Contents

Xamarin Community Toolkit

3/5/2021 • 2 minutes to read • Edit Online

The Xamarin Community Toolkit is a collection of reusable elements for mobile development with Xamarin.Forms, including animations, behaviors, converters, effects, and helpers. It simplifies and demonstrates common developer tasks when building iOS, Android, macOS, WPF and Universal Windows Platform (UWP) apps using Xamarin.Forms.

The Xamarin Community Toolkit is available as a Visual Studio NuGet package for new or existing Xamarin.Forms projects.

You can also preview the capabilities of the toolkit by running the sample app available in the Xamarin Community Toolkit repo.

Feel free to browse the documentation using the table of contents on the left side of this page.

Get started

Follow the Getting started guide to install the **Xamarin.CommunityToolkit** NuGet package into your existing or new Xamarin.Forms, Android, iOS, or UWP projects.

Open source

The Xamarin Community Toolkit is built as an open source project hosted on GitHub by the community.

Troubleshooting

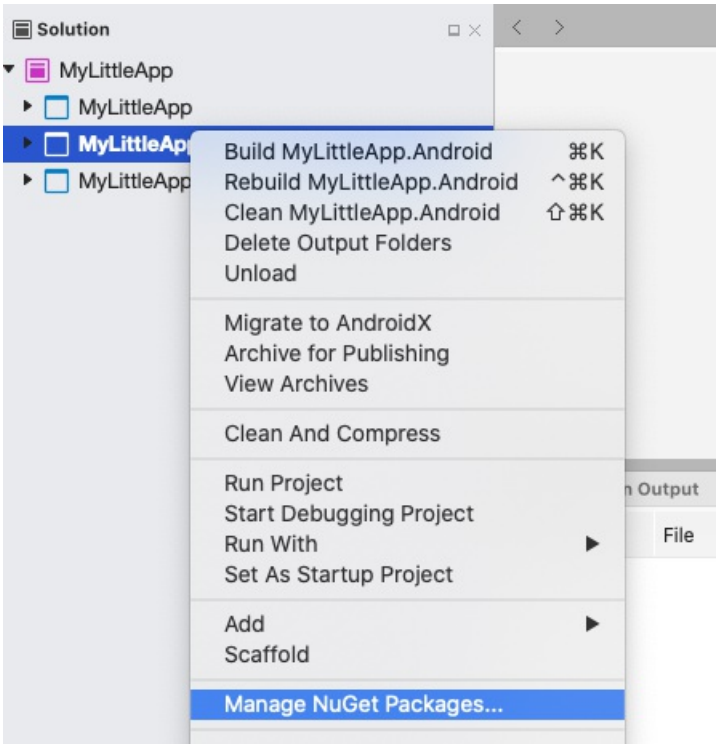
Find help if you are running into issues.

Getting Started with the Xamarin Community Toolkit

3/5/2021 • 2 minutes to read • Edit Online

The toolkit is available as a NuGet package that can be added to any existing or new project using Visual Studio.

1. Open an existing project, or create a new project using the Blank Forms App template.
2. In the Solution Explorer panel, right click on your project name and select **Manage NuGet Packages**. Search for **Xamarin.CommunityToolkit**, and choose the desired NuGet Package from the list.



3. To add the namespace to the toolkit:
   - In your C# page, add:

     ```
     using Xamarin.CommunityToolkit;
     ```

   - In your XAML page, add the namespace attribute:

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

4. Check out the rest of the documentation to learn more about implementing specific features.

Xamarin Show: Xamarin Community Toolkit introduction video

Other resources

Download the Xamarin Community Toolkit Sample App from the repository to see the controls in an actual app.

We recommend developers who are new to Xamarin.Forms to visit the Get started with Xamarin documentation.

Visit the Xamarin Community Toolkit GitHub Repository to see the current source code, what is coming next, and clone the repository. Community contributions are welcome!

Xamarin Community Toolkit CharactersValidationBehavior

3/5/2021 • 2 minutes to read • Edit Online

The CharactersValidationBehavior is a behavior that allows the user to validate text input depending on specified parameters. For example, an `Entry` control can be styled differently depending on whether a valid or an invalid text value is provided. This behavior includes built-in checks such as checking for a certain number of digits or alphanumeric characters. Additional properties handling validation are inherited from ValidationBehavior.

Syntax

```
<Entry>
    <Entry.Behaviors>
        <xct:CharactersValidationBehavior
            InvalidStyle="{StaticResource InvalidEntryStyle}"
            CharacterType="Digit"
            MaximumCharacterCount="10"
        />
    </Entry.Behaviors>
</Entry>
```

Properties

PROPERTY	TYPE	DESCRIPTION
CharacterType	CharacterType	Provides an enumerated value to use to set how to handle comparisons.
MaximumCharacterCount	int	The maximum length of the text input that's allowed.
MinimumCharacterCount	int	The minimum length of the text input that's allowed.

Sample

- CharactersValidationBehavior sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- CharactersValidationBehavior source code

Xamarin Community Toolkit EmailValidationBehavior

3/5/2021 • 2 minutes to read • Edit Online

The EmailValidationBehavior is a behavior that allows users to determine whether or not text input is a valid e-mail address. For example, an `Entry` control can be styled differently depending on whether a valid or an invalid e-mail address is provided. The validation is achieved through a regular expression that is used to verify whether or not the text input is a valid e-mail address. It can be overridden to customize the validation through the properties it inherits from ValidationBehavior.

Syntax

```
<Entry Placeholder="Email">
    <Entry.Behaviors>
        <xct:EmailValidationBehavior
            DecorationFlags="Trim"
            InvalidStyle="{StaticResource InvalidEntryStyle}"
        />
    </Entry.Behaviors>
</Entry>
```

Properties

PROPERTY	TYPE	DESCRIPTION
DefaultRegexPattern	string	The regular expression used to verify whether or not the text input is a valid e-mail address.

Sample

- EmailValidationBehavior sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- EmailValidationBehavior source code

Xamarin Community Toolkit EventToCommandBehavior

5/4/2021 • 2 minutes to read • Edit Online

The EventToCommandBehavior is a behavior that allows the user to invoke a Command through an event. It is designed to associate Commands to events exposed by controls that were not designed to support Commands. It allows you to map any arbitrary event on a control to a Command.

Syntax

This behavior can be used on any control that exposes events, such as a `Button`:

```
<Button.Behaviors>
    <xct:EventToCommandBehavior
        EventName="Clicked"
        Command="{Binding MyCustomCommand}" />
</Button.Behaviors>
```

When using this behavior with selection or tap events exposed by `ListView` an additional converter is required. This converter converts the event arguments to a command parameter which is then passed onto the Command. They are also available in the Xamarin Community Toolkit:

- ItemSelectedEventArgsConverter
- ItemTappedEventArgsConverter

Properties

PROPERTY	TYPE	DESCRIPTION
EventName	string	The name of the event that should be associated with a `Command`.
Command	ICommand	The `Command` that should be executed.
CommandParameter	object	An optional parameter to forward to the `Command`.
EventArgsConverter	IValueConverter	An optional `IValueConverter` that can be used to convert `EventArgs` values to values passed into the `Command`.

Sample

- EventToCommandBehavior sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- EventToCommandBehavior source code

Xamarin Community Toolkit ImpliedOrderGridBehavior

3/5/2021 • 2 minutes to read • Edit Online

The ImpliedOrderGridBehavior enables you to automatically assign a `Grid` row and column to a view based on the order the view is added to the `Grid`. You only need to setup the row and column definitions and then add children to the Grid. You may still assign `RowSpan` and `ColumnSpan` to views and their values will be taken into account when assigning a row and column to a view. If a view has a user defined row or column value it will be honored.

Logging and exceptions

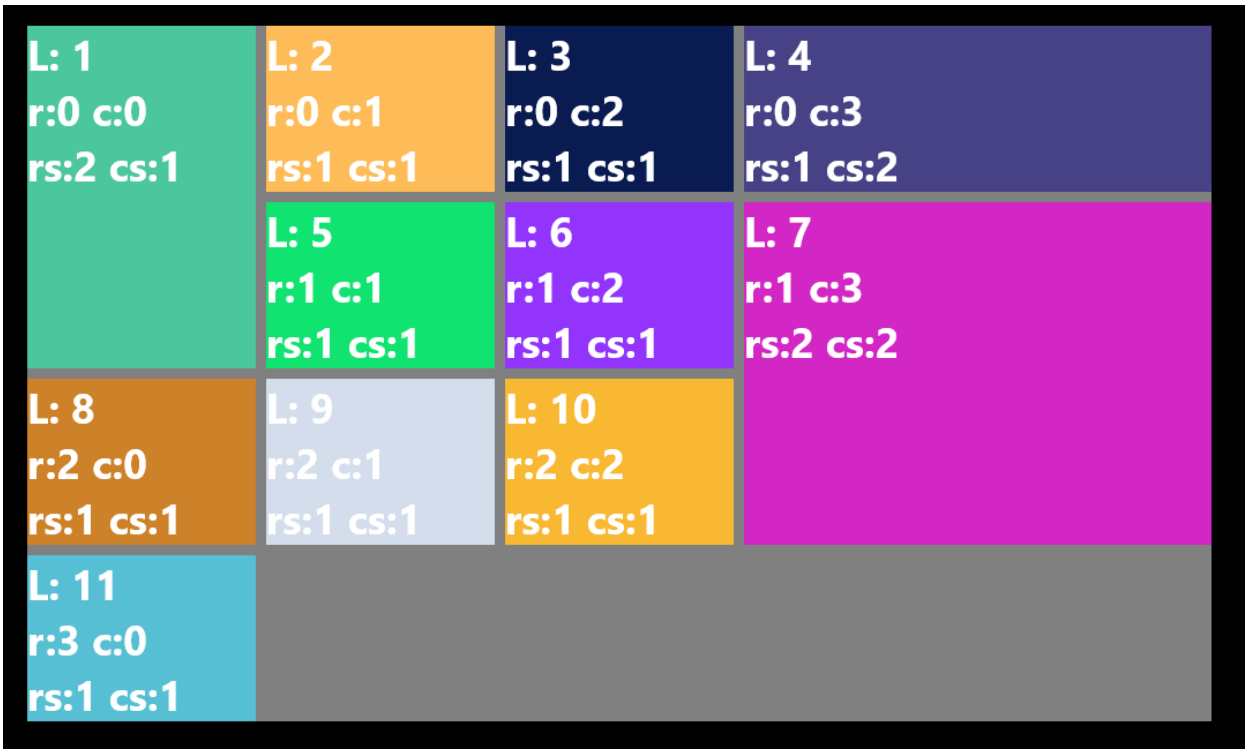
By default the behavior will log warnings for the following conditions:

- The number of children added to the Grid exceeds the number of available cells based on the row and column definitions.
- A child with a ColumnSpan value that exceeds the number of columns to it's right.
- A child with a RowSpan value that exceeds the number of rows below it.
- A child with a user assigned row and/or column is placed over an already placed child.

> **NOTE**
> You may optionally set the `ThrowOnLayoutWarning` property to `true`, which will raise exceptions instead of warnings.

Syntax

```
<Grid x:Name="TestGrid" Margin="30" BackgroundColor="Gray">
    <Grid.Behaviors>
    <autoLayoutGrid:ImpliedOrderGridBehavior />
    </Grid.Behaviors>

    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid .ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <!-- Row 0 -->
    <autoLayoutGrid:TestLabel Grid .RowSpan="2"/>
    <autoLayoutGrid:TestLabel />
    <autoLayoutGrid:TestLabel />
    <autoLayoutGrid:TestLabel Grid .ColumnSpan="2"/>

    <!-- Row 1 -->
    <autoLayoutGrid:TestLabel/>
    <autoLayoutGrid:TestLabel />
    <autoLayoutGrid:TestLabel Grid .ColumnSpan="2"
                              Grid .RowSpan="2" />

    <!-- Row 2 -->
    <autoLayoutGrid:TestLabel />
    <autoLayoutGrid:TestLabel />
    <autoLayoutGrid:TestLabel />

    <!-- Row 3 -->
    <autoLayoutGrid:TestLabel />

</Grid >
```



Properties

PROPERTY	TYPE	DESCRIPTION
ThrowOnLayoutWarning	bool	When true warnings will throw an exception instead of being logged. Defaults to false.

Sample

- ImpliedOrderGridBehavior sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- ImpliedOrderGridBehavior source code

Xamarin Community Toolkit MaskedBehavior

3/5/2021 • 2 minutes to read • Edit Online

The MaskedBehavior is a behavior that allows the user to define an input mask for data entry. Adding this behavior to an `Entry` control will force the user to only input values matching a given mask. Examples of its usage include input of a credit card number or a phone number.

Syntax

```
<Entry>
    <Entry.Behaviors>
        <xct:MaskedBehavior
            Mask="AA-AA-AA"
            UnMaskedCharacter="A"
        />
    </Entry.Behaviors>
</Entry>
```

Properties

PROPERTY	TYPE	DESCRIPTION
Mask	string	The mask that the input value needs to match.
UnMaskedCharacter	string	The placeholder character for when no input has been given yet.

Sample

- MaskedBehavior sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- MaskedBehavior source code

Xamarin Community Toolkit MaxLengthReachedBehavior

3/5/2021 • 2 minutes to read • Edit Online

The MaxLengthReachedBehavior is a behavior that allows the user to trigger an action when a user has reached the maximum length allowed on an `Entry`. It can either trigger a `Command` or an `event` depending on the user's preferred scenario.

Syntax

```
<<Entry
    Placeholder="Start typing until MaxLength is reached..."
    MaxLength="10">
    <Entry.Behaviors>
        <xct:MaxLengthReachedBehavior
            MaxLengthReached="MaxLengthReachedBehavior_MaxLengthReached"
            ShouldDismissKeyboardAutomatically="True"
        />
    </Entry.Behaviors>
</Entry>

<Entry
    Placeholder="Start typing until MaxLength is reached..."
    MaxLength="10">
    <Entry.Behaviors>
        <xct:MaxLengthReachedBehavior
            Command="{Binding MaxLengthReachedCommand}"
            ShouldDismissKeyboardAutomatically="False"
        />
    </Entry.Behaviors>
</Entry>
```

Properties

PROPERTY	TYPE	DESCRIPTION
Command	ICommand	The command to execute when the user has reached the maximum length.
ShouldDismissKeyboardAutomatically	int	Indicates whether or not the keyboard should be dismissed automatically.

Events

EVENT	DESCRIPTION
MaxLengthReached	The event to raise when the user has reached the maximum length.

Sample

- MaxLengthReachedBehavior sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- MaxLengthReachedBehavior source code

Xamarin Community Toolkit MultiValidationBehavior

3/5/2021 • 2 minutes to read • Edit Online

The MultiValidationBehavior is a behavior that allows the user to combine multiple validators to validate text input depending on specified parameters. For example, an `Entry` control can be styled differently depending on whether a valid or an invalid text input is provided. By allowing the user to chain multiple existing validators together, it offers a high degree of customizability when it comes to validation. Additional properties handling validation are inherited from ValidationBehavior.

Syntax

```
<Entry>
    <Entry.Behaviors>
        <xct:MultiValidationBehavior
            x:Name="MultiValidation"
            InvalidStyle="{StaticResource InvalidEntryStyle}">

            <xct:NumericValidationBehavior
                xct:MultiValidationBehavior.Error="NaN"
            />
            <xct:NumericValidationBehavior
                MinimumValue="-10"
                xct:MultiValidationBehavior.Error="Min: -10"
            />
            <xct:NumericValidationBehavior
                MaximumValue="5"
                xct:MultiValidationBehavior.Error="Max: 5"
            />

        </xct:MultiValidationBehavior>
    </Entry.Behaviors>
</Entry>
```

Properties

PROPERTY	TYPE	DESCRIPTION
Error	object	An attached property on nested validators setting the error message for that validator.
Errors	`List<object>`	Holds the errors from all of the nested invalid validators.

Sample

- MultiValidationBehavior sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- MultiValidationBehavior source code

Xamarin Community Toolkit NumericValidationBehavior

3/5/2021 • 2 minutes to read • Edit Online

The NumericValidationBehavior is a behavior that allows the user to determine if text input is a valid numeric value. For example, an `Entry` control can be styled differently depending on whether a valid or an invalid numeric input is provided. Additional properties handling validation are inherited from ValidationBehavior.

Syntax

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

Properties

PROPERTY	TYPE	DESCRIPTION
MaximumDecimalPlaces	int	The maximum number of decimal places that will be allowed.
MaximumValue	double	The maximum numeric value that will be allowed.
MinimumDecimalPlaces	int	The minimum number of decimal places that will be allowed.
MinimumValue	double	The minimum numeric value that will be allowed.

Sample

- NumericValidationBehavior sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- NumericValidationBehavior source code

Xamarin Community Toolkit RequiredStringValidationBehavior

3/5/2021 • 2 minutes to read • Edit Online

The RequiredStringValidationBehavior is a behavior that allows the user to determine if text input is equal to specific text. For example, an `Entry` control can be styled differently depending on whether a valid or an invalid text input is provided. Additional properties handling validation are inherited from ValidationBehavior.

Syntax

```
<Entry>
    <Entry.Behaviors>
        <xct:RequiredStringValidationBehavior
            InvalidStyle="{StaticResource InvalidEntryStyle}"
            RequiredString="OK"
        />
    </Entry.Behaviors>
</Entry>
```

Properties

PROPERTY	TYPE	DESCRIPTION
RequiredString	string	The string that will be compared to the value provided by the user.

Sample

- RequiredStringValidationBehavior sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- RequiredStringValidationBehavior source code

Xamarin Community Toolkit SetFocusedOnEntryCompletedBehavior

3/25/2021 • 2 minutes to read • Edit Online

The `SetFocusedOnEntryCompletedBehavior` is a behavior that gives focus to a specified visual element when an entry is completed. For example, a page might have several entries in sequence, and it would be convenient to the user if completing an entry automatically switched focus to the next entry.

Syntax

```
<StackLayout>
    <Entry x:Name="Entry1"
           Placeholder="Entry 1"
           xct:SetFocusOnEntryCompletedBehavior.NextElement="{x:Reference Entry2}"
           />
    <Entry x:Name="Entry2"
           Placeholder="Entry 2"
           xct:SetFocusOnEntryCompletedBehavior.NextElement="{x:Reference Entry3}"
           />
    <Entry x:Name="Entry3"
           Placeholder="Entry 3 (no next entry this time)"
           />
</StackLayout>
```

Properties

PROPERTY	TYPE	DESCRIPTION
NextElement	VisualElement	The `VisualElement` that should gain focus once the `Entry` is completed.

Sample

- SetFocusedOnEntryCompletedBehavior sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- SetFocusedOnEntryCompletedBehavior source code

Xamarin Community Toolkit TextValidationBehavior

3/5/2021 • 2 minutes to read • Edit Online

The TextValidationBehavior is a behavior that allows the user to validate a given text depending on specified parameters. By adding this behavior to an `Entry` control it can be styled differently depending on whether a valid or an invalid text value is provided. It offers various built-in checks such as checking for a certain length or whether or not the input value matches a specific regular expression. Additional properties handling validation are inherited from ValidationBehavior.

Syntax

```
<Entry>
    <Entry.Behaviors>
        <xct:TextValidationBehavior
            InvalidStyle="{StaticResource InvalidEntryStyle}"
            MinimumLength="1"
            MaximumLength="10"
        />
    </Entry.Behaviors>
</Entry>
```

Properties

PROPERTY	TYPE	DESCRIPTION
DecorationFlags	TextDecorationFlags	Provides enumerated value to use to set how to handle white spaces.
MaximumLength	int	The maximum length of the value that will be allowed.
MinimumLength	int	The minimum length of the value that will be allowed.
RegexOptions	RegexOptions	Provides enumerated values to use to set regular expression options.
RegexPattern	string	The regular expression pattern which the value will have to match before it will be allowed.

Sample

NOTE

Currently there's no sample available for this feature yet. Want to add one? We are open to community contributions.

API

- TextValidationBehavior source code

Xamarin Community Toolkit UriValidationBehavior

3/5/2021 • 2 minutes to read • Edit Online

The UriValidationBehavior is a behavior that allows users to determine whether or not text input is a valid URI. For example, an `Entry` control can be styled differently depending on whether a valid or an invalid URI is provided. Additional properties handling validation are inherited from ValidationBehavior.

Syntax

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

Properties

PROPERTY	TYPE	DESCRIPTION
UriKind	UriKind	Provides an enumerated value that specifies how to handle different URI types.

Sample

- UriValidationBehavior sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- UriValidationBehavior source code

Xamarin Community Toolkit UserStoppedTypingBehavior

5/4/2021 • 2 minutes to read • Edit Online

The UserStoppedTypingBehavior is a behavior that allows the user to trigger an action when a user has stopped data input an `Entry`. Examples of its usage include triggering a search when a user has stopped entering their search query.

Syntax

```
<Entry>
    <Entry.Behaviors>
        <xct:UserStoppedTypingBehavior
            Command="{Binding SearchCommand}"
            StoppedTypingTimeThreshold="1000"
            MinimumLengthThreshold="3"
            ShouldDismissKeyboardAutomatically="True" />
    </Entry.Behaviors>
</Entry>
```

Properties

PROPERTY	TYPE	DESCRIPTION
Command	ICommand	The command to execute when the user has stopped providing input.
MinimumLengthThreshold	int	The minimum length of the input value required before the command will be executed.
ShouldDismissKeyboardAutomatically	bool	Indicates whether or not the keyboard should be dismissed automatically.
StoppedTypingTimeThreshold	int	The time of inactivity in milliseconds after which the command will be executed.

