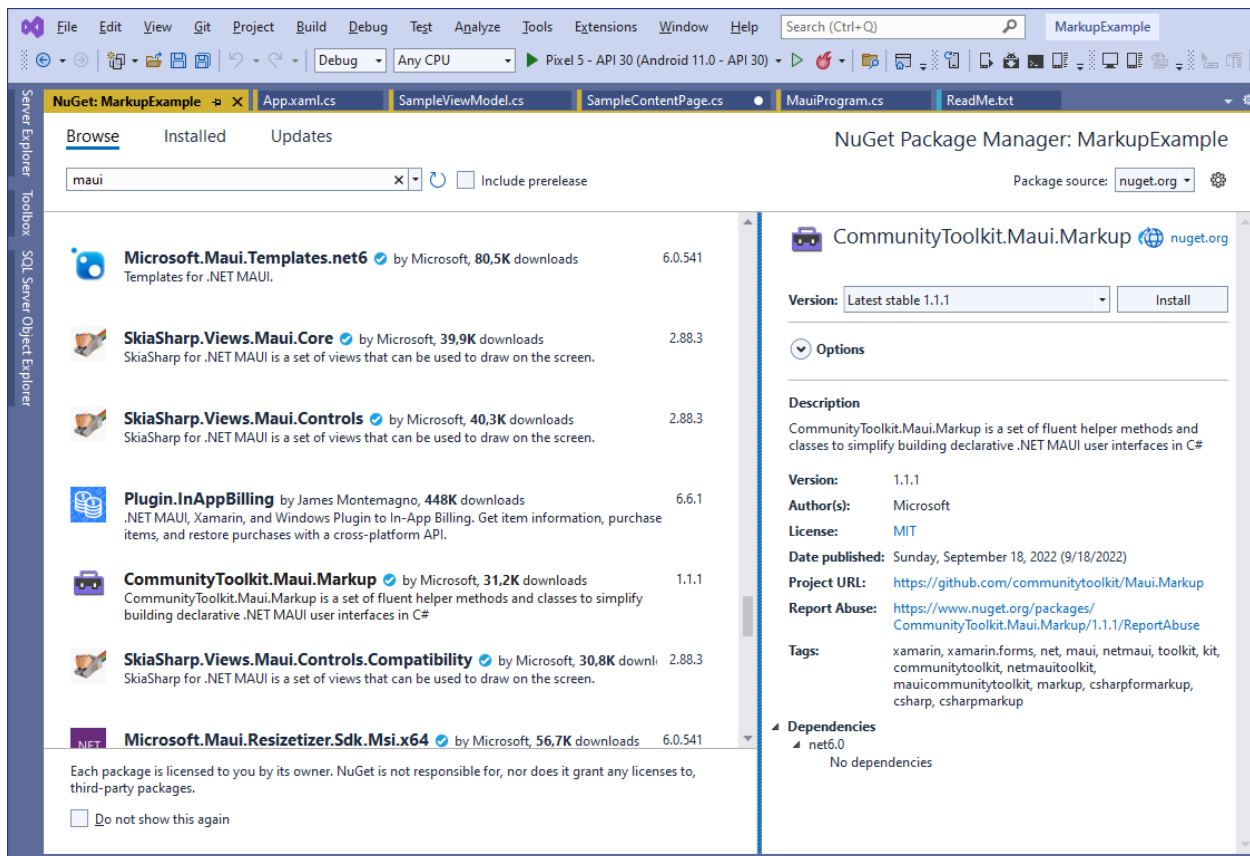


Figure 38: Installing the Required NuGet Packages

Once the package has been installed, you need to enable the Community Toolkit. This is accomplished by invoking the `UseMauiCommunityToolkitMarkup` method in the `MauiProgram` class, as follows.

```
var builder = MauiApp.CreateBuilder();
builder.UseMauiApp<App>().UseMauiCommunityToolkitMarkup();
```

This is an extension method that you invoke on the instance of the `MauiAppBuilder` class, responsible for enabling and handling services used in the app. The final step to set up the sample project is adding a new `ContentPage` based on C# only. To accomplish this, right-click the project name in Solution Explorer and then select **Add New Item**. In the Add New Item dialog (see Figure 39), select the **.NET MAUI ContentPage (C#)** and assign `SampleContentPage.cs` as the file name.

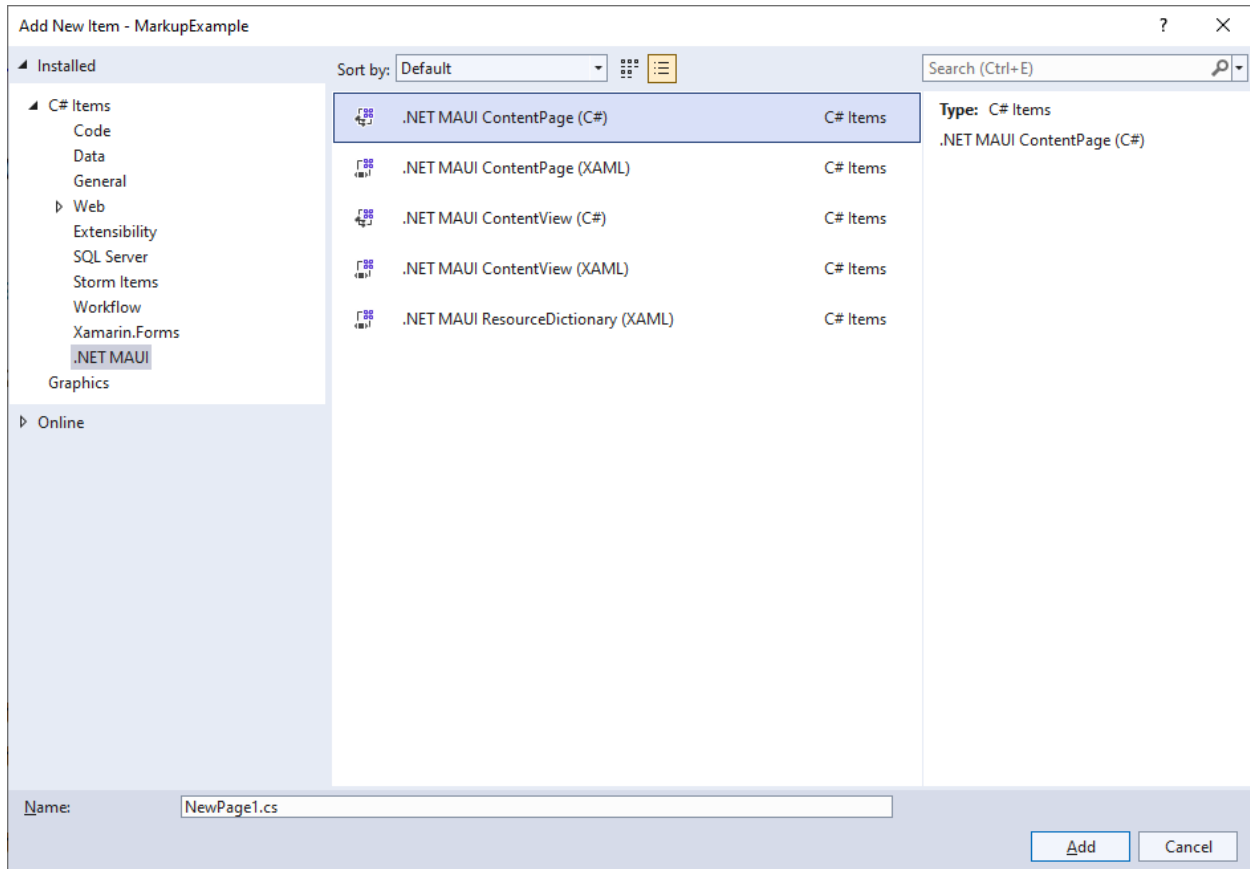


Figure 39: Adding a C# *ContentPage*

Click **Add** when ready. When the new file is added, you can start writing some user interface with C# Markup.

Using C# Markup

The sample project discussed in this chapter is based on an [example](#) provided by the official documentation. However, some edits are required because the example does not show the viewmodel definition and because some margins also need to be adjusted. The final result to be achieved is very simple, but it gives you enough understanding of the power of C# Markup. It's shown in Figure 40.

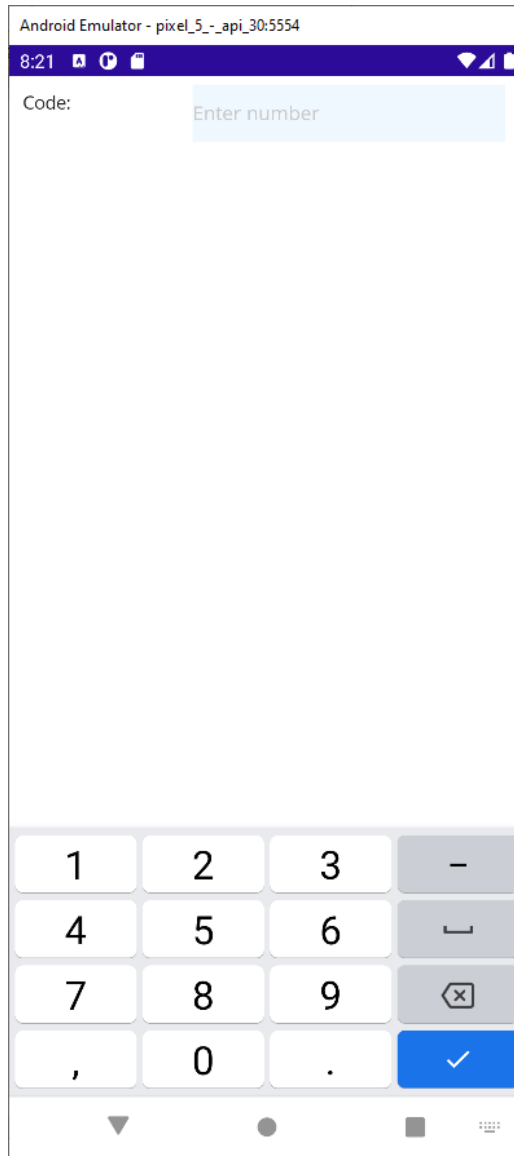


Figure 40: The User Interface Created with C# Markup

The example binds an **Entry** to a property called **RegistrationCode**, defined inside a class called **SampleViewModel1**. In order to make the official example work, you could add the following simple class to the project.

```
namespace MarkupExample
{
    public class SampleViewModel1
    {
        public string RegistrationCode { get; set; }
    }
}
```