Sample

- UserStoppedTypingBehavior sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- UserStoppedTypingBehavior source code

Related video

Xamarin Community Toolkit ValidationBehavior

3/23/2021 • 2 minutes to read • Edit Online

The ValidationBehavior allows users to create custom validation behaviors. All of the validation behaviors in the Xamarin Community Toolkit inherit from this behavior, to expose a number of shared properties. Users can inherit from this class to create a custom validation behavior currently not supported through the Xamarin Community Toolkit.

Properties

PROPERTY	TYPE	DESCRIPTION
ForceValidateCommand	ICommand	Allows the user to provide a custom ICommand that handles forcing validation. This is a bindable property.
Flags	ValidationFlags	Provides an enumerated value that specifies how to handle validation. This is a bindable property.
InvalidStyle	Style	The Style to apply to the element when validation fails. This is a bindable property.
IsNotValid	bool	Indicates whether or not the current value is considered invalid. This is a bindable property.
IsValid	bool	Indicates whether or not the current value is considered valid. This is a bindable property.
ValidStyle	Style	The Style to apply to the element when validation is successful.
Value	object	The value to validate. This is a bindable property.
ValuePropertyName	string	Allows the user to override the property that will be used as the value to validate. This is a bindable property.

Visual States

`ValidationBehavior` defines two visual states, `Valid` and `Invalid`, that can be used with the Visual State Manager, instead of the `InvalidStyle` and `ValidStyle` properties.

Usage sample:

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

 <VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="CommonStates">
   <VisualState x:Name="Valid">
    <VisualState.Setters>
     <Setter Property="TextColor" Value="Green"/>
    </VisualState.Setters>
   </VisualState>
   <VisualState x:Name="Invalid">
    <VisualState.Setters>
     <Setter Property="TextColor" Value="IndianRed"/>
    </VisualState.Setters>
   </VisualState>
  </VisualStateGroup>
 </VisualStateManager.VisualStateGroups>
</Entry>
```

Sample

WARNING

This class should not be used without inheriting from it. Therefore, there is no sample available.

API

- ValidationBehavior source code

Xamarin Community Toolkit BoolToObjectConverter

3/5/2021 • 2 minutes to read • Edit Online

The BoolToObjectConverter is a converter that allows users to convert a `bool` value binding to a specific object. By providing both a `TrueObject` and a `FalseObject` in the converter the appropriate object will be used depending on the value of the binding.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <xct:BoolToObjectConverter x:Key="BoolToObjectConverter" TrueObject="16" FalseObject="10" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <Label Text="Hi there from the Docs!" FontSize="{Binding MyBoolean, Converter={StaticResource
BoolToObjectConverter}}" />

    </StackLayout>
</ContentPage>
```

Properties

PROPERTY	TYPE	DESCRIPTION
TrueObject	object	The object that will be used when the binding value is `true`.
FalseObject	object	The object that will be used when the binding value is `false`.

Sample

> **NOTE**
> Currently there's no sample available for this feature yet. Want to add one? We are open to community contributions.

API

- BoolToObjectConverter source code

Xamarin Community Toolkit ByteArrayToImageSourceConverter

3/5/2021 • 2 minutes to read • Edit Online

The ByteArrayToImageSourceConverter is a converter that allows the user to convert an incoming value from byte array and returns an object of type `ImageSource`. This object can then be used as the `Source` of an `Image` control.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">
    <ContentPage.Resources>
         <ResourceDictionary>
             <xct:ByteArrayToImageSourceConverter x:Key="ByteArrayToImageSourceConverter" />
         </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <Image Source="{Binding MyByteArray, Converter={StaticResource ByteArrayToImageSourceConverter}}" />

    </StackLayout>
</ContentPage>
```

Sample

ByteArrayToImageSourceConverter sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- ByteArrayToImageSourceConverter source code

Xamarin Community Toolkit DateTimeOffsetConverter

4/7/2021 • 2 minutes to read • Edit Online

The DateTimeOffsetConverter is a converter that allows users to convert a `DateTimeOffset` to a `DateTime`. Sometimes a datetime value is stored with the offset on a backend to allow for storing the timezone in which a `DateTime` originated from. Controls like the `DatePicker` in Xamarin.Forms will only work with `DateTime`. This converter can be used in those scenarios.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">
    <ContentPage.Resources>
         <ResourceDictionary>
             <xct:DateTimeOffsetConverter x:Key="DateTimeOffsetConverter" />
         </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <Label Text="{Binding MyDateTimeOffset, Converter={StaticResource DateTimeOffsetConverter}}" />

    </StackLayout>
</ContentPage>
```

Sample

DateTimeOffsetConverter sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- DateTimeOffsetConverter source code

Related video

Xamarin Community Toolkit DoubleToIntConverter

3/5/2021 • 2 minutes to read • Edit Online

The DoubleToIntConverter is a converter that allows users to convert an incoming `double` value to an `int`. Optionally the user can provide a multiplier to the conversion through the `Ratio` property.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <xct:DoubleToIntConverter x:Key="DoubleToIntConverter" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <Label Text="{Binding MyDouble, Converter={StaticResource DoubleToIntConverter},
ConverterParameter=2}" />

    </StackLayout>
</ContentPage>
```

Properties

PROPERTY	TYPE	DESCRIPTION
Ratio	int	The multiplier to apply to the conversion.

Sample

NOTE

Currently there's no sample available for this feature yet. Want to add one? We are open to community contributions.

API

- DoubleToIntConverter source code

Xamarin Community Toolkit EnumToBoolConverter

3/23/2021 • 2 minutes to read • Edit Online

The EnumToBoolConverter is a converter that allows users to convert a `Enum` value binding to a `bool` value. The `Enum` value can be compared against the `TrueList` or against the `ConverterParameter`.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             xmlns:myEnum="MyLittleApp.Models"
             x:Class="MyLittleApp.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <!-- Converter with TRUE list -->
            <xct:EnumToBoolConverter x:Key="OpenIssueConverter">
                <xct:EnumToBoolConverter.TrueValues>
                    <myEnum:MyStateEnum>New</myEnum:MyStateEnum>
                    <myEnum:MyStateEnum>InReview</myEnum:MyStateEnum>
                    <myEnum:MyStateEnum>Developing</myEnum:MyStateEnum>
                </xct:EnumToBoolConverter.TrueValues>
            </xct:EnumToBoolConverter>
            <xct:EnumToBoolConverter x:Key="ClosedIssueConverter">
                <xct:EnumToBoolConverter.TrueValues>
                    <myEnum:MyStateEnum>WantFix</myEnum:MyStateEnum>
                    <myEnum:MyStateEnum>Resolved</myEnum:MyStateEnum>
                </xct:EnumToBoolConverter.TrueValues>
            </xct:EnumToBoolConverter>
            <!-- Converter, that uses parameter -->
            <xct:EnumToBoolConverter x:Key="IssueStateConverter" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <!-- Converter with TRUE list -->
        <Label IsVisible="{Binding IssueState, Converter={StaticResource OpenIssueConverter}}" />
        <Label IsVisible="{Binding IssueState, Converter={StaticResource ClosedIssueConverter}}" />
        <!-- Converter, that uses parameter -->
        <Label IsVisible="{Binding IssueState, Converter={StaticResource IssueStateConverter},
ConverterParameter={x:Static myEnum:MyStateEnum.WaitingForCustomer}}" />

    </StackLayout>
</ContentPage>
```

Sample

NOTE

Currently there's no sample available for this feature yet. Want to add one? We are open to community contributions.

API

- IndexToArrayItemConverter source code

Xamarin Community Toolkit EnumToIntConverter

5/21/2021 • 2 minutes to read • Edit Online

The EnumToIntConverter is a converter that allows you to convert a standard `Enum` (extending `int`) to its underlying primitive `int` type. It is useful when binding a collection of values representing an enumeration type with default numbering to a control such as a `Picker`.

For localization purposes or due to other requirements, the enum values often need to be converted to a human-readable string. In this case, when the user selects a value, the resulting `SelectedIndex` can easily be converted to the underlying `enum` value without requiring additional work in the associated viewmodel.

Syntax

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

  <StackLayout Padding="10,10" Spacing="10">
    <Picker ItemsSource="{Binding AllStates}" SelectedIndex="{Binding SelectedState, Converter=
{xct:EnumToIntConverter}}" />
    <Label Text="{Binding Path=SelectedState, Converter={xct:EnumToIntConverter}}" />
  </StackLayout>
</ContentPage>
```

Sample

- EnumToIntConverter sample page source
- EnumToIntConverter sample viewmodel source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- EnumToIntConverter source code

Related video

Xamarin Community Toolkit EqualConverter

3/5/2021 • 2 minutes to read • Edit Online

The EqualConverter is a converter that allows users to convert any value binding to a `bool` depending on whether or not it is equal to a different value. The initial binding contains the object that will be compared and the `ConverterParameter` contains the object to compare it to.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <xct:EqualConverter x:Key="EqualConverter" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <Label IsVisible="{Binding MyFirstObject, Converter={StaticResource EqualConverter},
ConverterParameter=100}" />

    </StackLayout>
</ContentPage>
```

Sample

NOTE

Currently there's no sample available for this feature yet. Want to add one? We are open to community contributions.

API

- EqualConverter source code

Xamarin Community Toolkit IndexToArrayItemConverter

3/5/2021 • 2 minutes to read • Edit Online

The IndexToArrayItemConverter is a converter that allows users to convert a `int` value binding to an item in an array. The `int` value being data bound represents the indexer used to access the array. The array is passed in through the `ConverterParameter`.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <xct:IndexToArrayItemConverter x:Key="IndexToArrayItemConverter" />
            <x:Array x:Key="MyArray" Type="x:String">
                <x:String>Value 1</x:String>
                <x:String>Value 2</x:String>
                <x:String>Value 3</x:String>
                <x:String>Value 4</x:String>
                <x:String>Value 5</x:String>
            </x:Array>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <Label IsVisible="{Binding MyIntegerValue, Converter={StaticResource IndexToArrayItemConverter},
ConverterParameter={StaticResource MyArray}}" />

    </StackLayout>
</ContentPage>
```

Sample

NOTE

Currently there's no sample available for this feature yet. Want to add one? We are open to community contributions.

API

- IndexToArrayItemConverter source code

Xamarin Community Toolkit IntToBoolConverter

3/5/2021 • 2 minutes to read • Edit Online

The IntToBoolConverter is a converter that allows users to convert an incoming `int` value to a `bool`. If the incoming `int` value is 0, it will be converted to `false`. Any other incoming value will be converted to `true`.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <xct:IntToBoolConverter x:Key="IntToBoolConverter" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <Label IsVisible="{Binding MyIntegerValue, Converter={StaticResource IntToBoolConverter}}" />

    </StackLayout>
</ContentPage>
```

Sample

NOTE

Currently there's no sample available for this feature yet. Want to add one? We are open to community contributions.

API

- IntToBoolConverter source code

Xamarin Community Toolkit InvertedBoolConverter

3/5/2021 • 2 minutes to read • Edit Online

The InvertedBoolConverter is a converter that allows users to convert a `bool` value binding to its inverted value.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <xct:InvertedBoolConverter x:Key="InvertedBoolConverter" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <Label IsVisible="{Binding MyBooleanValue, Converter={StaticResource InvertedBoolConverter}}" />

    </StackLayout>
</ContentPage>
```

Sample

NOTE

Currently there's no sample available for this feature yet. Want to add one? We are open to community contributions.

API

- InvertedBoolConverter source code

Xamarin Community Toolkit IsNotNullOrEmptyConverter

3/5/2021 • 2 minutes to read • Edit Online

The IsNotNullOrEmptyConverter is a converter that allows users to convert an incoming binding to a `bool` value. This value represents if the incoming binding value is **not** `null` or empty.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <xct:IsNotNullOrEmptyConverter x:Key="IsNotNullOrEmptyConverter" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <Label IsVisible="{Binding MyStringValue, Converter={StaticResource IsNotNullOrEmptyConverter}}" />

    </StackLayout>
</ContentPage>
```

Sample

NOTE

Currently there's no sample available for this feature yet. Want to add one? We are open to community contributions.

API

- IsNotNullOrEmptyConverter source code

Xamarin Community Toolkit IsNullOrEmptyConverter

3/5/2021 • 2 minutes to read • Edit Online

The IsNullOrEmptyConverter is a converter that allows users to convert an incoming binding to a `bool` value. This value represents if the incoming binding value is `null` or empty.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <xct:IsNullOrEmptyConverter x:Key="IsNullOrEmptyConverter" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <Label IsVisible="{Binding MyStringValue, Converter={StaticResource IsNullOrEmptyConverter}}" />

    </StackLayout>
</ContentPage>
```

Sample

NOTE

Currently there's no sample available for this feature yet. Want to add one? We are open to community contributions.

API

- IsNullOrEmptyConverter source code

Xamarin Community Toolkit ItemSelectedEventArgsConverter

3/5/2021 • 2 minutes to read • Edit Online

The ItemSelectedEventArgsConverter is a converter that allows users to extract the `SelectedItem` value from an SelectedItemChangedEventArgs object. It can subsequently be used in combination with `EventToCommandBehavior`.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <xct:ItemSelectedEventArgsConverter x:Key="ItemSelectedEventArgsConverter" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <ListView ItemsSource="{Binding Items}" HasUnevenRows="True">

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

    </ListView>
</ContentPage>
```

Sample

ItemSelectedEventArgsConverter sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- ItemSelectedEventArgsConverter source code

Xamarin Community Toolkit ItemTappedEventArgsConverter

3/5/2021 • 2 minutes to read • Edit Online

The ItemTappedEventArgsConverter is a converter that allows users to extract the `Item` value from an ItemTappedEventArgs object. It can subsequently be used in combination with `EventToCommandBehavior`.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <xct:ItemTappedEventArgsConverter x:Key="ItemTappedEventArgsConverter" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <ListView ItemsSource="{Binding Items}" HasUnevenRows="True">

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

    </ListView>
</ContentPage>
```

Sample

ItemTappedEventArgsConverter sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- ItemTappedEventArgsConverter source code

Xamarin Community Toolkit ListIsNotNullOrEmptyConverter

3/5/2021 • 2 minutes to read • Edit Online

The ListIsNotNullOrEmptyConverter is a converter that allows users to convert an incoming binding that implements `IEnumerable` to a `bool` value. This value represents if the incoming binding value is **not** `null` or an empty list.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <xct:ListIsNotNullOrEmptyConverter x:Key="ListIsNotNullOrEmptyConverter" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <Label IsVisible="{Binding MyListValue, Converter={StaticResource ListIsNotNullOrEmptyConverter}}"
/>

    </StackLayout>
</ContentPage>
```

Sample

NOTE

Currently there's no sample available for this feature yet. Want to add one? We are open to community contributions.

API

- ListIsNotNullOrEmptyConverter source code

Xamarin Community Toolkit ListIsNullOrEmptyConverter

3/5/2021 • 2 minutes to read • Edit Online

The ListIsNullOrEmptyConverter is a converter that allows users to convert an incoming binding that implements `IEnumerable` to a `bool` value. This value represents if the incoming binding value is either `null` or an empty list.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <xct:ListIsNullOrEmptyConverter x:Key="ListIsNullOrEmptyConverter" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <Label IsVisible="{Binding MyListValue, Converter={StaticResource ListIsNullOrEmptyConverter}}" />

    </StackLayout>
</ContentPage>
```

Sample

ListIsNullOrEmptyConverter sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- ListIsNullOrEmptyConverter source code

Xamarin Community Toolkit ListToStringConverter

3/5/2021 • 2 minutes to read • Edit Online

The ListToStringConverter is a converter that allows users to convert an incoming binding that implements `IEnumerable` to a single `string` value. The `Separator` property is used to join the items in the `IEnumerable`.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="clr-namespace:Xamarin.CommunityToolkit.Converters;assembly=Xamarin.CommunityToolkit"
             x:Class="MyLittleApp.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <xct:ListToStringConverter x:Key="ListToStringConverter" Separator=", " />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <Label IsVisible="{Binding MyListValue, Converter={StaticResource ListToStringConverter}}" />

    </StackLayout>
</ContentPage>
```

Properties

PROPERTY	TYPE	DESCRIPTION
Separator	string	The separator that will be used to join the items in the list.

Sample

> **NOTE**
> Currently there's no sample available for this feature yet. Want to add one? We are open to community contributions.

API

- ListToStringConverter source code

Xamarin Community Toolkit MathExpressionConverter

5/21/2021 • 2 minutes to read • Edit Online

The `MathExpressionConverter` is a converter that allows users to calculate an expression at runtime from supplied arguments:

- **x** or **x0** — The first argument
- **x1** — The second argument
- ...
- **xN-1** — The N argument

> **WARNING**
> Avoid negative operations, constants or variables such as "-cos(30)", "-x" or "-pi", which will return an error. Instead, use multiplication by -1.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">
    <ContentPage.Resources>
         <ResourceDictionary>
             <xct:MathExpressionConverter x:Key="MathExpressionConverter" />
             <xct:MultiMathExpressionConverter x:Key="MultiMathExpressionConverter" />
         </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <Frame
           x:Name="CalculatedFrame"
           HeightRequest="120"
           CornerRadius="{Binding Source={x:Reference CalculatedFrame}, Path=HeightRequest, Converter=
{StaticResource MathExpressionConverter}, ConverterParameter='x/2'}">

        <Label TextColor="Black">
            <Label.Text>
                <MultiBinding Converter="{StaticResource MultiMathExpressionConverter}"
                              ConverterParameter="x0 * x1"
                              StringFormat="The area of Frame = {0}">
                    <Binding Path="{Binding Source={x:Reference CalculatedFrame}, Path=HeightRequest}" />
                    <Binding Path="{Binding Source={x:Reference CalculatedFrame}, Path=WidthRequest}" />
                </MultiBinding>
            </Label.Text>
        </Label>

    </StackLayout>
</ContentPage>
```

Operations

OPERATION	EXAMPLE	EQUIVALENT
+	x + 1	—
-	x - 2	—
*	x * -3	—
/	x / 4	—
^	x ^ 5	Math.Pow
abs	abs(x)	Math.Abs
acos	acos(x)	Math.Acos
asin	asin(x)	Math.Asin
atan	atan(x)	Math.Atan
atan2	atan2(x, 10)	Math.Atan2
ceiling	ceiling(x)	Math.Ceiling

OPERATION	EXAMPLE	EQUIVALENT
cos	cos(x)	Math.Cos
cosh	cosh(x)	Math.Cosh
exp	exp(x)	Math.Exp
floor	floor(x)	Math.Floor
ieeeremainder	ieeeremainder(x, 16)	Math.IEEERemainder
log	log(x, 17)	Math.Log
max	max(x, 18)	Math.Max
min	min(x, 19)	Math.Min
pow	round(x, 2)	Math.Pow
round	round(x, 1)	Math.Round
sign	sign(x)	Math.Sign
sin	sin(x)	Math.Sin
sinh	sinh(x)	Math.Sinh
sqrt	sqrt(x)	Math.Sqrt
tan	tan(x)	Math.Tan
tanh	tanh(x)	Math.Tanh
truncate	truncate(x)	Math.Truncate

Constants

CONSTANT	EQUIVALENT
e	Math.E
pi	Math.PI

Sample

MathExpressionConverter sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- MathExpressionConverter source code
- MultiMathExpressionConverter.cs source code

Xamarin Community Toolkit MultiConverter

3/5/2021 • 2 minutes to read • Edit Online

The MultiConverter is a converter that allows users to chain multiple converters together. The initial binding value is passed through to the first converter and, depending on what these converters return, that value is subsequently passed through to the next converter.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <xct:MultiConverter x:Key="MultiConverter">
                <xct:TextCaseConverter />
                <xct:NotEqualConverter />
            </xct:MultiConverter>
            <x:Array x:Key="MultiConverterParams" Type="{x:Type xct:MultiConverterParameter}">
                <xct:MultiConverterParameter ConverterType="{x:Type xct:TextCaseConverter}" Value="{x:Static
xct:TextCaseType.Upper}" />
                <xct:MultiConverterParameter ConverterType="{x:Type xct:NotEqualConverter}" Value="ANDREI
ROCKS " />
            </x:Array>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <Label IsVisible="{Binding MyStringValue, Converter={StaticResource MultiConverter},
ConverterParameter={StaticResource MultiConverterParams}}" />

    </StackLayout>
</ContentPage>
```

Working with parameters

Due to the nature of how converters work it is not possible to pass parameters to each individual converter in the MultiConverter. To work around this an `IList` of `MultiConverterParameter` objects is accepted as the converter parameter of the MultiConverter. These objects represent the parameters you want to provide for each individual converter. The MultiConverter subsequently matches the type of one of its converters to the type provided in the `ConverterType` property of a `MultiConverterParameter`. It then takes the provided `Value` property and uses that as the `ConverterParameter` of that specific converter.