In the **SampleContentPage** class, it is first necessary to add the following property and its instance.

```
private SampleViewModel ViewModel { get; set; }

public SampleContentPage()
{
    ViewModel = new SampleViewModel();
}
```

The user interface is defined inside the class's constructor, but this is only to make the example easier. Going step by step, the first part of the code defines a **Grid** as the content of the page.

```
Content = new Grid
{
    RowDefinitions = Rows.Define(
        (Row.TextEntry, 36)),

    ColumnDefinitions = Columns.Define(
        (Column.Description, Star),
        (Column.Input, Stars(2))),
}
```

The **Define** extension method on the **Rows** and **Columns** collections simplifies the way you declare rows and columns. In this case, there is one row and two columns. The position is expressed via named values of the **Row** and **Column** enumerations, defined as follows.

```
enum Row { TextEntry }
enum Column { Description, Input }
```

The reason for this is just to use some named values that are easier to understand, instead of plain integers. Continuing with the **Grid** definition, the code declares its **Children** object and starts with these lines:

```
Children =
{
    new Label()
        .Text("Code:")
        .Row(Row.TextEntry).Column(Column.Description),
}
```

The **Label** is fully defined via C# Markup, with the **Text**, **Row**, and **Column** extension methods that assign the text and the position in the **Grid** (row and column) respectively. The second part of the user interface is the **Entry** and is declared as follows.

```
new Entry
{
    Keyboard = Keyboard.Numeric,
    BackgroundColor = Colors.AliceBlue,
}.Row(Row.TextEntry).Column(Column.Input)
    .FontSize(15).Placeholder("Enter number").TextColor(Colors.Black)
    .Height(44).Margin(5, 5).Bind(Entry.TextProperty,
    nameof(ViewModel.RegistrationCode))
```

Notice how properties for which there are no C# Markup methods are still assigned the usual way, and then methods are invoked. **Row** and **Column** set the position inside the **Grid**; **FontSize** sets the size of the typeface; **Placeholder** assigns the placeholder text; **TextColor** sets the foreground color; **Height** assigns the height for the **Entry**; **Margin** sets the horizontal and vertical margins, and finally **Bind** establishes data binding between the view and the data source.



Tip: The *Margin* method as two overloads: the one used in the current example takes two double values as the parameters, for the horizontal and vertical margins. The second overload takes an object of type *Thickness*, like you would do in XAML.

The closing lines for the sample class are the following.

```
    }  
    }.Margin(10);  
}
```

Notice how a margin is applied to the **Grid** via C# Markup. Before running the project, you need to change the startup page in the **App.xaml.cs** file by assigning the **MainPage** property as follows:

```
MainPage = new SampleContentPage();
```

If you now run the application, you will get the result shown in Figure 40.

Working with styles

Views expose the **Style** extension method, which allows for assigning a style via C# Markup. Suppose you want to create a style that renders labels as hyperlinks. If you look at the [Styles.cs file](#) of the companion sample project, you will see how to declare styles with the C# Markup syntax. For example, consider the style defined in Code Listing 7.

Code Listing 7

```
public class Styles  
{  
    static Style<Label> link;  
  
    public static Style<Label> Link => link ??= new Style<Label>(  
        (Span.FontSizeProperty, 14),  
        (Span.TextColorProperty, Colors.CornflowerBlue),  
        (Span.TextDecorationsProperty, TextDecorations.Underline)  
    );  
}
```

C# Markup allows for declaring styles by defining static properties of type **Style<T>**, where **T** is the target type for the style. In the previous code, the style is applied to **Label** objects and it simplifies the way properties are assigned with the desired value. You then apply a style by invoking the **Style** extension method as follows.

```
new Label().Style(Styles.Link)
```

More C# Markup

Including the full reference for C# Markup in an ebook of the *Succinctly* series is not possible, so, as I mentioned at the beginning of this chapter, you can read the [official documentation](#) to get the complete information. However, as a developer, you can also look into the [source code](#) of the `CommunityToolkit.Maui.Markup` library. 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 fluent APIs that simplify the way developers can create the user interface in C#. New extension methods and a syntax based on lambda expressions provide a more convenient and elegant way to design views and assign their properties, as well as implement their interaction behaviors via events. These capabilities are now available via the `CommunityToolkit.Maui.Markup` NuGet package and will really boost the way you create the UI in C#, especially when you need to add views at runtime.