Sample

MultiConverter sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- MultiConverter source code

Related video

Xamarin Community Toolkit NotEqualConverter

3/5/2021 • 2 minutes to read • Edit Online

The NotEqualConverter is a converter that allows users to convert any value binding to a `boolean` depending on whether or not it is equal to a different value. The initial binding contains the object that will be compared and the `ConverterParameter` contains the object to compare it to.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <xct:NotEqualConverter x:Key="NotEqualConverter" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <Label IsVisible="{Binding MyFirstObject, Converter={StaticResource NotEqualConverter},
ConverterParameter=100}" />

    </StackLayout>
</ContentPage>
```

Sample

NOTE

Currently there's no sample available for this feature yet. Want to add one? We are open to community contributions.

API

- NotEqualConverter source code

Xamarin Community Toolkit TextCaseConverter

5/14/2021 • 2 minutes to read • Edit Online

The TextCaseConverter is a converter that allows users to convert the casing of an incoming `string` type binding. The `Type` property is used to define what kind of casing will be applied to the `string`.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <xct:TextCaseConverter x:Key="TextCaseConverter" Type="Upper" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>

        <Label Text="{Binding MyString, Converter={StaticResource TextCaseConverter}}" />

    </StackLayout>
</ContentPage>
```

Properties

PROPERTY	TYPE	DESCRIPTION
Type	TextCaseType	The type of casing to apply to the `string` value.

TextCaseType

The `TextCaseType` enumeration defines the following members:

- `None` - Applies no specific formatting to the string.
- `Upper` - Applies upper case formatting to the string.
- `Lower` - Applies lower case formatting to the string.

Sample

> **NOTE**
> Currently there's no sample available for this feature yet. Want to add one? We are open to community contributions.

API

- TextCaseConverter source code

- TextCaseType source code

Xamarin Community Toolkit VariableMultiValueConverter

3/5/2021 • 2 minutes to read • Edit Online

The VariableMultiValueConverter is a converter that allows users to convert multiple `boolean` value bindings to a single `boolean`. It does this by enabling them to specify whether `All`, `Any`, `None` or a specific number of values are `true`. This is useful when combined with the MultiBinding included in Xamarin.Forms.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="Xamarin.XamarinCommunityToolkit.MultiBindingConverterPage"
             Title="VariableMultiValueConverter">

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

    <CheckBox>
        <CheckBox.IsChecked>
            <MultiBinding Converter="{StaticResource AllTrueConverter}">
                <Binding Path="Employee.IsOver16" />
                <Binding Path="Employee.HasPassedTest" />
                <Binding Path="Employee.IsSuspended"
                         Converter="{StaticResource InvertedBoolConverter}" />
            </MultiBinding>
        </CheckBox.IsChecked>
    </CheckBox>
</ContentPage>
```

Properties

PROPERTY	TYPE	DESCRIPTION
ConditionType	MultiBindingCondition	Indicates how many values should be true out of the provided boolean values in the MultiBinding. Supports the following values: `All`, `None`, `Any`, `GreaterThan`, `LessThan`.
Count	int	The number of values that should be true when using ConditionType `GreaterThan`, `LessThan` or `Exact`.

Sample

VariableMultiValueConverter sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- VariableMultiValueConverter source code

Related video

Xamarin Community Toolkit StringToListConverter

5/21/2021 • 2 minutes to read • Edit Online

The `StringToListConverter` is a converter that allows the users to convert a `string` value into a string array that contains the substrings in this string that are delimited by the `Separator`, `Separators`, or `ConverterParameter` property.

Syntax

```
<?xml version="1.0" encoding="UTF-8" ?>
<pages:BasePage
    x:Class="Xamarin.CommunityToolkit.Sample.Pages.Converters.StringToListConverterPage"
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:pages="clr-namespace:Xamarin.CommunityToolkit.Sample.Pages"
    xmlns:xct="http://xamarin.com/schemas/2020/toolkit">
    <pages:BasePage.Resources>
        <ResourceDictionary>
            <xct:StringToListConverter x:Key="StringToListConverter" SplitOptions="RemoveEmptyEntries">
                <xct:StringToListConverter.Separators>
                    <x:String>,</x:String>
                    <x:String>.</x:String>
                    <x:String>;</x:String>
                </xct:StringToListConverter.Separators>
            </xct:StringToListConverter>
        </ResourceDictionary>
    </pages:BasePage.Resources>
    <pages:BasePage.Content>
        <Grid Margin="20,0" RowDefinitions="Auto,Auto,*">
            <Label
                Grid.Row="0"
                FontAttributes="Bold"
                Text="Enter some text separated by ',' or '.' or ';'" />
            <Entry
                x:Name="ExampleText"
                Grid.Row="1"
                FontSize="Medium"
                Placeholder="Enter some text separated by ',' or '.' or ';'"
                Text="Item 1,Item 2,Item 3" />
            <ListView
                Grid.Row="2"
                BindingContext="{x:Reference Name=ExampleText}"
                ItemsSource="{Binding Path=Text, Converter={StaticResource StringToListConverter}}">
                <ListView.ItemTemplate>
                    <DataTemplate>
                        <ViewCell>
                            <Label FontSize="Medium" Text="{Binding .}" />
                        </ViewCell>
                    </DataTemplate>
                </ListView.ItemTemplate>
            </ListView>
        </Grid>
    </pages:BasePage.Content>
</pages:BasePage>
```

Properties

PROPERTY	TYPE	DESCRIPTION
Separator	string	The string that delimits the substrings in this string.
Separators	IList<string>	The strings that delimit the substrings in this string.
SplitOptions	StringSplitOptions	A bitwise combination of the enumeration values that specifies whether to trim substrings and include empty substrings.

Sample project

StringToListConverter sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- StringToListConverter source code

Xamarin Community Toolkit Lifecycle Effect

5/14/2021 • 2 minutes to read • Edit Online

The `LifecycleEffect` allows you to determine when a `VisualElement` has its renderer allocated by the platform.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage"
             xmlns:local="clr-namespace:Xamarin.CommunityToolkit.Sample.Pages.Views.TabView">

    <pages:BasePage
    x:Class="Xamarin.CommunityToolkit.Sample.Pages.Effects.LifeCycleEffectPage"
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:pages="clr-namespace:Xamarin.CommunityToolkit.Sample.Pages"
    xmlns:xct="http://xamarin.com/schemas/2020/toolkit">
    <ContentPage.Content>
        <StackLayout x:Name="stack">
            <StackLayout.Effects>
                <xct:LifecycleEffect Loaded="LifeCycleEffect_Loaded" Unloaded="LifeCycleEffect_Unloaded" />
            </StackLayout.Effects>
            <Label
                HorizontalOptions="CenterAndExpand"
                Text="When you press the button, the Image will appear and after 3 seconds will be removed!"
                VerticalOptions="CenterAndExpand">
                <Label.Effects>
                    <xct:LifecycleEffect Loaded="LifeCycleEffect_Loaded" Unloaded="LifeCycleEffect_Unloaded"
/>
                </Label.Effects>
            </Label>
            <Image
                x:Name="img"
                IsVisible="false"

Source="https://raw.githubusercontent.com/xamarin/XamarinCommunityToolkit/main/assets/XamarinCommunityToolki
t_128x128.png">
                <Image.Effects>
                    <xct:LifecycleEffect Loaded="LifeCycleEffect_Loaded" Unloaded="LifeCycleEffect_Unloaded"
/>
                </Image.Effects>
            </Image>
        </StackLayout>
    </ContentPage.Content>

</ContentPage>
```

The `LifeCycleEffect` event handlers are shown below:

```
        void LifeCycleEffect_Loaded(object? sender, EventArgs e)
        {
            if (sender is Button)
                Console.WriteLine("Button loaded");
            if (sender is Image)
                Console.WriteLine("Image loaded");
            if (sender is Label)
                Console.WriteLine("Label loaded");
            if (sender is StackLayout)
                Console.WriteLine("StackLayout loaded");
        }

    void LifeCycleEffect_Unloaded(object? sender, EventArgs e)
    {
     if (sender is Button)
      Console.WriteLine("Button unloaded");
     if (sender is Image)
      Console.WriteLine("Image unloaded");
     if (sender is Label)
      Console.WriteLine("Label unloaded");
     if (sender is StackLayout)
      Console.WriteLine("StackLayout unloaded");
    }
```

Properties

PROPERTY	TYPE	DESCRIPTION
Loaded	event	Triggers when the renderer for the `VisualElement` is allocated.
Unloaded	event	Triggers when the renderer for the `VisualElement` is unallocated.

Sample

LifecycleEffect sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

Related video

Xamarin Community Toolkit SafeAreaEffect

3/5/2021 • 2 minutes to read • Edit Online

The `SafeAreaEffect` is an effect that can be added to any element through an attached property to indicate whether or not that element should take current safe areas into account. This is an area of the screen that is safe for all devices that use iOS 11 and greater. Specifically, it will help to make sure that content isn't clipped by rounded device corners, the home indicator, or the sensor housing on an iPhone X. The effect only targets iOS, meaning that on other platforms it does not do anything.

Syntax

```
<StackLayout VerticalAlignment="Center" SafeAreaEffect.SafeArea="true" HorizontalAlignment="Center"
Width="400" Height="400">
  ...
</StackLayout>
```

Properties

PROPERTY	TYPE	DESCRIPTION
SafeArea	SafeArea	Indicates which safe areas should be taken into account for this element.

Specifying a SafeArea

The `SafeArea` property is of type `SafeArea`. This structure takes up to 4 `boolean` type values indicating which safe areas should be taken into account for the element that this effect is applied to. There are three possibilities when creating a `SafeArea` structure:

- Create a `SafeArea` structure defined by a single uniform value. The single value is applied to the left, top, right, and bottom sides of the element.
- Create a `SafeArea` structure defined by horizontal and vertical values. The horizontal value is symmetrically applied to the left and right sides of the element, with the vertical value being symmetrically applied to the top and bottom sides of the element.
- Create a `SafeArea` structure defined by four distinct values that are applied to the left, top, right, and bottom sides of the element.

Code-behind support

This effect can be also be used from code-behind:

```
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
        SafeAreaEffect.SetSafeArea(stackLayout, new SafeArea(true));
    }
}
```

Sample

SafeAreaEffect sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- SafeAreaEffect source code

Xamarin Community Toolkit ImageResourceExtension

3/5/2021 • 2 minutes to read • Edit Online

The `ImageResourceExtension` is an extension that can be used to display an image from an embedded resource. By providing the resource ID of the embedded resource to this extension, you can bind the embedded resource to the `Source` property of an Image control.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

    <StackLayout>

        <Image Source="{xct:ImageResource Id=MyLittleApp.Resources.MyImage}" />

    </StackLayout>
</ContentPage>
```

Sample

NOTE

Currently there's no sample available for this feature yet. Want to add one? We are open to community contributions.

API

- ImageResourceExtension source code

Xamarin Community Toolkit TranslateExtension

3/25/2021 • 2 minutes to read • Edit Online

The `TranslateExtension` allows users to handle multi-language support at runtime. It uses the built-in `LocalizationResourceManager` helper to retrieve the correct translation resource for the current active `CultureInfo`.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

    <StackLayout>

        <Label Text="{xct:Translate AppResources.ATranslatedMessage}" />

        <Label Text="{xct:Translate AppResources.AnotherTranslatedMessage, StringFormat='#{0}'}" />

    </StackLayout>
</ContentPage>
```

Properties

PROPERTY	TYPE	DESCRIPTION
StringFormat	string	Allows the user to provide additional formatting to the translated text.
Text	string	The resource that will be translated.

Sample

Settings sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- TranslateExtension source code

Related links

- LocalizationResourceManager
- LocalizedString

Xamarin Community Toolkit DelegateWeakEventManager

3/5/2021 • 2 minutes to read • Edit Online

An `event Delegate` implementation that enables the garbage collector to collect an object without needing to unsubscribe event handlers.

Inspired by Xamarin.Forms.WeakEventManager, expanding the functionality of Xamarin.Forms.WeakEventManager to support `Delegate` events.

Syntax

```
public DelegateWeakEventManager()
```

Methods

METHODS	RETURN TYPE	DESCRIPTION
AddEventHandler(Delegate, string eventName)	void	Adds the event handler.
RemoveEventHandler(Delegate, string eventName)	void	Removes the event handler.
HandleEvent(object, object, string	void	Invokes the event EventHandler.
HandleEvent(string	void	Invokes the event Action.
RaiseEvent(object, object, string	void	Invokes the event EventHandler.
RaiseEvent(string	void	Invokes the event Action.

Examples

This section shows how to use this type.

Use Delegate

```
readonly DelegateWeakEventManager _propertyChangedEventManager = new DelegateWeakEventManager();

public event PropertyChangedEventHandler? PropertyChanged
{
    add => _propertyChangedEventManager.AddEventHandler(value);
    remove => _propertyChangedEventManager.RemoveEventHandler(value);
}

void OnPropertyChanged([CallerMemberName]string propertyName = "") =>
_propertyChangedEventManager.RaiseEvent(this, new PropertyChangedEventArgs(propertyName),
nameof(PropertyChanged));
```

Use EventHandler

```
readonly DelegateWeakEventManager _canExecuteChangedEventManager = new DelegateWeakEventManager();

public event EventHandler CanExecuteChanged
{
    add => _canExecuteChangedEventManager.AddEventHandler(value);
    remove => _canExecuteChangedEventManager.RemoveEventHandler(value);
}

void OnCanExecuteChanged() => _canExecuteChangedEventManager.RaiseEvent(this, EventArgs.Empty,
nameof(CanExecuteChanged));
```

Use Action

```
readonly DelegateWeakEventManager _weakActionEventManager = new DelegateWeakEventManager();

public event Action ActionEvent
{
    add => _weakActionEventManager.AddEventHandler(value);
    remove => _weakActionEventManager.RemoveEventHandler(value);
}

void OnActionEvent(string message) => _weakActionEventManager.RaiseEvent(message, nameof(ActionEvent));
```

API

- DelegateWeakEventManager

Related links

- WeakEventManager

Related video

Xamarin Community Toolkit LocalizationResourceManager

3/25/2021 • 2 minutes to read • Edit Online

The `LocalizationResourceManager` class is a helper class that enables users to respond to culture changes at runtime. This class is typically used by the `TranslateExtension` class and `LocalizedString`.

Examples

The following sections show examples of how to use the `LocalizationResourceManager` class.

Initialization

Call the `Init` method in your `App` class constructor, and pass your resource manager to it:

```
LocalizationResourceManager.Current.PropertyChanged += (_, _) => AppResources.Culture =
LocalizationResourceManager.Current.CurrentCulture;
LocalizationResourceManager.Current.Init(AppResources.ResourceManager);
```

You can also subscribe to the `PropertyChanged` event to ensure that your app responds to system culture changes.

Change culture

Use the `CurrentCulture` property to change the culture:

```
LocalizationResourceManager.Current.CurrentCulture = newCulture;
```

Properties

PROPERTY	TYPE	DESCRIPTION
CurrentCulture	CultureInfo	The culture used to provide resource values.

Methods

METHODS	RETURN TYPE	DESCRIPTION
Init(ResourceManager)	void	Initializes a LocalizationResourceManager.
Init(ResourceManager, CultureInfo)	void	Initializes a LocalizationResourceManager.
GetValue(string)	string	Retrieves the localized resource value based on `CurrentCulture`.

Events

EVENTS	DESCRIPTION
PropertyChanged	Provides notification of a culture change.

Sample project

App class Source Settings sample page Source

You can see this class in action in the Xamarin community toolkit sample app.

API

- LocalizationResourceManager

Related links

- TranslateExtension
- LocalizedString

Xamarin Community Toolkit LocalizedString

4/19/2021 • 2 minutes to read • Edit Online

The `LocalizedString` class enables users to respond to system culture changes in C# code at runtime. It uses the built-in LocalizationResourceManager helper class to react to changes in the active CultureInfo.

Examples

The following code raises the `PropertyChanged` event and regenerates the string using function provided in constructor, when the `LocalizationResourceManager.Current.PropertyChanged` event is raised. As a result, the page will be updated with the localized value using the new culture.

ViewModel:

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

Page:

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

Output:

```
Version: 1.0.0
```

NOTE

For this example to work, you also need to update `AppResources.Culture` when `LocalizationResourceManager.Current.CurrentCulture` changes. To do this, add the following code above the `LocalizationResourceManager.Current.Init()` call:

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

Properties

PROPERTY	TYPE	DESCRIPTION
Localized	string	Returns a localized string using the current culture.

Events

EVENTS	DESCRIPTION
PropertyChanged	Provides notification of a culture change.

Sample project

Settings sample page Source

You can see this class in action in the Xamarin community toolkit sample app.

API

- LocalizedString

Related links

- LocalizationResourceManager
- TranslateExtension

Related video

Xamarin Community Toolkit WeakEventManager

3/5/2021 • 2 minutes to read • Edit Online

An event implementation that enables the garbage collector to collect an object without needing to unsubscribe event handlers.

Inspired by Xamarin.Forms.WeakEventManager.

Syntax

```
public WeakEventManager<TEventArgs>()
```

Methods

METHODS	RETURN TYPE	DESCRIPTION
AddEventHandler(EventHandler<TEventArgs>, string eventName)	void	Adds the event handler.
AddEventHandler(Action<TEventArgs>, string eventName)	void	Adds the event handler.
RemoveEventHandler(EventHandler<TEventArgs>, string eventName)	void	Removes the event handler.
RemoveEventHandler(Action<TEventArgs>, string eventName)	void	Removes the event handler.
HandleEvent(object, TEventArgs, string	void	Invokes the event EventHandler.
HandleEvent(TEventArgs, string	void	Invokes the event Action.
RaiseEvent(object, TEventArgs, string	void	Invokes the event EventHandler.
RaiseEvent(TEventArgs, string	void	Invokes the event Action.

Examples

This section shows how to use this type.

Use EventHandler<T>

```
readonly WeakEventManager<string> _errorOcurredEventManager = new WeakEventManager<string>();

public event EventHandler<string> ErrorOcurred
{
    add => _errorOcurredEventManager.AddEventHandler(value);
    remove => _errorOcurredEventManager.RemoveEventHandler(value);
}

void OnErrorOcurred(string message) => _errorOcurredEventManager.RaiseEvent(this, message,
nameof(ErrorOcurred));
```

Use Action<T>

```
readonly WeakEventManager<string> _weakActionEventManager = new WeakEventManager<string>();

public event Action<string> ActionEvent
{
    add => _weakActionEventManager.AddEventHandler(value);
    remove => _weakActionEventManager.RemoveEventHandler(value);
}

void OnActionEvent(string message) => _weakActionEventManager.RaiseEvent(message, nameof(ActionEvent));
```

API

- WeakEventManager<T>

Related links

- DelegateWeakEventManager

Related video

Xamarin Community Toolkit WeakEventManagerExtensions

3/5/2021 • 2 minutes to read • Edit Online

Extension methods for the Xamarin.Forms.WeakEventManager type.

Methods

METHODS	RETURN TYPE	DESCRIPTION
RaiseEvent(this Xamarin.Forms.WeakEventManager, object, object, string	void	Invokes the event EventHandler.

Examples

```
readonly weakEventManager = new Xamarin.Forms.WeakEventManager();

public event EventHandler MyEvent
{
    add => weakEventManager.AddEventHandler(value);
    remove => weakEventManager.RemoveEventHandler(value);
}

void OnEvent() => weakEventManager.RaiseEvent(this, EventArgs.Empty, nameof(MyEvent));
```

API

- WeakEventManagerExtensions

Related links

- DelegateWeakEventManager
- WeakEventManager<T>

Xamarin Community Toolkit AsyncCommand

3/5/2021 • 2 minutes to read • Edit Online

Enables the `Task` type to safely be used asynchronously with an `ICommand`.

Syntax

`AsyncCommand<TExecute, TCanExecute> : IAsyncCommand<TExecute, TCanExecute>`

```
public AsyncCommand(
    Func<TExecute, Task> execute,
    Func<TCanExecute, bool> canExecute = null,
    Action<Exception> onException = null,
    bool continueOnCapturedContext = false,
    bool allowsMultipleExecutions = true)
```

`AsyncCommand<T> : IAsyncCommand<T>`

```
public AsyncCommand(
    Func<T, Task> execute,
    Func<object, bool> canExecute = null,
    Action<Exception> onException = null,
    bool continueOnCapturedContext = false,
    bool allowsMultipleExecutions = true)
```

```
public AsyncCommand(
    Func<T, Task> execute,
    Func<bool> canExecute,
    Action<Exception> onException = null,
    bool continueOnCapturedContext = false,
    bool allowsMultipleExecutions = true)
    : this(execute, _ => canExecute(), onException, continueOnCapturedContext, allowsMultipleExecutions)
```

`AsyncCommand : IAsyncCommand`

```
public AsyncCommand(
    Func<Task> execute,
    Func<object, bool> canExecute = null,
    Action<Exception> onException = null,
    bool continueOnCapturedContext = false,
    bool allowsMultipleExecutions = true)
```

```
public AsyncCommand(
    Func<Task> execute,
    Func<bool> canExecute,
    Action<Exception> onException = null,
    bool continueOnCapturedContext = false,
    bool allowsMultipleExecutions = true)
    : this(execute, _ => canExecute(), onException, continueOnCapturedContext, allowsMultipleExecutions)
```

`IAsyncCommand<TExecute, TCanExecute>`

```
interface IAsyncCommand<TExecute, TCanExecute> : IAsyncCommand<TExecute>
```

`IAsyncCommand<T>`

```
interface IAsyncCommand<T> : ICommand
```

`IAsyncCommand`

```
interface IAsyncCommand : ICommand
```

Methods

METHODS	RETURN TYPE	DESCRIPTION
ExecuteAsync()	Task	Executes the Command as a Task.
ExecuteAsync(T)	Task	Executes the Command as a Task.
ExecuteAsync(TExecute)	Task	Executes the Command as a Task.

Examples

`AsyncCommand`

```
class MyViewModel
{
    bool _isBusy;

    public MyViewModel()
    {
        ButtonCommand = new AsyncCommand(() => ExecuteButtonCommand(), _ => !IsBusy);
    }

    public IAsyncCommand ButtonCommand { get; }

    public bool IsBusy
    {
        get => _isBusy;
        set
        {
            if(_isBusy != value)
            {
                _isBusy = value;
                ButtonCommand.RaiseCanExecuteChanged();
            }
        }
    }

    async Task ExecuteButtonCommand()
    {
        // ...
    }
}
```

`AsyncCommand<T>`

```
class MyViewModel
{
    bool _isBusy;

    public MyViewModel()
    {
        ButtonCommand = new AsyncCommand<int>(buttonClicks => ExecuteButtonCommand(buttonClicks), _ =>
!IsBusy);
    }

    public IAsyncCommand<int> ButtonCommand { get; }

    public bool IsBusy
    {
        get => _isBusy;
        set
        {
            if(_isBusy != value)
            {
                _isBusy = value;
                ButtonCommand.RaiseCanExecuteChanged();
            }
        }
    }


    async Task ExecuteButtonCommand(int buttonClicks)
    {
        // ...
    }
}
```

`AsyncCommand<TExecute, TCanExecute>`

```
class MyViewModel
{
    public MyViewModel()
    {
        ButtonCommand = new AsyncCommand<int, bool>(buttonClicks => ExecuteButtonCommand(buttonClicks),
isBusy => !isBusy);
    }

    public IAsyncCommand<int, bool> ButtonCommand { get; }


    async Task ExecuteButtonCommand(int buttonClicks)
    {
        // ...
    }
}
```

Sample project

AboutViewModel.

You can see this element in action in the Xamarin community toolkit sample app.

API

- AsyncCommand

Related links

- AsyncValueCommand

Related video

Xamarin Community Toolkit AsyncValueCommand

3/5/2021 • 2 minutes to read • Edit Online

Enables the `ValueTask` type to safely be used asynchronously with an `ICommand`.

For more information about the `ValueTask` type, see Understanding the Whys, Whats, and Whens of ValueTask .

Syntax

`AsyncValueCommand<TExecute, TCanExecute> : IAsyncValueCommand<TExecute, TCanExecute>`

```
public AsyncValueCommand(
    Func<TExecute, ValueTask> execute,
    Func<TCanExecute, bool> canExecute = null,
    Action<Exception> onException = null,
    bool continueOnCapturedContext = false,
    bool allowsMultipleExecutions = true)
```

`AsyncValueCommand<T> : IAsyncValueCommand<T>`

```
public AsyncValueCommand(
    Func<T, ValueTask> execute,
    Func<object, bool> canExecute = null,
    Action<Exception> onException = null,
    bool continueOnCapturedContext = false,
    bool allowsMultipleExecutions = true)
```

```
public AsyncValueCommand(
    Func<T, ValueTask> execute,
    Func<bool> canExecute,
    Action<Exception> onException = null,
    bool continueOnCapturedContext = false,
    bool allowsMultipleExecutions = true)
    : this(execute, _ => canExecute(), onException, continueOnCapturedContext, allowsMultipleExecutions)
```

`AsyncValueCommand : IAsyncValueCommand`

```
public AsyncValueCommand(
    Func<Task> execute,
    Func<object, bool> canExecute = null,
    Action<Exception> onException = null,
    bool continueOnCapturedContext = false,
    bool allowsMultipleExecutions = true)
```

```
public AsyncValueCommand(
    Func<Task> execute,
    Func<bool> canExecute,
    Action<Exception> onException = null,
    bool continueOnCapturedContext = false,
    bool allowsMultipleExecutions = true)
    : this(execute, _ => canExecute(), onException, continueOnCapturedContext, allowsMultipleExecutions)
```

`IAsyncValueCommand<TExecute, TCanExecute>`

```
interface IAsyncValueCommand<TExecute, TCanExecute> : IAsyncValueCommand<TExecute>
```

`IAsyncValueCommand<T>`

```
interface IAsyncValueCommand<T> : ICommand
```

`IAsyncValueCommand`

```
interface IAsyncValueCommand : ICommand
```

Methods

METHODS	RETURN TYPE	DESCRIPTION
ExecuteAsync()	ValueTask	Executes the Command as a ValueTask.
ExecuteAsync(T)	ValueTask	Executes the Command as a ValueTask.
ExecuteAsync(TExecute)	ValueTask	Executes the Command as a ValueTask.

Examples

`AsyncValueCommand`

```
class MyViewModel
{
    bool _isBusy;

    public MyViewModel()
    {
        ButtonCommand = new AsyncValueCommand(() => ExecuteButtonCommand(), _ => !IsBusy);
    }

    public IAsyncValueCommand ButtonCommand { get; }

    public bool IsBusy
    {
        get => _isBusy;
        set
        {
            if(_isBusy != value)
            {
                _isBusy = value;
                ButtonCommand.RaiseCanExecuteChanged();
            }
        }
    }

    async ValueTask ExecuteButtonCommand()
    {
        // ...
    }
}
```

`AsyncValueCommand<T>`

```
class MyViewModel
{
    bool _isBusy;

    public MyViewModel()
    {
        ButtonCommand = new AsyncValueCommand<int>(buttonClicks => ExecuteButtonCommand(buttonClicks), _ =>
!IsBusy);
    }

    public IAsyncValueCommand<int> ButtonCommand { get; }

    public bool IsBusy
    {
        get => _isBusy;
        set
        {
            if(_isBusy != value)
            {
                _isBusy = value;
                ButtonCommand.RaiseCanExecuteChanged();
            }
        }
    }


    async ValueTask ExecuteButtonCommand(int buttonClicks)
    {
        // ...
    }
}
```

`AsyncValueCommand<TExecute, TCanExecute>`

```
class MyViewModel
{
    public MyViewModel()
    {
        ButtonCommand = new AsyncValueCommand<int, bool>(buttonClicks => ExecuteButtonCommand(buttonClicks),
isBusy => !isBusy);
    }

    public IAsyncValueCommand<int, bool> ButtonCommand { get; }


    async ValueTask ExecuteButtonCommand(int buttonClicks)
    {
        // ...
    }
}
```

Sample project

You can see this element in action in the Xamarin community toolkit sample app.

API

- AsyncValueCommand

Related links

- AsyncValueCommand

Related video

Xamarin Community Toolkit CommandFactory

3/25/2021 • 2 minutes to read • Edit Online

The `CommandFactory` class provides a unified approach to creating new `Command`, `AsyncCommand,` and `AsyncValueCommand` objects.

Syntax

```
public static CommandFactory.Create()
```

Examples

To consume the `CommandFactory` class, replace `new Command`, `new AsyncCommand` and `new AsyncValueCommand` with the `CommandFactory.Create` method:

```
Command command = CommandFactory.Create(() => Debug.WriteLine("Command executed"));
Command<string> commandWithParameter = CommandFactory.Create<string>(p => Debug.WriteLine("Command executed:
{0}", p));

IAsyncCommand asyncCommand = CommandFactory.Create(ExecuteCommandAsync)
IAsyncCommand<int> asyncCommandWithParameter = CommandFactory.Create<int>(ExecuteCommandAsync)

async Task ExecuteCommandAsync()
{
    // ...
}

async Task ExecuteCommandAsync(string commandParameter)
{
    // ...
}
```

Methods

METHODS	RETURN TYPE	DESCRIPTION
Create(Action)	Command	Initializes Xamarin.Forms.Command.
Create(Action, Func<bool>)	Command	Initializes Xamarin.Forms.Command.
Create(Action<object>)	Command	Initializes Xamarin.Forms.Command.
Create(Action<object>, Func<object, bool>)	Command	Initializes Xamarin.Forms.Command.
Create<T>(Action<T>)	Command<T>	Initializes Xamarin.Forms.Command<T>.
Create<T>(Action<T>, Func<T, bool>)	Command<T>	Initializes Xamarin.Forms.Command<T>.

METHODS	RETURN TYPE	DESCRIPTION
Create(Func<Task> execute, Func<object, bool> canExecute = null, Action<Exception> onException = null, bool continueOnCapturedContext = false, bool allowsMultipleExecutions = true)	IAsyncCommand	Initializes a new instance of IAsyncCommand.
Create(Func<Task> execute, Func<bool> canExecute, Action<Exception> onException = null, bool continueOnCapturedContext = false, bool allowsMultipleExecutions = true)	IAsyncCommand	Initializes a new instance of IAsyncCommand.
Create<TExecute>(Func<TExecute, Task> execute, Func<object, bool> canExecute = null, Action<Exception> onException = null, bool continueOnCapturedContext = false, bool allowsMultipleExecutions = true)	IAsyncCommand<TExecute>	Initializes a new instance of IAsyncCommand<TExecute>.
Create<TExecute>(Func<TExecute, Task> execute, Func<bool> canExecute, Action<Exception> onException = null, bool continueOnCapturedContext = false, bool allowsMultipleExecutions = true)	IAsyncCommand<TExecute>	Initializes a new instance of IAsyncCommand<TExecute>.
Create<TExecute, TCanExecute> (Func<TExecute, Task> execute, Func<TCanExecute, bool> canExecute = null, Action<Exception> onException = null, bool continueOnCapturedContext = false, bool allowsMultipleExecutions = true)	IAsyncCommand<TExecute, TCanExecute>	Initializes a new instance of IAsyncCommand<TExecute, TCanExecute>.
Create(Func<ValueTask>, Func<object, bool> canExecute = null, Action<Exception> onException = null, bool continueOnCapturedContext = false, bool allowsMultipleExecutions = true)	IAsyncValueCommand	Initializes a new instance of IAsyncValueCommand.
Create(Func<ValueTask> execute, Func<bool> canExecute, Action<Exception> onException = null, bool continueOnCapturedContext = false, bool allowsMultipleExecutions = true)	IAsyncValueCommand	Initializes a new instance of IAsyncValueCommand.
Create<TExecute>(Func<TExecute, ValueTask> execute, Func<object, bool> canExecute = null, Action<Exception> onException = null, bool continueOnCapturedContext = false, bool allowsMultipleExecutions = true)	IAsyncValueCommand<TExecute>	Initializes a new instance of IAsyncValueCommand<TExecute>.

METHODS	RETURN TYPE	DESCRIPTION
Create<TExecute>(Func<TExecute, ValueTask> execute, Func<bool> canExecute, Action<Exception> onException = null, bool continueOnCapturedContext = false, bool allowsMultipleExecutions = true)	IAsyncValueCommand<TExecute>	Initializes a new instance of IAsyncValueCommand<TExecute>.
Create<TExecute, TCanExecute>(Func<TExecute, ValueTask> execute, Func<TCanExecute, bool> canExecute = null, Action<Exception> onException = null, bool continueOnCapturedContext = false, bool allowsMultipleExecutions = true)	IAsyncValueCommand<TExecute, TCanExecute>	Initializes a new instance of IAsyncValueCommand<TExecute, TCanExecute>.

Sample project

You can see this element in action in the Xamarin community toolkit sample app. In the sample app, every command is created using this approach. For more information, see BasePage Source as an example.

API

- CommandFactory.Command
- CommandFactory.IAsyncCommand
- CommandFactory.IAsyncValueCommand

Related links

- AsyncCommand
- AsyncValueCommand

Xamarin Community Toolkit C# Markup

3/5/2021 • 14 minutes to read • Edit Online

Download Xamarin.CommunityToolkit.MarkupSample

C# Markup is a set of fluent helper methods and classes to simplify the process of building declarative Xamarin.Forms user interfaces in C#. The fluent API provided by C# Markup is available in the `Xamarin.CommunityToolkit.Markup` namespace.

Just as with XAML, C# Markup enables a clean separation between UI markup and UI logic. This can be achieved by separating UI markup and UI logic into distinct partial class files. For example, for a login page the UI markup would be in a file named *LoginPage.cs*, while the UI logic would be in a file named *LoginPage.logic.cs*.

The latest version of C# Markup requires **Xamarin.Forms 5** and is available in the Xamarin.CommunityToolkit.Markup NuGet package.

C# Markup is available on all platforms supported by Xamarin.Forms.

> **NOTE**
>
> The preview version of C# Markup is available in Xamarin.Forms 4.6 through 4.8 as an experimental feature.
>
> To migrate from the C# Markup preview version to XCT C# Markup:
>
> 1. Update to Xamarin.Forms Forms 5.
> 2. Install the Xamarin.CommunityToolkit.Markup NuGet package.
> 3. Change all references to the `Xamarin.Forms.Markup` namespace to `Xamarin.CommunityToolkit.Markup`, and ensure you include `using Xamarin.Forms;` in your markup files.
> 4. Update `Font` helper calls where needed. `Font` now has `family` as it's first parameter instead of `size`. For example, replace `.Font(15)` with `.Font(size: 15)` or `.FontSize(15)`.

If you are already familiar with the preview version of C# Markup, see Additional Functionality In Xamarin Community Toolkit below.

Basic example

The following example shows setting the page content to a new `Grid` containing a `Label` and an `Entry`, in C#:

```
Grid grid = new Grid();

Label label = new Label { Text = "Code: " };
grid.Children.Add(label, 0, 1);

Entry entry = new Entry
{
    Placeholder = "Enter number", Keyboard = Keyboard.Numeric, BackgroundColor = Color.AliceBlue, TextColor
= Color.Black, FontSize = 15,
    HeightRequest = 44, Margin = fieldMargin
};
grid.Children.Add(entry, 0, 2);
Grid.SetColumnSpan(entry, 2);
entry.SetBinding(Entry.TextProperty, new Binding("RegistrationCode"));

Content = grid;
```

This example creates a `Grid` object, with child `Label` and `Entry` objects. The `Label` displays text, and the `Entry` data binds to the `RegistrationCode` property of the viewmodel. Each child view is set to appear in a specific row in the `Grid`, and the `Entry` spans all the columns in the `Grid`. In addition, the height of the `Entry` is set, along with its keyboard, colors, the font size of its text, and its `Margin`. Finally, the `Page.Content` property is set to the `Grid` object.

C# Markup enables this code to be re-written using its fluent API:

```
Content = new Grid { Children =
{
    new Label { Text = "Code:" }
                .Row (BodyRow.CodeHeader) .Column (BodyCol.Header),

    new Entry { Placeholder = "Enter number", Keyboard = Keyboard.Numeric, BackgroundColor =
Color.AliceBlue, TextColor = Color.Black } .FontSize (15)
                .Row (BodyRow.CodeEntry) .ColumnSpan (All<BodyCol>()) .Margin (fieldMargin) .Height (44)
                .Bind (nameof(vm.RegistrationCode))
}};
```

This example is identical to the previous example, but the C# Markup fluent API simplifies the process of building the UI in C#.

> **NOTE**
> C# Markup includes extension methods that set specific view properties. These extension methods are not meant to replace all property setters. Instead, they are designed to improve code readability, and can be used in combination with property setters. It's recommended to always use an extension method when one exists for a property, but you can choose your preferred balance.

Namespace usings

To use C# Markup, include the following `using` statements in your markup files:

```
using Xamarin.Forms;
using Xamarin.CommunityToolkit.Markup;
```

If you design your markup for:

- LTR only: also include `using Xamarin.CommunityToolkit.Markup.LeftToRight;`
- RTL only: also include `using Xamarin.CommunityToolkit.Markup.RightToLeft;`
- Both LTR and RTL: do not include `LeftToRight` or `RightToLeft` namespaces

To work with `Grid` rows and columns, also include `using static Xamarin.CommunityToolkit.Markup.GridRowsColumns;`.

Data binding

C# Markup includes a `Bind` extension method, along with overloads, that creates a data binding between a view bindable property and a specified property. The `Bind` method knows the default bindable property for the majority of the controls that are included in Xamarin.Forms. Therefore, it's typically not necessary to specify the target property when using this method. You can also register the default bindable property for additional controls:

```
DefaultBindableProperties.Register(
  HoverButton.CommandProperty,
  RadialGauge.ValueProperty
);
```

The `Bind` method can be used to bind to any bindable property:

```
new Label { Text = "No data available" }
          .Bind (Label.IsVisibleProperty, nameof(vm.Empty))
```

In addition, the `BindCommand` extension method can bind to a control's default `Command` and `CommandParameter` properties in a single method call:

```
new TextCell { Text = "Tap me" }
             .BindCommand (nameof(vm.TapCommand))
```

By default, the `CommandParameter` is bound to the binding context. You can also specify the binding path and source for the `Command` and the `CommandParameter` bindings:

```
new TextCell { Text = "Tap Me" }
             .BindCommand (nameof(vm.TapCommand), vm, nameof(Item.Id))
```

In this example, the binding context is an `Item` instance, so you don't need to specify a source for the `Id` `CommandParameter` binding.

If you only need to bind to `Command`, you can pass `null` to the `parameterPath` argument of the `BindCommand` method. Alternatively, use the `Bind` method.

You can also register the default `Command` and `CommandParameter` properties for additional controls:

```
DefaultBindableProperties.RegisterCommand(
    (CustomViewA.CommandProperty, CustomViewA.CommandParameterProperty),
    (CustomViewB.CommandProperty, CustomViewB.CommandParameterProperty)
);
```

Inline converter code can be passed into the `Bind` method with the `convert` and `convertBack` parameters:

```
new Label { Text = "Tree" }
          .Bind (Label.MarginProperty, nameof(TreeNode.TreeDepth),
                 convert: (int depth) => new Thickness(depth * 20, 0, 0, 0))
```

Type-safe converter parameters are also supported:

```
new Label { }
          .Bind (nameof(viewModel.Text),
                 convert: (string text, int repeat) => string.Concat(Enumerable.Repeat(text, repeat)))
```

In addition, converter code and instances can be re-used with the `FuncConverter` class:

```
FuncConverter<int, Thickness> treeMarginConverter = new FuncConverter<int, Thickness>(depth => new
Thickness(depth * 20, 0, 0, 0));
new Label { Text = "Tree" }
           .Bind (Label.MarginProperty, nameof(TreeNode.TreeDepth), converter: treeMarginConverter),
```

The `FuncConverter` class also supports `CultureInfo` objects:

```
cultureAwareConverter = new FuncConverter<DateTimeOffset, string, int>(
    (date, daysToAdd, culture) => date.AddDays(daysToAdd).ToString(culture)
);
```

It's also possible to data bind to `Span` objects that are specified with the `FormattedText` property:

```
new Label { } .FormattedText (
    new Span { Text = "Built with " },
    new Span { TextColor = Color.Blue, TextDecorations = TextDecorations.Underline }
               .BindTapGesture (nameof(vm.ContinueToCSharpForMarkupCommand))
               .Bind (nameof(vm.Title))
)
```

Gesture recognizers

`Command` and `CommandParameter` properties can be data bound to `GestureElement` and `View` types using the `BindClickGesture`, `BindSwipeGesture`, and `BindTapGesture` extension methods:

```
new Label { Text = "Tap Me" }
           .BindTapGesture (nameof(vm.TapCommand))
```

This example creates a gesture recognizer of the specified type, and adds it to the `Label`. The `Bind*Gesture` extension methods offer the same parameters as the `BindCommand` extension methods. However, by default `Bind*Gesture` does not bind `CommandParameter`, while `BindCommand` does.

To initialize a gesture recognizer with parameters, use the `ClickGesture`, `PanGesture`, `PinchGesture`, `SwipeGesture`, and `TapGesture` extension methods:

```
new Label { Text = "Tap Me" }
           .TapGesture (g => g.Bind(nameof(vm.DoubleTapCommand)).NumberOfTapsRequired = 2)
```

Since a gesture recognizer is a `BindableObject`, you can use the `Bind` and `BindCommand` extension methods when you initialize it. You can also initialize custom gesture recognizer types with the `Gesture<TGestureElement, TGestureRecognizer>` extension method.

Layout

C# Markup includes a series of layout extension methods that support positioning views in layouts, and content in views:

TYPE	EXTENSION METHODS
`FlexLayout`	`AlignSelf`, `Basis`, `Grow`, `Menu`, `Order`, `Shrink`
`Grid`	`Row`, `Column`, `RowSpan`, `ColumnSpan`

TYPE	EXTENSION METHODS
`Label`	`TextLeft`, `TextCenterHorizontal`, `TextRight` `TextTop`, `TextCenterVertical`, `TextBottom` `TextCenter`
`IPaddingElement` (e.g. `Layout`)	`Padding`, `Paddings`
`LayoutOptions`	`Left`, `CenterHorizontal`, `FillHorizontal`, `Right` `LeftExpand`, `CenterExpandHorizontal`, `FillExpandHorizontal`, `RightExpand` `Top`, `Bottom`, `CenterVertical`, `FillVertical` `TopExpand`, `BottomExpand`, `CenterExpandVertical`, `FillExpandVertical` `Center`, `Fill`, `CenterExpand`, `FillExpand`
`View`	`Margin`, `Margins`
`VisualElement`	`Height`, `Width`, `MinHeight`, `MinWidth`, `Size`, `MinSize`

Left-to-right and right-to-left support

For C# Markup that is designed to support either left-to-right (LTR) or right-to-left (RTL) flow direction, the extension methods listed above offer the most intuitive set of names: `Left`, `Right`, `Top` and `Bottom`.

To make the correct set of left and right extension methods available, and in the process make explicit which flow direction the markup is designed for, include one of the following two `using` directives: `using Xamarin.CommunityToolkit.Markup.LeftToRight;`, or `using Xamarin.CommunityToolkit.Markup.RightToLeft;`.

For C# Markup that is designed to support both left-to-right and right-to-left flow direction, it's recommended to use the extension methods in the following table rather than either of the above namespaces:

TYPE	EXTENSION METHODS
`Label`	`TextStart`, `TextEnd`
`LayoutOptions`	`Start`, `End` `StartExpand`, `EndExpand`

Layout line convention

The recommended convention is to put all the layout extension methods for a view on a single line in the following order:

1. The row and column that contain the view.
2. Alignment within the row and column.
3. Margins around the view.
4. View size.
5. Padding within the view.
6. Content alignment within the padding.

The following code shows an example of this convention:

```
new Label { }
          .Row (BodyRow.Prompt) .ColumnSpan (All<BodyCol>()) .FillExpandHorizontal () .CenterVertical ()
.Margin (fieldNameMargin) .TextCenterHorizontal () // Layout line
```

Consistently following the convention enables you to quickly read C# Markup and build a mental map of where the view content is located in the UI.

Grid rows and columns

Enumerations can be used to define `Grid` rows and columns, instead of using numbers. This offers the advantage that renumbering is not required when adding or removing rows or columns.

> **IMPORTANT**
> Defining `Grid` rows and columns using enumerations requires the following `using` directive:
> `using static Xamarin.CommunityToolkit.Markup.GridRowsColumns;`

The following code shows an example of how to define and consume `Grid` rows and columns using enumerations:

```
using Xamarin.Forms;
using Xamarin.CommunityToolkit.Markup;
using Xamarin.CommunityToolkit.Markup.LeftToRight;
using static Xamarin.CommunityToolkit.Markup.GridRowsColumns;
// ...

enum BodyRow { Prompt, CodeHeader, CodeEntry, Button }
enum BodyCol { FieldLabel, FieldValidation }

View Build() => new Grid
{
    RowDefinitions = Rows.Define(
        (BodyRow.Prompt    , 170 ),
        (BodyRow.CodeHeader, 75  ),
        (BodyRow.CodeEntry , Auto),
        (BodyRow.Button    , Auto)
    ),

    ColumnDefinitions = Columns.Define(
        (BodyCol.FieldLabel     , Stars(0.5)),
        (BodyCol.FieldValidation, Star)
    ),

    Children =
    {
        new Label { LineBreakMode = LineBreakMode.WordWrap } .FontSize (15) .Bold ()
                  .Row (BodyRow.Prompt) .ColumnSpan (All<BodyCol>()) .FillExpandHorizontal ()
.CenterVertical () .Margin (fieldNameMargin) .TextCenterHorizontal ()
                  .Bind (nameof(vm.RegistrationPrompt)),

        new Label { Text = "Registration code" } .Bold ()
                  .Row (BodyRow.CodeHeader) .Column(BodyCol.FieldLabel) .Bottom () .Margin
(fieldNameMargin),

        new Label { } .Italic ()
                  .Row (BodyRow.CodeHeader) .Column (BodyCol.FieldValidation) .Right () .Bottom () .Margin
(fieldNameMargin)
                  .Bind (nameof(vm.RegistrationCodeValidationMessage)),

        new Entry { Placeholder = "E.g. 123456", Keyboard = Keyboard.Numeric, BackgroundColor =
Color.AliceBlue, TextColor = Color.Black } .FontSize (15)
                  .Row (BodyRow.CodeEntry) .ColumnSpan (All<BodyCol>()) .Margin (fieldMargin) .Height (44)
                  .Bind (nameof(vm.RegistrationCode), BindingMode.TwoWay),

        new Button { Text = "Verify" } .Style (FilledButton)
                   .Row (BodyRow.Button) .ColumnSpan (All<BodyCol>()) .FillExpandHorizontal () .Margin
(PageMarginSize)
                   .Bind (Button.IsVisibleProperty, nameof(vm.CanVerifyRegistrationCode))
                   .Bind (nameof(vm.VerifyRegistrationCodeCommand)),
    }
};
```

In addition, you can concisely define rows and columns without enumerations:

```
new Grid
{
    RowDefinitions = Rows.Define (Auto, Star, 20),
    ColumnDefinitions = Columns.Define (Auto, Star, 20, 40)
    // ...
}
```

Fonts

Controls that implement `IFontElement` can call the `FontSize`, `Bold`, `Italic`, and `Font` extension methods to set the appearance of the text displayed by the control, e.g.:

- `Button`
- `DatePicker`
- `Editor`
- `Entry`
- `Label`
- `Picker`
- `SearchBar`
- `Span`
- `TimePicker`

Effects

Effects can be attached to controls with the `Effects` extension method:

```
new Button { Text = "Tap Me" }
            .Effects (new ButtonMixedCaps())
```

Logic integration

The `Invoke` extension method can be used to execute code inline in your C# Markup:

```
new ListView { } .Invoke (l => l.ItemTapped += OnListViewItemTapped)
```

In addition, you can use the `Assign` extension method to access a control from outside the UI markup (in the UI logic file):

```
new ListView { } .Assign (out MyListView)
```

Styles

The following example shows how to create implicit and explicit styles using C# Markup:

```
using Xamarin.Forms;
using Xamarin.CommunityToolkit.Markup;

namespace CSharpForMarkupDemos
{
    public static class Styles
    {
        static Style<Button> buttons, filledButton;
        static Style<Label> labels;
        static Style<Span> link;

        #region Implicit styles

        public static ResourceDictionary Implicit => new ResourceDictionary { Buttons, Labels };

        public static Style<Button> Buttons => buttons ?? (buttons = new Style<Button>(
            (Button.HeightRequestProperty, 44),
            (Button.FontSizeProperty, 13),
            (Button.HorizontalOptionsProperty, LayoutOptions.Center),
            (Button.VerticalOptionsProperty, LayoutOptions.Center)
        ));

        public static Style<Label> Labels => labels ?? (labels = new Style<Label>(
            (Label.FontSizeProperty, 13),
            (Label.TextColorProperty, Color.Black)
        ));

        #endregion Implicit styles

        #region Explicit styles

        public static Style<Button> FilledButton => filledButton ?? (filledButton = new Style<Button>(
            (Button.TextColorProperty, Color.White),
            (Button.BackgroundColorProperty, Color.FromHex("#1976D2")),
            (Button.CornerRadiusProperty, 5)
        )).BasedOn(Buttons);

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

        #endregion Explicit styles
    }
}
```

The implicit styles can be consumed by loading them into the application resource dictionary:

```
public App()
{
    Resources = Styles.Implicit;
    // ...
}
```

Explicit styles can be consumed with the `Style` extension method.

```
using static CSharpForMarkupExample.Styles;
// ...

new Button { Text = "Tap Me" } .Style (FilledButton),
```

> **NOTE**
> In addition to the `Style` extension method, there are also `ApplyToDerivedTypes`, `BasedOn`, `Add`, and `CanCascade` extension methods.

Alternatively, you can create your own styling extension methods:

```
public static TButton Filled<TButton>(this TButton button) where TButton : Button
{
    button.Buttons(); // Equivalent to Style .BasedOn (Buttons)
    button.TextColor = Color.White;
    button.BackgroundColor = Color.Red;
    return button;
}
```

The `Filled` extension method can then be consumed as follows:

```
new Button { Text = "Tap Me" } .Filled ()
```

Platform-specifics

The `Invoke` extension method can be used to apply platform-specifics. However, to avoid ambiguity errors, don't include `using` directives for the `Xamarin.Forms.PlatformConfiguration.*Specific` namespaces directly. Instead, create a namespace alias and consume the platform-specific via the alias:

```
using Xamarin.Forms;
using Xamarin.CommunityToolkit.Markup;
using PciOS = Xamarin.Forms.PlatformConfiguration.iOSSpecific;
// ...

new ListView { } .Invoke (l => PciOS.ListView.SetGroupHeaderStyle(l, PciOS.GroupHeaderStyle.Grouped))
```

In addition, if you consume certain platform-specifics frequently you can create fluent extension methods for them in your own extensions class:

```
public static T iOSGroupHeaderStyle<T>(this T listView, PciOS.GroupHeaderStyle style) where T :
Forms.ListView
{
  PciOS.ListView.SetGroupHeaderStyle(listView, style);
  return listView;
}
```

The extension method can then be consumed as follows:

```
new ListView { } .iOSGroupHeaderStyle(PciOS.GroupHeaderStyle.Grouped)
```

For more information about platform-specifics, see Android platform features, iOS platform features, and Windows platform features.

Recommended convention

A recommended order and grouping of properties and helper methods is:

- **Purpose**: any property or helper methods whose value identifies the control's purpose (e.g. `Text`, `Placeholder`, . `Assign`).
- **Other**: all properties or helper methods that are not layout or binding, on the same line or multiple lines.
- **Layout**: layout is ordered inwards: rows and columns, layout options, margin, size, padding, and content alignment.
- **Bind**: data binding is performed at the end of the method chain, with one bound property per line. If the *default* bindable property is bound, it should be at the end of the method chain.

The following code shows an example of following this convention:

```
new Button { Text = "Verify" /* purpose */ } .Style (FilledButton) // other
            .Row (BodyRow.Button) .ColumnSpan (All<BodyCol>()) .FillExpandHorizontal () .Margin (10) //
layout
            .Bind (Button.IsVisibleProperty, nameof(vm.CanVerifyRegistrationCode)) // bind
            .Bind (nameof(vm.VerifyRegistrationCodeCommand)), // bind default

new Label { }
           .Assign (out animatedMessageLabel) // purpose
           .Invoke (label => label.SizeChanged += MessageLabel_SizeChanged) // other
           .Row (BodyRow.Message) .ColumnSpan (All<BodyCol>()) // layout
           .Bind (nameof(vm.Message)), // bind default
```

Consistently applying this convention enables you to quickly scan your C# Markup and build a mental image of the UI layout.

Additional functionality in Xamarin Community Toolkit

In the Xamarin Community Toolkit, C# Markup adds support for:

- `MultiBinding`
- `MultiConverter`
- `BindableLayout`
- `RelativeLayout`
- `DynamicResource`

Multi-Binding helpers

New overloads of the `Bind` helper offer support for multi-binding.

There are overloads that support 2, 3 or 4 bindings with a type-safe inline converter:

```
new Label { }
    .Bind (Label.TextProperty,
        new Binding (nameof(vm.Name)),
        new Binding (nameof(vm.Score)),
        ((string name, bool score) v) => $"{v.name} Score: { v.score }"
    )
```

The value for all bindings are passed in as a `ValueTuple` with type-safe members.

You can also pass in a type-safe converter parameter:

```
new Label { }
    .Bind (Label.TextProperty,
        new Binding (nameof(vm.Name)),
        new Binding (nameof(vm.Score)),
        ((string name, int Score) v, bool winner) => $"{v.name} Score: { v.Score } Winner: { winner }",
        converterParameter: true
    )
```

Here `bool winner` gets the value from the `converterParameter`.

You can specify two-way conversion inline:

```
new Entry { }
    .Bind(Entry.TextProperty,
        new Binding (nameof(vm.Emoticon)),
        new Binding (nameof(vm.Repeat)),
        ((char emoticon, int repeat) v) => new string(v.emoticon, v.repeat),
        (string emoticons) => (emoticons[0], emoticons.Length)
    );
```

In the `convertBack` function you return the same `ValueTuple` that you receive in the `convert` function.

You can specify more than 4 bindings by passing in a multi-value converter:

```
new Label { }
    .Bind(Label.TextProperty,
        new List<BindingBase> {
            new Binding(nameof(vm.Name)),
            new Binding(nameof(vm.Score))
        },
        new FuncMultiConverter<string, bool>(
            (object[] values, bool winner) => $"{values[0]} Score: { values[1] } Winner: { winner }"
        )
    )
```

This is not type-safe: you will need to cast the values to their type in the `convert` function.

The `FuncMultiConverter` classes implement `IMultiValueConverter`. The class used for any number of bindings is `FuncMultiConverter<TDest, TParam>`, which only specifies the destination type and the parameter type of the convertor. The binding values are passed as an `object[]`.

There are also type-safe generic overloads for `FuncMultiConverter` that take 2, 3 or 4 values (and optionally a convertor parameter). These classes pass the binding values in a type-safe `ValueTuple`.

Bindable layout helpers

The `EmptyView`, `EmptyViewTemplate`, `ItemsSource`, `ItemTemplate` and `ItemTemplateSelector` helpers offer support for bindable layouts on all `Layout<View>` types:

```
new StackLayout { }
    .ItemTemplate (() =>
        new Label { }
            .Bind (nameof(Item.Name))
        )
    .ItemsSource (vm.Items)
```

RelativeLayout helpers

The `Children` helpers lets you add constrained child views to a `RelativeLayout`.

To create constrained views from normal views, four helpers have been added: `Unconstrained`, `Constraints` and two `Constrain` overloads. Each overload returns a corresponding `*ConstrainedView` class, which offers a fluent API for setting constraints on `RelativeLayout` child views.

Constraints on a child view can be set with:

- A single Bounds expression.
- Separate expressions for X, Y, Width and Height.
- Separate `Constraint` instances for X, Y, Width and Height. Each of these Constraint instances has overloads for:
  - Constant.
  - Relative to parent.
  - Relative to view.

The following code shows examples of using these helpers:

```
new RelativeLayout { } .Children (
    new Label { } // Bounds constrained
        .Assign (out Label child0)
        .Constrain(() => new Rectangle(30, 20, layout.Height / 2, layout.Height / 4)),

    new Label { } // Expressions constrained
        .Constrain() .X      (() => 30)
                     .Y      (() => 20)
                     .Width  (() => layout.Height / 2)
                     .Height (() => layout.Height / 4),

    new Label { } // Constraints constrained - parent relative
        .Constraints() .X      (30)
                       .Y      (20)
                       .Width  (parent => parent.Height / 5)
                       .Height (parent => parent.Height / 10),

    new Label { } // Constraints constrained - view relative
        .Constraints() .X      (child0, (layout, view) => view.Bounds.Right + 10)
                       .Y      (child0, (layout, view) => view.Y)
                       .Width  (child0, (layout, view) => view.Width)
                       .Height (child0, (layout, view) => view.Height),
) .Assign (out layout)
```

Dynamic resource helpers

The `DynamicResource`, `DynamicResources` and `RemoveDynamicResources` helpers add support for setting dynamic resources on an `Element`:

```
new Label { }
    .DynamicResource (Label.TextProperty, "TextKey")

new Label { }
    .DynamicResources((Label.TextProperty     , "TextKey"),
                      (Label.TextColorProperty, "ColorKey"));
```

Related links

- Xamarin Community Toolkit
- Xamarin Forms Android platform features

- Xamarin Forms iOS platform features
- Xamarin Forms Windows platform features

Xamarin Community Toolkit AvatarView

3/5/2021 • 2 minutes to read • Edit Online

The AvatarView control allows the user to display an avatar or the user's initials if no avatar is available. By binding the `Source` property the user can assign an image to the AvatarView. Simultaneously binding the `Text` property will allow the user to also set the initials to be shown if no valid image is provided.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

     <StackLayout>

        <xct:AvatarView Text="{Binding Initials}" Source="{Binding AvatarSource}" />

    </StackLayout>

</ContentPage>
```

Properties

PROPERTY	TYPE	DESCRIPTION
BorderColor	`Color`	Gets or sets the border color for the AvatarView.
Color	`Color`	Gets or sets the color that will fill the background of the AvatarView.
CornerRadius	double	Gets or sets the corner radius of the AvatarView.
FontAttributes	`FontAttributes`	Gets a value that indicates whether the font for the text is bold, italic, or neither.
FontFamily	string	Gets the font family to which the font for the text belongs.
FontSize	double	Gets the size of the font for the text.
Size	double	Gets or sets the desired height and width of the AvatarView.
Source	`ImageSource`	Gets or sets the source of the image. This is a bindable property.

PROPERTY	TYPE	DESCRIPTION
Text	string	Gets or sets the text for the label. This is a bindable property.
TextColor	`Color`	Gets or sets the Color for the text of the label. This is a bindable property.

Sample

AvatarView sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- AvatarView source code

Xamarin Community Toolkit BadgeView

3/5/2021 • 2 minutes to read • Edit Online

The `BadgeView` allows the user to show a badge with a string value on top of any control. By wrapping a control in a `BadgeView` control, you can show a badge value on top of it. This is very much like the badges you see on the app icons on iOS and Android.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

     <StackLayout>

        <xct:BadgeView
            BackgroundColor="Red"
            FontAttributes="Bold"
            FontSize="Medium"
            TextColor="White"
            Text="1">
            <Label
                Text="This label has a badge in the top-right"/>
        </xct:BadgeView>

    </StackLayout>

</ContentPage>
```

Properties

PROPERTY	TYPE	DESCRIPTION
AutoHide	bool	Gets or sets if the badge should be hidden automatically when the value is `"0"`. This is a bindable property.
BackgroundColor	`Color`	Gets or sets the background color for the badge. This is a bindable property.
BadgeAnimation	IBadgeAnimation	Gets or sets the animation that should be used when the badge is shown or hidden. The animation only occurs if `IsAnimated` is set to `true`. This is a bindable property.
BadgePosition	BadgePosition	Gets or sets the position where the badge will be shown relative to the `Content`. This is a bindable property.
BorderColor	`Color`	Gets or sets the border color for the badge. This is a bindable property.

PROPERTY	TYPE	DESCRIPTION
Content	`View`	Gets or sets the `View` on top of which the `BadgeView` will be shown. This is a bindable property.
FontAttributes	`FontAttributes`	Gets or sets the font attributes to be used for the text of the `BadgeView`. This is a bindable property.
FontFamily	string	Gets or sets the font to be used for the text of the `BadgeView`. This is a bindable property.
FontSize	double	Gets or sets the font size for the text of the `BadgeView`. `NamedSize` values can be used. This is a bindable property.
HasShadow	bool	Gets or sets if the badge should have a shadow when shown. This is a bindable property.
IsAnimated	bool	Gets or sets if the badge should be animated when it is shown or hidden. This is a bindable property.
Text	string	Gets or sets the text for the badge. This is a bindable property.
TextColor	`Color`	Gets or sets the text color for the badge. This is a bindable property.

Sample

BadgeView sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- BadgeView source code

Xamarin Community Toolkit CameraView

3/5/2021 • 2 minutes to read • Edit Online

The CameraView control enables the user to display a preview of the camera output. In addition, it can take photos or record videos. The CameraView also offers the options you would expect to support taking photos and recording videos such as turning the flash on or off, saving the captured media to a file, and offering different hooks for events.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

     <StackLayout>

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

    </StackLayout>

</ContentPage>
```

Properties

PROPERTY	TYPE	DESCRIPTION
CameraOptions	CameraOptions	Gets or sets which camera device should be used for the CameraView.
CaptureMode	CameraCaptureMode	Gets or sets the capture mode for the CameraView. Either default, video or photo.
FlashMode	CameraFlashMode	Gets or sets the flash mode of the CameraView.
IsAvailable	bool	Gets or sets if the camera device is currently available for use.
IsBusy	bool	Gets or sets if the camera is currently busy capturing media.
MaxZoom	double	Gets or sets the maximum zoom level of the CameraView.

PROPERTY	TYPE	DESCRIPTION
ShutterCommand	`ICommand`	Gets or sets a `Command` that is invoked when the shutter is triggered.
VideoStabilization	bool	Gets or sets the video stabilization on the camera of the CameraView.
Zoom	double	Gets or sets the current zoom level of the CameraView.

Events

EVENT	TYPE	DESCRIPTION
MediaCaptured	EventHandler<MediaCapturedEventArgs>	Event that is triggered whenever media is captured successfully.
MediaCaptureFailed	EventHandler<string>	Event that is triggered whenever media capture failed.
OnAvailable	EventHandler<bool>	Event that is triggered whenever the selected camera device availability changes.

Sample

CameraView sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- CameraView source code

Xamarin Community Toolkit DockLayout

3/5/2021 • 2 minutes to read • Edit Online

The DockLayout makes it easy to dock content in all four directions (top, bottom, left and right).

This makes it a great choice in many situations, where you want to divide the screen into specific areas. By default, the last element inside the DockLayout will automatically fill the rest of the space (center), unless this feature is specifically disabled.

The dock position on the child elements are set through an attached property.

Inspired by WPF DockPanel.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

     <StackLayout>

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

    </StackLayout>

</ContentPage>
```

Properties

PROPERTY	TYPE	DESCRIPTION
Dock	Dock	This should be set on the child elements of the `DockLayout`. This determines in what direction the child element is docked. This is a bindable property.
LastChildFill	bool	Gets or sets whether or not the last child defined in the `DockLayout` should fill the remaining space in the center. This is a bindable property.

Sample

DockLayout sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- DockLayout source code

Xamarin Community Toolkit Expander

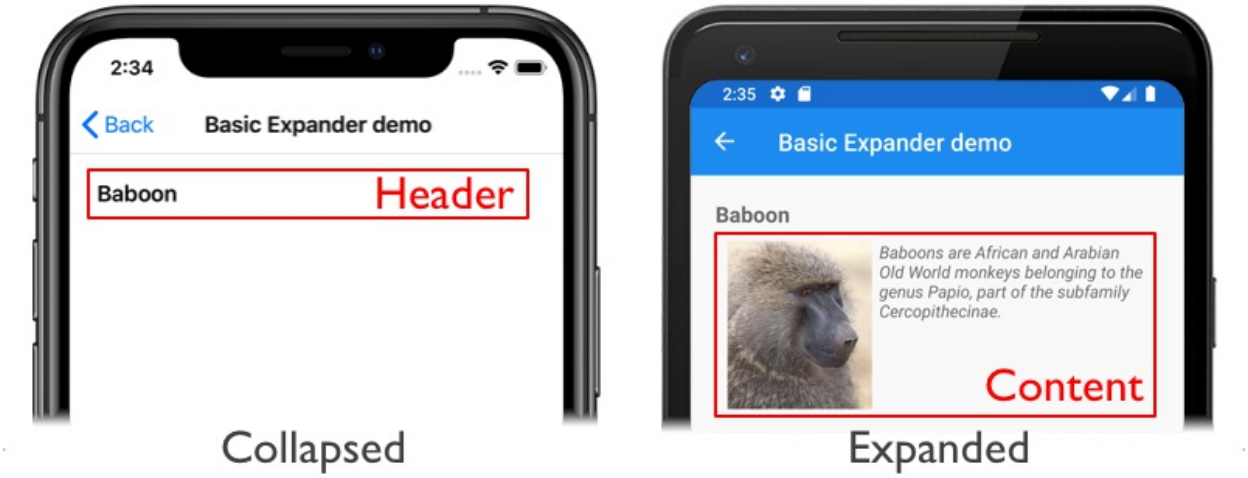
3/5/2021 • 7 minutes to read • Edit Online

Download the sample

The Xamarin.CommunityToolkit `Expander` control provides an expandable container to host any content. The control has a header and content, and the content is shown or hidden by tapping the `Expander` header. When only the `Expander` header is shown, the `Expander` is *collapsed*. When the `Expander` content is visible the `Expander` is *expanded*.

> **NOTE**
> The `Expander` control is known to show unwanted behavior when used in a `ListView` or `CollectionView`. At this time we recommend not using a `Expander` in one of these controls.

The following screenshots show an `Expander` in its collapsed and expanded states, with red boxes indicating the header and the content:



> **NOTE**
> The `Expander` control is fully implemented in cross-platform code. Therefore, it's available on all platforms supported by Xamarin.Forms.

The `Expander` control defines the following properties:

- `CollapseAnimationEasing`, of type `Easing`, which represents the easing function to be applied to the `Expander` content when it's collapsing.
- `CollapseAnimationLength`, of type `uint`, which defines the duration of the animation when the `Expander` is collapsing. The default value of this property is 250ms.
- `Command`, of type `ICommand`, which is executed when the `Expander` header is tapped.
- `CommandParameter`, of type `object`, which is the parameter that's passed to the `Command`.
- `Content`, of type `View`, which defines the content to be displayed when the `Expander` expands.
- `ContentTemplate`, of type `DataTemplate`, which is the template used to dynamically inflate the content of the `Expander`.

- `ExpandAnimationEasing`, of type `Easing`, which represents the easing function to be applied to the `Expander` content during expansion.
- `ExpandAnimationLength`, of type `uint`, which defines the duration of the animation when the `Expander` expands. The default value of this property is 250ms.
- `ForceUpdateSizeCommand`, of type `ICommand`, which defines the command that's executed when the size of the `Expander` is force updated. This property uses the `OneWayToSource` binding mode.
- `Header`, of type `View`, which defines the header content.
- `IsExpanded`, of type `bool`, which determines if the `Expander` is expanded . This property uses the `TwoWay` binding mode, and has a default value of `false`.
- `State`, of type `ExpandState`, which represents the state of the `Expander`. This property uses the `OneWayToSource` binding mode.

These properties are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

> **NOTE**
> The `Content` property is the content property of the `Expander` class, and therefore does not need to be explicitly set from XAML.

The `ExpandState` enumeration defines the following members:

- `Expanding` indicates that the `Expander` is expanding.
- `Expanded` indicates that the `Expander` is expanded.
- `Collapsing` indicates that the `Expander` is collapsing.
- `Collapsed` indicates that the `Expander` is collapsed.

The `Expander` control also defines a `Tapped` event that's fired when the `Expander` header is tapped. In addition, `Expander` includes a `ForceUpdateSize` method that can be called to programmatically resize the `Expander` at runtime.

> **IMPORTANT**
> The Expander has been part of Xamarin.Forms and has been moved to the Xamarin.CommunityToolkit. As part of that move, the `ExpanderState` enum has been renamed to `ExpandState`

Create an Expander

The following example shows how to instantiate an `Expander` in XAML:

```
<Expander>
    <Expander.Header>
        <Label Text="Baboon"
               FontAttributes="Bold"
               FontSize="Medium" />
    </Expander.Header>
    <Grid Padding="10">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>
        <Image
Source="http://upload.wikimedia.org/wikipedia/commons/thumb/f/fc/Papio_anubis_%28Serengeti%2C_2009%29.jpg/200px-Papio_anubis_%28Serengeti%2C_2009%29.jpg"
               Aspect="AspectFill"
               HeightRequest="120"
               WidthRequest="120" />
        <Label Grid.Column="1"
               Text="Baboons are African and Arabian Old World monkeys belonging to the genus Papio, part of the subfamily Cercopithecinae."
               FontAttributes="Italic" />
    </Grid>
</Expander>
```

In this example, the `Expander` is collapsed by default and displays a `Label` as its header. Tapping on the header results in the `Expander` expanding to reveal its content, which is a `Grid` containing child controls. When the `Expander` is expanded, tapping its header collapses the `Expander`.

> **IMPORTANT**
>
> When setting the `Expander.Content` property, either implicitly or explicitly, the `Expander` content is created when the page containing it is navigated to, even if the `Expander` is collapsed. However, the `Expander.ContentTemplate` property can be set to content that only gets inflated when the `Expander` expands for the first time. For more information, see Create Expander content on demand.

Alternatively, an `Expander` can be created in code:

```
Expander expander = new Expander
{
    Header = new Label
    {
        Text = "Baboon",
        FontAttributes = FontAttributes.Bold,
        FontSize = Device.GetNamedSize(NamedSize.Medium, typeof(Label))
    }
};

Grid grid = new Grid
{
    Padding = new Thickness(10),
    ColumnDefinitions =
    {
        new ColumnDefinition { Width = GridLength.Auto },
        new ColumnDefinition { Width = GridLength.Auto }
    }
};

grid.Children.Add(new Image
{
    Source =
"http://upload.wikimedia.org/wikipedia/commons/thumb/f/fc/Papio_anubis_%28Serengeti%2C_2009%29.jpg/200px-
Papio_anubis_%28Serengeti%2C_2009%29.jpg",
    Aspect = Aspect.AspectFill,
    HeightRequest = 120,
    WidthRequest = 120
});

grid.Children.Add(new Label
{
    Text = "Baboons are African and Arabian Old World monkeys belonging to the genus Papio, part of the
subfamily Cercopithecinae.",
    FontAttributes = FontAttributes.Italic
}, 1, 0);

expander.Content = grid;
```

Create Expander content on demand

`Expander` content can be created on demand, in response to the `Expander` expanding. This can be accomplished by setting the `Expander.ContentTemplate` property to a `DataTemplate` that contains the content:

```
<Expander>
    <Expander.Header>
        <Label Text="{Binding Name}"
               FontAttributes="Bold"
               FontSize="Medium" />
    </Expander.Header>
    <Expander.ContentTemplate>
        <DataTemplate>
            <Grid Padding="10">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto" />
                    <ColumnDefinition Width="Auto" />
                </Grid.ColumnDefinitions>
                <Image Source="{Binding ImageUrl}"
                       Aspect="AspectFill"
                       HeightRequest="120"
                       WidthRequest="120" />
                <Label Grid.Column="1"
                       Text="{Binding Details}"
                       FontAttributes="Italic" />
            </Grid>
        </DataTemplate>
    </Expander.ContentTemplate>
</Expander>
```

In this example, the `Expander` content is only inflated when the `Expander` expands for the first time.

The advantage of this approach is that when a page contains multiple `Expander` objects, the content for an `Expander` is only created when expanded for the first time by the user.

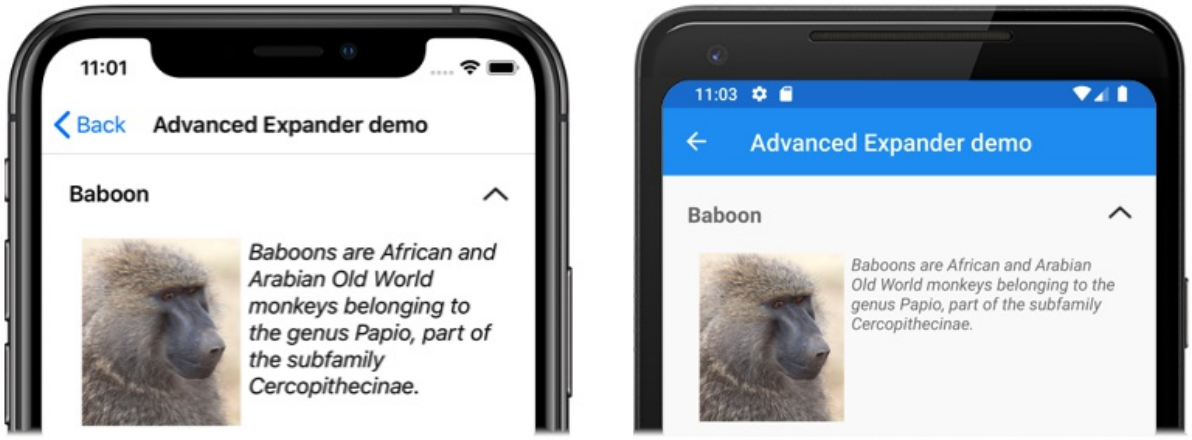
Add an expansion indicator

An `Image` can be added to an `Expander` header, to provide a visual indication of expansion state. A `DataTrigger` can be attached to the `Image`, that changes the `Source` property based on the value of the `Expander.IsExpanded` property:

```
<Expander>
    <Expander.Header>
        <Grid>
            <Label Text="{Binding Name}"
                   FontAttributes="Bold"
                   FontSize="Medium" />
            <Image Source="expand.png"
                   HorizontalOptions="End"
                   VerticalOptions="Start">
                <Image.Triggers>
                    <DataTrigger TargetType="Image"
                                 Binding="{Binding Source={RelativeSource AncestorType={x:Type Expander}},
Path=IsExpanded}"
                                 Value="True">
                        <Setter Property="Source"
                                Value="collapse.png" />
                    </DataTrigger>
                </Image.Triggers>
            </Image>
        </Grid>
    </Expander.Header>
    <Expander.ContentTemplate>
        <DataTemplate>
            <Grid Padding="10">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto" />
                    <ColumnDefinition Width="Auto" />
                </Grid.ColumnDefinitions>
                <Image Source="{Binding ImageUrl}"
                       Aspect="AspectFill"
                       HeightRequest="120"
                       WidthRequest="120" />
                <Label Grid.Column="1"
                       Text="{Binding Details}"
                       FontAttributes="Italic" />
            </Grid>
        </DataTemplate>
    </Expander.ContentTemplate>
</Expander>
```

In this example, the `Image` displays the `expand` icon by default:

The `IsExpanded` property becomes `true` when the `Expander` header is tapped, which results in the `collapse` icon being displayed:



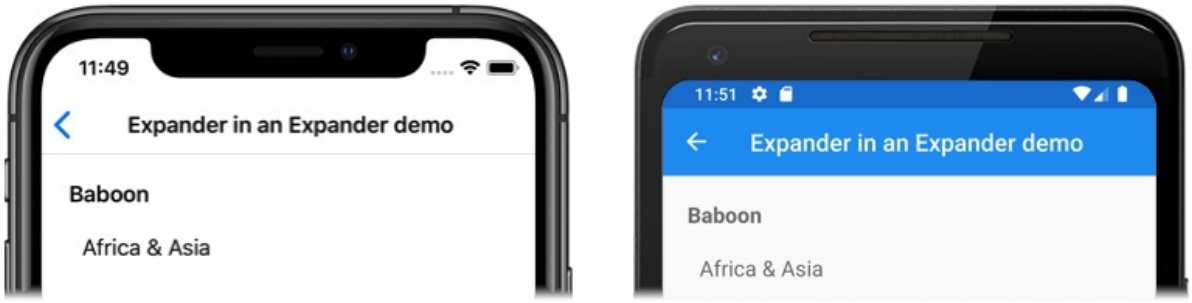
For more information about triggers, see Xamarin.Forms Triggers.

Embed an Expander in an Expander

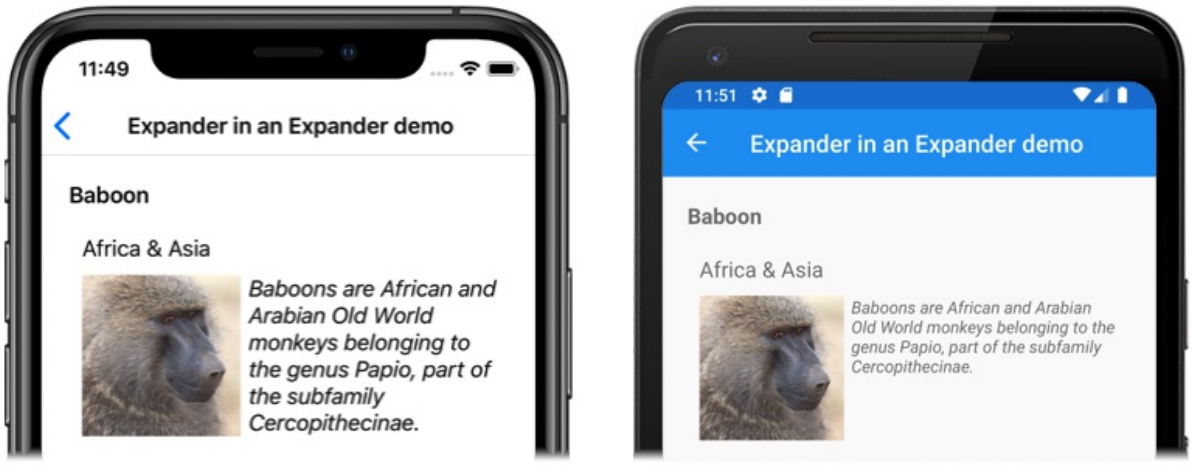
The content of an `Expander` can be set to another `Expander` control, to enable multiple levels of expansion. The following XAML shows an `Expander` whose content is another `Expander` object:

```
<Expander>
    <Expander.Header>
        <Label Text="{Binding Name}"
               FontAttributes="Bold"
               FontSize="Medium" />
    </Expander.Header>
    <Expander Padding="10">
        <Expander.Header>
            <Label Text="{Binding Location}"
                   FontSize="Medium" />
        </Expander.Header>
            <Expander.ContentTemplate>
                <DataTemplate>
                    <Grid>
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="Auto" />
                            <ColumnDefinition Width="Auto" />
                        </Grid.ColumnDefinitions>
                        <Image Source="{Binding ImageUrl}"
                               Aspect="AspectFill"
                               HeightRequest="120"
                               WidthRequest="120" />
                        <Label Grid.Column="1"
                               Text="{Binding Details}"
                               FontAttributes="Italic" />
                    </Grid>
                </DataTemplate>
            </Expander.ContentTemplate>
    </Expander>
</Expander>
```

In this example, tapping the root `Expander` header reveals the header for the child `Expander`:



Tapping the child `Expander` header results in its content being inflated and displayed:



Define the expand and collapse animation

The animation that occurs when an `Expander` expands or collapses can be defined by setting the `ExpandAnimationEasing` and `CollapseAnimationEasing` properties to any of the easing functions included in Xamarin.Forms, or custom easing functions. By default, the expand and collapse animations occur over 250ms. However, these durations can be changed by setting the `ExpandAnimationLength` and `CollapseAnimationLength` properties to `uint` values.

The following XAML shows an example of defining the animation that occurs when the `Expander` is expanded or collapsed by the user:

```
<Expander ExpandAnimationEasing="{CubicIn}"
          ExpandAnimationLength="500"
          CollapseAnimationEasing="{CubicOut}"
          CollapseAnimationLength="500">
    <Expander.Header>
        <Label Text="{Binding Name}"
               FontAttributes="Bold"
               FontSize="Medium" />
    </Expander.Header>
    <Expander.ContentTemplate>
        <DataTemplate>
            <Grid Padding="10">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto" />
                    <ColumnDefinition Width="Auto" />
                </Grid.ColumnDefinitions>
                <Image Source="{Binding ImageUrl}"
                       Aspect="AspectFill"
                       HeightRequest="120"
                       WidthRequest="120" />
                <Label Grid.Column="1"
                       Text="{Binding Details}"
                       FontAttributes="Italic" />
            </Grid>
        </DataTemplate>
    </Expander.ContentTemplate>
</Expander>
```

In this example, the `CubicIn` easing function slowly accelerates the expand animation over 500ms, and the `CubicOut` easing function quickly decelerates the collapse animation over 500ms.

For more information about easing functions, see Xamarin.Forms Easing Functions.

Resize an Expander at runtime

An `Expander` can be programmatically resized at runtime with the `ForceUpdateSize` method.

Given an `Expander` named `expander`, whose content includes a `Label` that has a `TapGestureRecognizer` attached to it, the following code example shows calling the `ForceUpdateSize` method:

```
void OnLabelTapped(object sender, EventArgs e)
{
    Label label = sender as Label;
    Expander expander = label.Parent.Parent.Parent as Expander;

    if (label.FontSize == Device.GetNamedSize(NamedSize.Default, label))
    {
        label.FontSize = Device.GetNamedSize(NamedSize.Large, label);
    }
    else
    {
        label.FontSize = Device.GetNamedSize(NamedSize.Default, label);
    }
    expander.ForceUpdateSize();
}
```

In this example, the `FontSize` of a `Label` changes when the `Label` is tapped. Due to the size of the font changing, it's necessary to update the size of the `Expander` by calling its `ForceUpdateSize` method.

Disable an Expander

An application may enter a state where expanding an `Expander` is not a valid operation. In such cases, the `Expander` can be disabled by setting its `IsEnabled` property to false. This will prevent users from expanding or collapsing the `Expander`.

Related links

- Expander Demos (sample)
- Xamarin.Forms Easing Functions
- Xamarin.Forms Triggers
- Xamarin.Forms Bindable Layouts

Xamarin Community Toolkit LazyView

4/7/2021 • 2 minutes to read • Edit Online

The `LazyView` control allows you to delay the initialization of a `View`. You need to provide the type of the `View` that you want to be rendered, using the `x:TypeArguments` XAML namespace attribute, and handle its initialization using the `LoadViewAsync` method. The `IsLoaded` property can be examined to determine when the `LazyView` is loaded.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage"
             xmlns:local="clr-namespace:Xamarin.CommunityToolkit.Sample.Pages.Views.TabView">

     <StackLayout>

        <xct:LazyView x:TypeArguments="local:LazyTestView" IsLoaded = "{Binding IsLoaded}" />

    </StackLayout>

</ContentPage>
```

Properties

PROPERTY	TYPE	DESCRIPTION
IsLoaded	bool	Gets the loaded status of the `LazyView`.

Methods

PROPERTY	RETURN TYPE	DESCRIPTION
LoadViewAsync	ValueTask	Initialize the `View`.
Dispose	void	Cleans up the `View`, if required.

Sample

LazyView sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

Related video

Xamarin Community Toolkit MediaElement

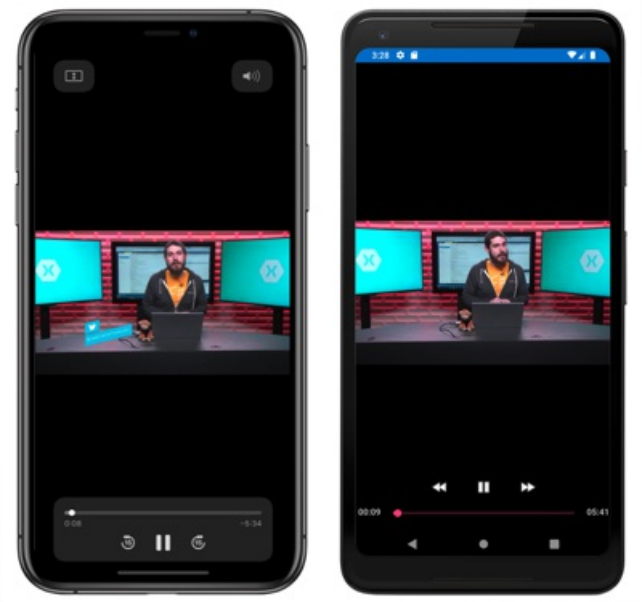
5/14/2021 • 14 minutes to read • Edit Online

Download the sample

`MediaElement` is a view for playing video and audio. Media that's supported by the underlying platform can be played from the following sources:

- The web, using a URI (HTTP or HTTPS).
- A resource embedded in the platform application, using the `ms-appx:///` URI scheme.
- Files that come from the app's local and temporary data folders, using the `ms-appdata:///` URI scheme.
- The device's library.

`MediaElement` can use the platform playback controls, which are referred to as transport controls. However, they are disabled by default and can be replaced with your own transport controls. The following screenshots show `MediaElement` playing a video with the platform transport controls:



NOTE

`MediaElement` is available on iOS, Android, the Universal Windows Platform (UWP), macOS, Windows Presentation Foundation, and Tizen.

`MediaElement` defines the following properties:

- `Aspect`, of type `Aspect`, determines how the media will be scaled to fit the display area. The default value of this property is `AspectFit`.
- `AutoPlay`, of type `bool`, indicates whether media playback will begin automatically when the `Source` property is set. The default value of this property is `true`.
- `BufferingProgress`, of type `double`, indicates the current buffering progress. The default value of this property is 0.0.
- `CanSeek`, of type `bool`, indicates whether media can be repositioned by setting the value of the `Position` property. This is a read-only property.

- `CurrentState`, of type `MediaElementState`, indicates the current status of the control. This is a read-only property, whose default value is `MediaElementState.Closed`.
- `Duration`, of type `TimeSpan?`, indicates the duration of the currently opened media. This is a read-only property whose default value is `null`.
- `IsLooping`, of type `bool`, describes whether the currently loaded media source should resume playback from the start after reaching its end. The default value of this property is `false`.
- `KeepScreenOn`, of type `bool`, determines whether the device screen should stay on during media playback. The default value of this property is `false`.
- `Position`, of type `TimeSpan`, describes the current progress through the media's playback time. This property uses a `TwoWay` binding, and its default value is `TimeSpan.Zero`.
- `ShowsPlaybackControls`, of type `bool`, determines whether the platforms playback controls are displayed. The default value of this property is `false`. Note that on iOS the controls are only shown for a brief period after interacting with the screen. There is no way of keeping the controls visible at all times. On WPF, no system controls are supported so this property has no effect.
- `Source`, of type `MediaSource`, indicates the source of the media loaded into the control.
- `VideoHeight`, of type `int`, indicates the height of the control. This is a read-only property.
- `VideoWidth`, of type `int`, indicates the width of the control. This is a read-only property.
- `Volume`, of type `double`, determines the media's volume, which is represented on a linear scale between 0 and 1. This property uses a `TwoWay` binding, and its default value is 1.

These properties, with the exception of the `CanSeek` property, are backed by `BindableProperty` objects, which means that they can be targets of data bindings, and styled.

The `MediaElement` class also defines four events:

- `MediaOpened` is fired when the media stream has been validated and opened.
- `MediaEnded` is fired when the `MediaElement` finishes playing its media.
- `MediaFailed` is fired when there's an error associated with the media source.
- `SeekCompleted` is fired when the seek point of a requested seek operation is ready for playback.

In addition, `MediaElement` includes `Play`, `Pause`, and `Stop` methods.

For information about supported media formats on Android, see Supported media formats on developer.android.com. For information about supported media formats on the Universal Windows Platform (UWP), see Supported codecs.

Play remote media

A `MediaElement` can play remote media files using the HTTP and HTTPS URI schemes. This is accomplished by setting the `Source` property to the URI of the media file:

```
<MediaElement Source="https://sec.ch9.ms/ch9/5d93/a1eab4bf-3288-4faf-81c4-294402a85d93/XamarinShow_mid.mp4"
              ShowsPlaybackControls="True" />
```

By default, the media that is defined by the `Source` property plays immediately after the media is opened. To suppress automatic media playback, set the `AutoPlay` property to `false`.

Media playback controls are disabled by default, and are enabled by setting the `ShowsPlaybackControls` property to `true`. `MediaElement` will then use the platform playback controls where available.

Play local media

Local media can be played from the following sources:

- A resource embedded in the platform application, using the `ms-appx:///` URI scheme.
- Files that come from the app's local and temporary data folders, using the `ms-appdata:///` URI scheme.
- The device's library.

For more information about these URI schemes, see URI schemes.

Play media embedded in the app package

A `MediaElement` can play media files that are embedded in the app package, using the `ms-appx:///` URI scheme. Media files are embedded in the app package by placing them in the platform project.

Storing a media file in the platform project is different for each platform:

- On iOS, media files must be stored in the **Resources** folder, or a subfolder of the **Resources** folder. The media file must have a `Build Action` of `BundleResource`.
- On Android, media files must be stored in a subfolder of **Resources** named **raw**. The **raw** folder cannot contain subfolders. The media file must have a `Build Action` of `AndroidResource`.
- On UWP, media files can be stored in any folder in the project. The media file must have a `BuildAction` of `Content`.

Media files that meet these criteria can then be played back using the `ms-appx:///` URI scheme:

```
<MediaElement Source="ms-appx:///XamarinForms101UsingEmbeddedImages.mp4"
              ShowsPlaybackControls="True" />
```

When using data binding, a value converter can be used to apply this URI scheme:

```
public class VideoSourceConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        if (value == null)
            return null;

        if (string.IsNullOrWhiteSpace(value.ToString()))
            return null;

        if (Device.RuntimePlatform == Device.UWP)
            return new Uri($"ms-appx:///Assets/{value}");
        else
            return new Uri($"ms-appx:///{value}");
    }
    // ...
}
```

An instance of the `VideoSourceConverter` can then be used to apply the `ms-appx:///` URI scheme to an embedded media file:

```
<MediaElement Source="{Binding MediaSource, Converter={StaticResource VideoSourceConverter}}"
              ShowsPlaybackControls="True" />
```

For more information about the ms-appx URI scheme, see ms-appx and ms-appx-web.

Play media from the app's local and temporary folders

A `MediaElement` can play media files that are copied into the app's local or temporary data folders, using the `ms-appdata:///` URI scheme.

The following example shows the `Source` property set to a media file that's stored in the app's local data folder:

```
<MediaElement Source="ms-appdata:///local/XamarinVideo.mp4"
              ShowsPlaybackControls="True" />
```

The following example shows the `Source` property to a media file that's stored in the app's temporary data folder:

```
<MediaElement Source="ms-appdata:///temp/XamarinVideo.mp4"
              ShowsPlaybackControls="True" />
```

> **IMPORTANT**
> In addition to playing media files that are stored in the app's local or temporary data folders, UWP can also play media files that are located in the app's roaming folder. This can be achieved by prefixing the media file with `ms-appdata:///roaming/`.

When using data binding, a value converter can be used to apply this URI scheme:

```
public class VideoSourceConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        if (value == null)
            return null;

        if (string.IsNullOrWhiteSpace(value.ToString()))
            return null;

        return new Uri($"ms-appdata:///{value}");
    }
    // ...
}
```

An instance of the `VideoSourceConverter` can then be used to apply the `ms-appdata:///` URI scheme to a media file in the app's local or temporary data folder:

```
<MediaElement Source="{Binding MediaSource, Converter={StaticResource VideoSourceConverter}}"
              ShowsPlaybackControls="True" />
```

For more information about the ms-appdata URI scheme, see ms-appdata.

Copying a media file to the app's local or temporary data folder

Playing a media file stored in the app's local or temporary data folder requires the media file to be copied there by the app. This can be accomplished, for example, by copying a media file from the app package:

```
// This method copies the video from the app package to the app data
// directory for your app. To copy the video to the temp directory
// for your app, comment out the first line of code, and uncomment
// the second line of code.
public static async Task CopyVideoIfNotExists(string filename)
{
    string folder = FileSystem.AppDataDirectory;
    //string folder = Path.GetTempPath();
    string videoFile = Path.Combine(folder, "XamarinVideo.mp4");

    if (!File.Exists(videoFile))
    {
        using (Stream inputStream = await FileSystem.OpenAppPackageFileAsync(filename))
        {
            using (FileStream outputStream = File.Create(videoFile))
            {
                await inputStream.CopyToAsync(outputStream);
            }
        }
    }
}
```

NOTE

The code example above uses the `FileSystem` class included in Xamarin.Essentials. For more information, see Xamarin.Essentials: File System Helpers.

Play media from the device library

Most modern mobile devices and desktop computers have the ability to record videos and audio using the device's camera and microphone. The media that's created are then stored as files on the device. These files can be retrieved from the library and played by the `MediaElement`.

Each of the platforms includes a facility that allows the user to select media from the device's library. In Xamarin.Forms, platform projects can invoke this functionality, and it can be called by the `DependencyService` class.

The video picking dependency service used in the sample application is very similar to one defined in Picking a Photo from the Picture Library, except that the picker returns a filename rather than a `Stream` object. The shared code project defines an interface named `IVideoPicker`, that defines a single method named `GetVideoFileAsync`. Each platform then implements this interface in a `VideoPicker` class.

The following code example shows how to retrieve a media file from the device library:

```
string filename = await DependencyService.Get<IVideoPicker>().GetVideoFileAsync();
if (!string.IsNullOrWhiteSpace(filename))
{
    mediaElement.Source = new FileMediaSource
    {
        File = filename
    };
}
```

The video picking dependency service is invoked by calling the `DependencyService.Get` method to obtain the implementation of an `IVideoPicker` interface in the platform project. The `GetVideoFileAsync` method is then called on that instance, and the returned filename is used to create a `FileMediaSource` object and to set it to the `Source` property of the `MediaElement`.

Change video aspect ratio

The `Aspect` property determines how video media will be scaled to fit the display area. By default, this property is set to the `AspectFit` enumeration member, but it can be set to any of the `Aspect` enumeration members:

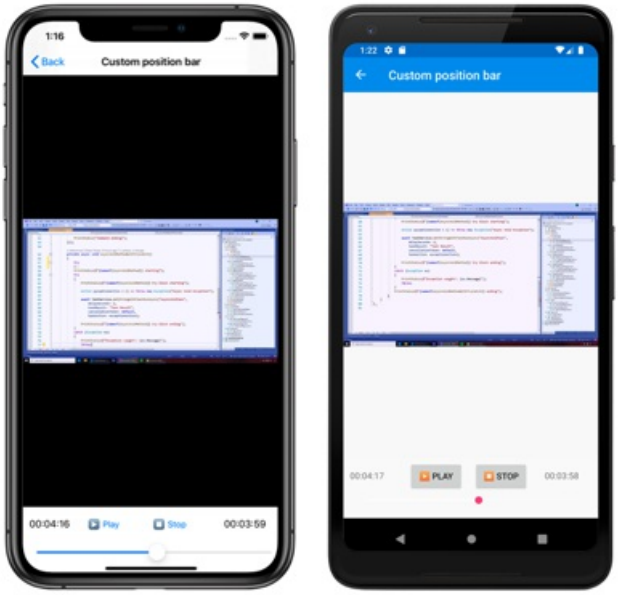
- `AspectFit` indicates that the video will be letterboxed, if required, to fit into the display area, while preserving the aspect ratio.
- `AspectFill` indicates that the video will be clipped so that it fills the display area, while preserving the aspect ratio.
- `Fill` indicates that the video will be stretched to fill the display area.

Binding to the Position property

The property change notification for the `Position` bindable property fire at 200ms intervals while playing. Therefore, the property can be data-bound to a `Slider` control (or similar) to show progress through the media. The CommunityToolkit also provides a `TimeSpanToDoubleConverter` which converts a `TimeSpan` into a floating point value representing total seconds elapsed. In this way you can set the Slider `Maximum` to the `Duration` of the media and the `Value` to the `Position` to provide accurate progress:

```
<?xml version="1.0" encoding="UTF-8"?>
<pages:BasePage xmlns="http://xamarin.com/schemas/2014/forms"
                xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
                xmlns:pages="clr-namespace:Xamarin.CommunityToolkit.Sample.Pages"
                x:Class="Xamarin.CommunityToolkit.Sample.Pages.Views.MediaElementPage">
    <pages:BasePage.Resources>
        <xct:TimeSpanToDoubleConverter x:Key="TimeSpanConverter"/>
    </pages:BasePage.Resources>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <xct:MediaElement
            x:Name="mediaElement"
            Source="https://sec.ch9.ms/ch9/5d93/a1eab4bf-3288-4faf-81c4-294402a85d93/XamarinShow_mid.mp4"
            ShowsPlaybackControls="True"
            HorizontalOptions="Fill"
            SeekCompleted="OnSeekCompleted" />
        <Slider Grid.Row="1" BindingContext="{x:Reference mediaElement}" Value="{Binding Position,
Converter={StaticResource TimeSpanConverter}}" Maximum="{Binding Duration, Converter={StaticResource
TimeSpanConverter}}">
            <Slider.Triggers>
                <DataTrigger TargetType="Slider"
                     Binding="{Binding CurrentState}"
                     Value="{x:Static MediaElementState.Buffering}">
                    <Setter Property="IsEnabled" Value="False" />
                </DataTrigger>
            </Slider.Triggers>
        </Slider>
        <Button Grid.Row="2" Text="Reset Source (Set Null)" Clicked="OnResetClicked" />
    </Grid>
</pages:BasePage>
```

In this example, the `Maximum` property of the `Slider` is data-bound to the `Duration` property of the `MediaElement` and the `Value` property of the `Slider` is data-bound to the `Position` property of the `MediaElement`. Therefore, dragging the `Slider` results in the media playback position changing:



In addition, a `DataTrigger` object is used to disable the `Slider` when the media is buffering. For more information about data triggers, see Xamarin.Forms Triggers.

> **NOTE**
> On Android, the `Slider` only has 1000 discrete steps, regardless of the `Minimum` and `Maximum` settings. If the media length is greater than 1000 seconds, then two different `Position` values would correspond to the same `Value` of the `Slider`. This is why the code above checks that the new position and existing position are greater than one-hundredth of the overall duration.

Understand MediaSource types

A `MediaElement` can play media by setting its `Source` property to a remote or local media file. The `Source` property is of type `MediaSource`, and this class defines two static methods:

- `FromFile`, returns a `MediaSource` instance from a `string` argument.
- `FromUri`, returns a `MediaSource` instance from a `Uri` argument.

In addition, the `MediaSource` class also has implicit operators that return `MediaSource` instances from `string` and `Uri` arguments.

> **NOTE**
> When the `Source` property is set in XAML, a type converter is invoked to return a `MediaSource` instance from a `string` or `Uri`.

The `MediaSource` class also has two derived classes:

- `UriMediaSource`, which is used to specify a remote media file from a URI. This class has a `Uri` property that can be set to a `Uri`.
- `FileMediaSource`, which is used to specify a local media file from a `string`. This class has a `File` property that can be set to a `string`. In addition, this class has implicit operators to convert a `string` to a `FileMediaSource` object, and a `FileMediaSource` object to a `string`.

> **NOTE**
> When a `FileMediaSource` object is created in XAML, a type converter is invoked to return a `FileMediaSource` instance from a `string`.

Determine MediaElement status

The `MediaElement` class defines a read-only bindable property named `CurrentState`, of type `MediaElementState`. This property indicates the current status of the control, such as whether the media is playing or paused, or if it's not yet ready to play the media.

The `MediaElementState` enumeration defines the following members:

- `Closed` indicates that the `MediaElement` contains no media.
- `Opening` indicates that the `MediaElement` is validating and attempting to load the specified source.
- `Buffering` indicates that the `MediaElement` is loading the media for playback. Its `Position` property does not advance during this state. If the `MediaElement` was playing video, it continues to display the last displayed frame.
- `Playing` indicates that the `MediaElement` is playing the media source.
- `Paused` indicates that the `MediaElement` does not advance its `Position` property. If the `MediaElement` was playing video, it continues to display the current frame.
- `Stopped` indicates that the `MediaElement` contains media but it is not being played or paused. Its `Position` property is 0 and does not advance. If the loaded media is video, the `MediaElement` displays the first frame.

It's generally not necessary to examine the `CurrentState` property when using the `MediaElement` transport controls. However, this property becomes important when implementing your own transport controls.

Implement custom transport controls

The transport controls of a media player include the buttons that perform the functions **Play**, **Pause**, and **Stop**. These buttons are generally identified with familiar icons rather than text, and the **Play** and **Pause** functions are generally combined into one button.

By default, the `MediaElement` playback controls are disabled. This enables you to control the `MediaElement` programmatically, or by supplying your own transport controls. In support of this, `MediaElement` includes `Play`, `Pause`, and `Stop` methods.

The following XAML example shows a page that contains a `MediaElement` and custom transport controls:

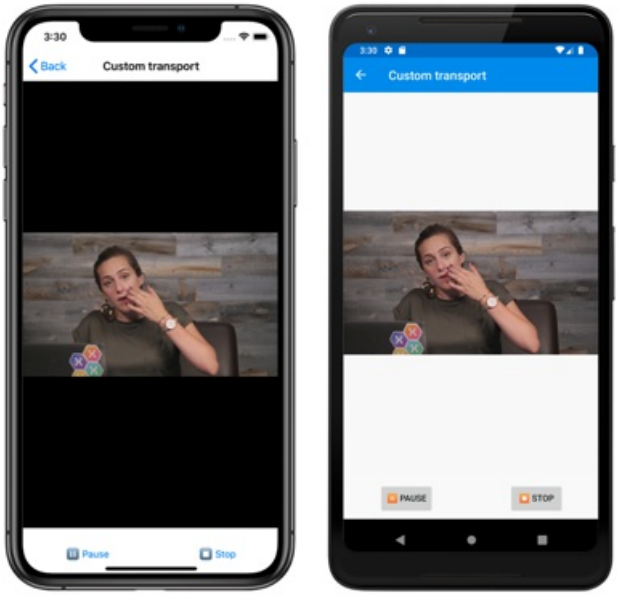
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MediaElementDemos.CustomTransportPage"
             Title="Custom transport">
    <Grid>
        ...
        <MediaElement x:Name="mediaElement"
                      AutoPlay="False"
                      ... />
        <StackLayout BindingContext="{x:Reference mediaElement}"
                     ...>
            <Button Text="&#x25B6;&#xFE0F; Play"
                    HorizontalOptions="CenterAndExpand"
                    Clicked="OnPlayPauseButtonClicked">
                <Button.Triggers>
                    <DataTrigger TargetType="Button"
                                 Binding="{Binding CurrentState}"
                                 Value="{x:Static MediaElementState.Playing}">
                        <Setter Property="Text"
                                Value="&#x23F8; Pause" />
                    </DataTrigger>
                    <DataTrigger TargetType="Button"
                                 Binding="{Binding CurrentState}"
                                 Value="{x:Static MediaElementState.Buffering}">
                        <Setter Property="IsEnabled"
                                Value="False" />
                    </DataTrigger>
                </Button.Triggers>
            </Button>
            <Button Text="&#x23F9; Stop"
                    HorizontalOptions="CenterAndExpand"
                    Clicked="OnStopButtonClicked">
                <Button.Triggers>
                    <DataTrigger TargetType="Button"
                                 Binding="{Binding CurrentState}"
                                 Value="{x:Static MediaElementState.Stopped}">
                        <Setter Property="IsEnabled"
                                Value="False" />
                    </DataTrigger>
                </Button.Triggers>
            </Button>
        </StackLayout>
    </Grid>
</ContentPage>
```

In this example, the custom transport controls are defined as `Button` objects. However, there are only two `Button` objects, with the first `Button` representing **Play** and **Pause**, and the second `Button` representing **Stop**. `DataTrigger` objects are used to enable and disable the buttons, and to switch the first button between **Play** and **Pause**. For more information about data triggers, see Xamarin.Forms Triggers.

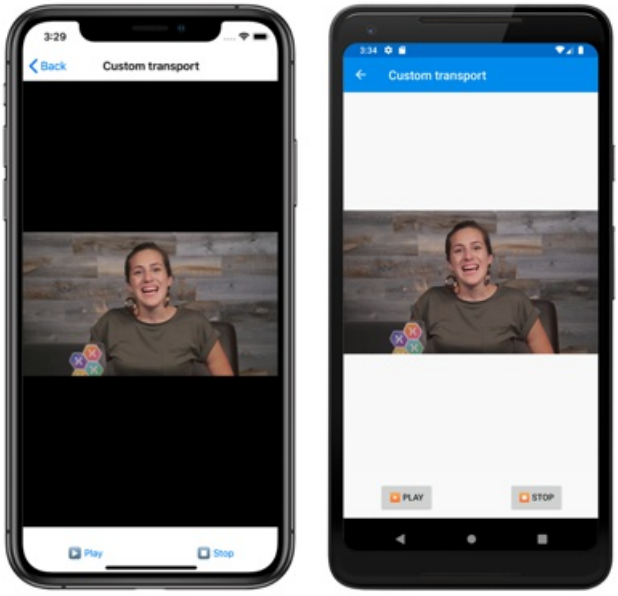
The code-behind file has the handlers for the `Clicked` events:

```
void OnPlayPauseButtonClicked(object sender, EventArgs args)
{
    if (mediaElement.CurrentState == MediaElementState.Stopped ||
        mediaElement.CurrentState == MediaElementState.Paused)
    {
        mediaElement.Play();
    }
    else if (mediaElement.CurrentState == MediaElementState.Playing)
    {
        mediaElement.Pause();
    }
}

void OnStopButtonClicked(object sender, EventArgs args)
{
    mediaElement.Stop();
}
```

The **Play** button can be pressed, once it becomes enabled, to begin playback:



Pressing the **Pause** button results in playback pausing:



Pressing the **Stop** button stops playback and returns the position of the media file to the beginning.

Implement a custom volume control

Media playback controls implemented by each platform include a volume bar. This bar resembles a slider and shows the volume of the media. In addition, you can manipulate the volume bar to increase or decrease the volume.

A custom volume bar can be implemented using a `Slider`, as shown in the following example:

```
<StackLayout>
    <MediaElement AutoPlay="False"
                  Source="{StaticResource AdvancedAsync}" />
    <Slider Maximum="1.0"
            Minimum="0.0"
            Value="{Binding Volume}"
            Rotation="270"
            WidthRequest="100" />
</StackLayout>
```

In this example, the `Slider` data binds its `Value` property to the `Volume` property of the `MediaElement`. This is possible because the `Volume` property uses a `TwoWay` binding. Therefore, changing the `Value` property will result in the `Volume` property changing.

> **NOTE**
>
> The `Volume` property has a validation callback that ensures that its value is greater than or equal to 0.0, and less than or equal to 1.0.

For more information about using a `Slider` see, Xamarin.Forms Slider

Related links

- MediaElementDemos (sample)
- URI schemes
- Xamarin.Forms Triggers
- Xamarin.Forms Slider
- Android: Supported media formats
- UWP: Supported codecs

Xamarin Community Toolkit RangeSlider

3/5/2021 • 4 minutes to read • Edit Online

The RangeSlider control enables the user to select a range of values through a slider bar interface. As opposed to a regular `Slider` that lets you select a single value by sliding the thumb, this control has two thumbs that allows the user to specify a range.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

     <StackLayout>

        <xct:RangeSlider
                x:Name="RangeSlider"
                MaximumValue="10"
                MinimumValue="-10"
                StepValue="1"
                LowerValue="-10"
                UpperValue="10" />

    </StackLayout>

</ContentPage>
```

Properties

PROPERTY	TYPE	DESCRIPTION
LowerThumbBorderColor	Color	Gets or sets the border color of the lower thumb for the RangeSlider. This is a bindable property.
LowerThumbColor	Color	Gets or sets the color of the lower thumb for the RangeSlider. This is a bindable property.
LowerThumbRadius	double	Gets or sets the corner radius of the lower thumb for the RangeSlider. This is a bindable property.
LowerThumbSize	double	Gets or sets the size of the lower thumb for the RangeSlider. This is a bindable property.
LowerThumbView	View	Gets or sets a custom view to be used for the lower thumb of the RangeSlider. This is a bindable property.

PROPERTY	TYPE	DESCRIPTION
LowerValue	double	Gets or sets the lower value of the range for the RangeSlider. This is a bindable property.
LowerValueLabelStyle	Style	Gets or sets the style used for the value label on the lower thumb for the RangeSlider. This is a bindable property.
MaximumValue	double	Gets or sets the maximum value for the range that can be selected with the RangeSlider. This is a bindable property.
MinimumValue	double	Gets or sets the minimum value for the range that can be selected with the RangeSlider. This is a bindable property.
StepValue	double	Gets or sets the increment by which MaximumValue and MinimumValue change for the RangeSlider.
ThumbBorderColor	Color	Gets or sets the border color of both the lower and the upper thumb for the RangeSlider. This is a bindable property.
ThumbColor	Color	Gets or sets the color for both the lower and the upper thumb for the RangeSlider. This is a bindable property.
ThumbSize	double	Gets or sets size for both the lower and the upper thumb for the RangeSlider. This is a bindable property.
TrackBorderColor	Color	Gets or sets the color of the track bar border for the RangeSlider. This is a bindable property.
TrackColor	Color	Gets or sets the color of the track bar for the RangeSlider. This is a bindable property.
TrackHighlightBorderColor	Color	Gets or sets the border highlight color of the track bar, which is the part between the lower and the upper thumb, for the RangeSlider. This is a bindable property.

PROPERTY	TYPE	DESCRIPTION
TrackHighlightColor	Color	Gets or sets highlight color of the track, which is the part between the lower and the upper thumb, for the RangeSlider. This is a bindable property.
TrackRadius	double	Gets or sets the corner radius of the track for the RangeSlider. This is a bindable property.
TrackSize	double	Gets or sets the size of the track bar for the RangeSlider. This is a bindable property.
UpperThumbBorderColor	Color	Gets or sets the border color of the upper thumb for the RangeSlider. This is a bindable property.
UpperThumbColor	Color	Gets or sets the color of the upper thumb for the RangeSlider. This is a bindable property.
UpperThumbRadius	double	Gets or sets the corner radius of the upper thumb for the RangeSlider. This is a bindable property.
UpperThumbSize	double	Gets or sets the size of the upper thumb for the RangeSlider. This is a bindable property.
UpperThumbView	View	Gets or sets a custom view to be used for the upper thumb of the RangeSlider. This is a bindable property.
UpperValue	double	Gets or sets the upper value of the range for the RangeSlider. This is a bindable property.
UpperValueLabelStyle	Style	Gets or sets the style used for the value label on the upper thumb for the RangeSlider. This is a bindable property.
ValueLabelStringFormat	double	Gets or sets the string format used for both the value labels on the lower and upper thumb for the RangeSlider. This is a bindable property.
ValueLabelSpacing	double	Gets or sets he spacing of the font used for both the value labels on the lower and upper thumb for the RangeSlider. This is a bindable property.

PROPERTY	TYPE	DESCRIPTION
ValueLabelStyle	Style	Gets or sets the style used for both the value labels on the lower and upper thumb of the RangeSlider. This is a bindable property.

Events

PROPERTY	TYPE	DESCRIPTION
ValueChanged	EventHandler	Occurs when the value changes.
LowerValueChanged	EventHandler	Occurs when the lower value of the range changes.
UpperValueChanged	EventHandler	Occurs when the upper value of the range changes.
DragStarted	EventHandler	Occurs when a drag action is started on the lower or upper thumb.
LowerDragStarted	EventHandler	Occurs when a drag action is started with the lower thumb.
UpperDragStarted	EventHandler	Occurs when a drag action is started with the upper thumb.
DragCompleted	EventHandler	Occurs when a drag action is completed on the lower or upper thumb.
LowerDragCompleted	EventHandler	Occurs when a drag action is completed on the lower thumb.
UpperDragCompleted	EventHandler	occurs when a drag action is completed on the upper thumb.

Sample

RangeSlider sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

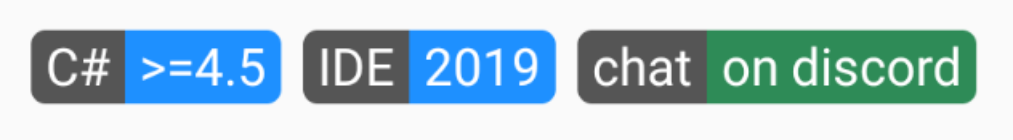
API

- RangeSlider source code

Xamarin Community Toolkit Shield

3/5/2021 • 2 minutes to read • Edit Online

The Shield is a type of badge that has two colored sections that contain text:



Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

     <StackLayout>

        <xct:Shield
                Subject="chat"
                Status="on discord"
                StatusBackgroundColor="SeaGreen"
                StatusTextColor="White"
                Margin="2,0" />

    </StackLayout>

</ContentPage>
```

Properties

PROPERTY	TYPE	DESCRIPTION
Command	`ICommand`	Gets or sets a `Command` that is invoked when the `Shield` is tapped. This is a bindable property.
CommandParameter	object	Gets or sets an object that is used as a parameter value for the invoked `Command` when the `Shield` is tapped. This is a bindable property.
FontAttributes	`FontAttributes`	Gets or sets the font attributes to be used for the text of the `Shield`. This is a bindable property.
FontFamily	double	Gets or sets the font family for the text of the `Shield`. This is a bindable property.

PROPERTY	TYPE	DESCRIPTION
FontSize	double	Gets or sets the font size for the text of the `Shield`. `NamedSize` values can be used. This is a bindable property.
Status	string	Gets or sets the text for the right (`Status`) part of the `Shield`. This is a bindable property.
StatusBackgroundColor	`Color`	Gets or sets the color for the right (`Status`) part of the `Shield`. This is a bindable property.
StatusTextColor	`Color`	Gets or sets the text color for the right (`Status`) part of the `Shield`. This is a bindable property.
Subject	string	Gets or sets the text for the left (`Subject`) part of the `Shield`. This is a bindable property.
SubjectBackgroundColor	`Color`	Gets or sets the color for the left (`Subject`) part of the `Shield`. This is a bindable property.
SubjectTextColor	`Color`	Gets or sets the text color for the left (`Subject`) part of the `Shield`. This is a bindable property.

NOTE

`TextColor` is deprecated since v1.1. Please use `StatusTextColor` instead. `Color` is deprecated since v1.1. Please use `StatusBackgroundColor` instead.

Events

EVENT	TYPE	DESCRIPTION
Tapped	EventHandler	Occurs when the `Shield` is tapped.

Sample

Shield sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- Shield source code

Xamarin Community Toolkit StateLayout

3/5/2021 • 2 minutes to read • Edit Online

Displaying a specific view when your app is in a specific state is a common pattern throughout any mobile app. Examples range from creating loading views to overlay on the screen, or on a subsection of the screen. Empty state views can be created for when there's no data to display, and error state views can be displayed when an error occurs.

Getting started

The StateLayout control enables the user to turn any layout element like a `Grid` or `StackLayout` into an individual state-aware element. Each layout that you make state-aware, using the StateLayout attached properties, contains a collection of `StateView` objects. These objects can be used as templates for the different states supported by StateLayout. Whenever the `CurrentState` property is set to a value that matches the `State` property of one of the StateViews its contents will be displayed instead of the main content.

LayoutState enumeration

The `LayoutState` enumeration supports one of the following values:

- `None` (default, this will show the initial view)
- `Loading`
- `Saving`
- `Success`
- `Error`
- `Empty`
- `Custom`

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

     <Grid xct:StateLayout.CurrentState="{Binding CurrentState}">
      <xct:StateLayout.StateViews>
         <xct:StateView StateKey="Loading">
            <Grid BackgroundColor="White">
               <StackLayout VerticalOptions="Center" HorizontalOptions="Center">
                  <ActivityIndicator Color="#1abc9c" />
                  <Label Text="Loading..." HorizontalOptions="Center" />
               </StackLayout>
            </Grid>
         </xct:StateView>
      </xct:StateLayout.StateViews>

     ...

  </Grid>

</ContentPage>
```

Use custom states

Besides the built-in states StateLayout also supports a `Custom` state. By setting `State` to `Custom` and adding a `CustomStateKey` you can create custom states beyond the built-in states. You can use the `CurrentCustomStateKey` on your root StateLayout element to bind a variable that indicates when to show one of your custom states.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="clr-namespace:Xamarin.CommunityToolkit.Views;assembly=Xamarin.CommunityToolkit"
             x:Class="MyLittleApp.MainPage">

    <StackLayout Padding="10" xct:StateLayout.CurrentState="{Binding CurrentState}"
xct:StateLayout.CurrentCustomStateKey="{Binding CustomState}">
        <xct:StateLayout.StateViews>
            <xct:StateView StateKey="Custom" CustomStateKey="ThisIsACustomState">
                <Label Text="Hi, I'm a custom state!" />
            </xct:StateView>
            <xct:StateView StateKey="Custom" CustomStateKey="ThisIsACustomStateToo">
                <Label Text="Hi, I'm a custom state too!" />
            </xct:StateView>
        </xct:StateLayout.StateViews>

        <Label Text="This is the normal state." />
    </StackLayout>

</ContentPage>
```

Use repeating states

When loading multiple items of the same type it could be beneficial to repeat a piece of XAML without having to copy paste it multiple times. This is where the `RepeatCount` property should be used. By defining a `Template` it's possible to repeat the same piece of XAML while only defining it once.

> **NOTE**
>
> When using a `RepeatCount` that is greater than 1 you have to use the `Template` property to provide your content. Providing `Content` to a `StateView` while providing a `RepeatCount` greater than 1 will result in an exception.

Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="clr-namespace:Xamarin.CommunityToolkit.Views;assembly=Xamarin.CommunityToolkit"
             x:Class="MyLittleApp.MainPage">

    <StackLayout xct:StateLayout.CurrentState="{Binding CurrentState}">
        <xct:StateLayout.StateViews>
            <xct:StateView StateKey="Loading" RepeatCount="3">
                <xct:StateView.Template>
                    <DataTemplate>
                        <StackLayout Spacing="8">
                            <BoxView CornerRadius="8" HeightRequest="40" BackgroundColor="#CCCCCC"
WidthRequest="120" />
                            <BoxView CornerRadius="8" HeightRequest="40" BackgroundColor="#CCCCCC"
WidthRequest="200" />
                        </StackLayout>
                    </DataTemplate>
                </xct:StateView.Template>
            </xct:StateView>
        </xct:StateLayout.StateViews>

        ...

    <StackLayout>
</ContentPage>
```

Properties

The following properties are available on the `StateLayout` object:

PROPERTY	TYPE	DESCRIPTION
CurrentState	`LayoutState`	Defines the current state of the layout and which template to show.
CurrentCustomStateKey	string	Pair this with `State="Custom"` on a StateView to add custom states.
StateViews	`IList<StateView>`	A list of StateView objects that contains a template per State.

The following properties are available on the `StateView` object:

PROPERTY	TYPE	DESCRIPTION
CustomStateKey	string	Defines the current state of the layout and which template to show.
RepeatCount	int	The amount of times the Template should be repeated.
Template	`DataTemplate`	Defines a template to show for this state.
State	`LayoutState`	Defines the current state of the layout and which template to show.

Sample

StateLayout sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- StateLayout source code

Related video

Xamarin Community Toolkit TabView

3/5/2021 • 3 minutes to read • Edit Online

The `TabView` control allows the user to display a set of tabs and their content. The `TabView` is fully customizable, other than the native tab bars.

Syntax

The following code shows a simple example of a `TabView` implementation:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

     <Grid>
        <xct:TabView
                TabStripPlacement="Bottom"
                TabStripBackgroundColor="Blue"
                TabStripHeight="60"
                TabIndicatorColor="Yellow"
                TabContentBackgroundColor="Yellow">

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

                <xct:TabViewItem
                    Icon="circle.png"
                    Text="Tab 2"
                    TextColor="White"
                    TextColorSelected="Yellow"
                    FontSize="12">
                    <Grid>
                        <Label
                            HorizontalOptions="Center"
                            VerticalOptions="Center"
                            Text="TabContent2" />
                    </Grid>
                </xct:TabViewItem>
        </xct:TabView>
  </Grid>

</ContentPage>
```

Properties

The following properties are available on the `TabView` object:

PROPERTY	TYPE	DESCRIPTION
IsSwipeEnabled	bool	Enable or disable the swipe gesture.
IsTabTransitionEnabled	bool	Enable or disable the transition between tabs.
SelectedIndex	int	Gets or sets the currently selected tab. Default is 0.
TabContentBackgroundColor	`Color`	The tab content background.
TabContentDataTemplate	`DataTemplate`	The template the TabView uses to generate tab items' content.
TabContentHeight	double	The tab content height.
TabIndicatorColor	`Color`	The TabIndicator background.
TabIndicatorHeight	double	The TabIndicator height.
TabIndicatorPlacement	TabIndicatorPlacement	The selected tab indicator placement (top, center or bottom).
TabIndicatorView	`View`	The TabIndicator content.
TabIndicatorWidth	double	The TabIndicator width.
TabItems	ObservableCollection<TabViewItem>	Property that reflects the current tab items.
TabItemsSource	`IList`	A collection used to generate the TabView's tab items.
TabStripPlacement	TabStripPlacement	The TabStrip placement (top or bottom).
TabStripBackgroundColor	`Color`	The `Color` of the TabStrip background. This can't be used with `TabStripBackgroundView`.
TabStripBackgroundView	`View`	The `View` representing the TabStrip background. This can't be used with `TabStripBackgroundColor`.
TabStripHeight	double	The TabStrip height.
TabViewItemDataTemplate	`DataTemplate`	The template the TabView uses to generate tab items' header.

The following properties are available on the `TabViewItem` object:

PROPERTY	TYPE	DESCRIPTION
BadgeBackgroundColor	`Color`	The badge color used in the tab.
BadgeBackgroundColorSelected	`Color`	The badge color used in the selected tab.
BadgeBorderColor	`Color`	The badge border color used in the tab.
BadgeBorderColorSelected	`Color`	The badge border color used in the selected tab.
BadgeText	bool	The badge text used in the tab.
BadgeTextColor	`Color`	The badge text color used in the tab.
BadgeTextColorSelected	`Color`	The badge text color used in the selected tab.
Content	`View`	The content of the tab. Anything can be used as content.
CurrentBadgeBackgroundColor	`Color`	Read-only property that reflects the currently used `BadgeBackgroundColor`
CurrentBadgeBorderColor	`Color`	Read-only property that reflects the currently used `BadgeBorderColor`
CurrentContent	`View`	Read-only property that reflects the currently used `Content`
CurrentFontAttributes	`FontAttributes`	Read-only property that reflects the currently used `FontAttributes`
CurrentFontFamily	string	Read-only property that reflects the currently used `FontFamily`
CurrentFontSize	double	Read-only property that reflects the currently used `FontSize`
CurrentIcon	`ImageSource`	Read-only property that reflects the currently used `Icon`
CurrentTextColor	`Color`	Read-only property that reflects the currently used `TextColor`
FontAttributes	`FontAttributes`	The font attributes used in the tab.
FontAttributesSelected	`FontAttributes`	The font attributes used in the selected tab.
FontFamily	string	The font family used in the tab.

PROPERTY	TYPE	DESCRIPTION
FontFamilySelected	string	The font family used in the selected tab.
FontSize	double	The font size used in the tab text.
FontSizeSelected	double	The font size used in the selected tab.
Icon	`ImageSource`	The icon of the tab.
IconSelected	`ImageSource`	The ImageSource used as icon in the selected tab.
IsSelected	bool	A bool that indicates if the tab is selected or not.
TabAnimation	ITabViewItemAnimation	The transition animation of a tab.
TabWidth	double	The width of a tab item
TapCommand	`ICommand`	Command that is executed when the user taps a tab.
TapCommandParameter	object	The tap command parameter.
Text	string	The tab text.
TextColor	`Color`	The text color of the tab.
TextColorSelected	`Color`	The text color of the selected tab.

Events

The following events are available on the `TabView` object:

EVENT	TYPE	DESCRIPTION
SelectionChanged	TabSelectionChangedEventHandler	Event that is raised when the selected tab changed.
Scrolled	TabViewScrolledEventHandler	Event that is raised when swiping between tabs.

The following events are available on the `TabViewItem` object:

EVENT	TYPE	DESCRIPTION
TabTapped	TabTappedEventHandler	Event that is raised when the user tap a tab.

Sample

TabView Sample Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- TabView source code
- TabViewItem source code

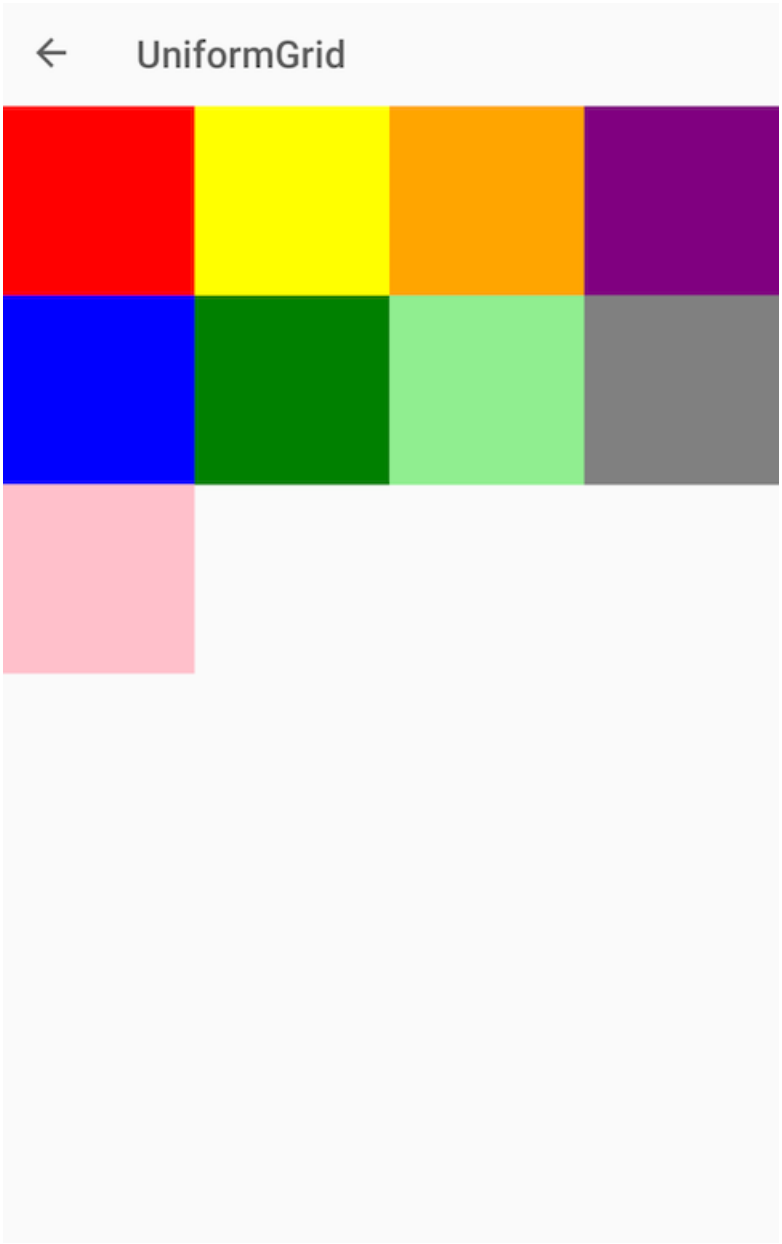
Related video

Xamarin Community Toolkit UniformGrid

3/5/2021 • 2 minutes to read • Edit Online

The `UniformGrid` is like the `Grid`, with the possibility of multiple rows and columns, but with one important difference: all rows and columns have the same size. That size is determined by the largest width and height of all the child elements. The child element with the largest width does not necessarily have to be the child element with the largest height.

Use the `UniformGrid` when you need the `Grid` behavior without the need to specify different sizes for the rows and columns.



Syntax

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:xct="http://xamarin.com/schemas/2020/toolkit"
             x:Class="MyLittleApp.MainPage">

     <StackLayout>

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

    </StackLayout>

</ContentPage>
```

Sample

UniformGrid sample page Source

You can see this in action in the Xamarin Community Toolkit Sample App.

API

- UniformGrid source code

Xamarin Community Toolkit: Troubleshooting

3/5/2021 • 2 minutes to read • Edit Online

If run into any issues or find a bug please report it on the Xamarin.CommunityToolkit GitHub repository.