

---

ComponentOne

# Xamarin Edition

**ComponentOne, a division of GrapeCity**

201 South Highland Avenue, Third Floor  
Pittsburgh, PA 15206 USA

**Website:** <http://www.componentone.com>

**Sales:** [sales@componentone.com](mailto:sales@componentone.com)

**Telephone:** 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

## Trademarks

The ComponentOne product name is a trademark and ComponentOne is a registered trademark of GrapeCity, Inc. All other trademarks used herein are the properties of their respective owners.

## Warranty

ComponentOne warrants that the media on which the software is delivered is free from defects in material and workmanship, assuming normal use, for a period of 90 days from the date of purchase. If a defect occurs during this time, you may return the defective media to ComponentOne, along with a dated proof of purchase, and ComponentOne will replace it at no charge. After 90 days, you can obtain a replacement for the defective media by sending it and a check for \$25 (to cover postage and handling) to ComponentOne.

Except for the express warranty of the original media on which the software is delivered is set forth here, ComponentOne makes no other warranties, express or implied. Every attempt has been made to ensure that the information contained in this manual is correct as of the time it was written. ComponentOne is not responsible for any errors or omissions. ComponentOne's liability is limited to the amount you paid for the product. ComponentOne is not liable for any special, consequential, or other damages for any reason.

## Copying and Distribution

While you are welcome to make backup copies of the software for your own use and protection, you are not permitted to make copies for the use of anyone else. We put a lot of time and effort into creating this product, and we appreciate your support in seeing that it is used by licensed users only.

## Table of Contents

Getting Started with Xamarin Edition	5
Breaking Changes for Xuni Users	5-6
NuGet Packages	6-7
Redistributable Files	7-8
System Requirements	8-9
Licensing	9
Licensing your app using GrapeCity License Manager Add-in	9-14
Licensing your app using website	14-15
Finding the Application Name	15-17
Creating a New Xamarin.Forms App	17-19
Creating a New Xamarin.Forms App	19-20
Adding NuGet Packages to your App	20-22
Adding Xamarin Components using C#	22-24
Adding Xamarin Components using XAML	24-25
About this Documentation	25
Technical Support	25-26
Controls	27
Calendar	27
Quick Start: Display a Calendar Control	27-30
Interaction Guide	30-31
Features	31
Customizing Appearance	31-33
Customizing Header	33-36
Customizing Day Content	36-39
Orientation	39-40
Selection	40-41
Customizing Selection	41-42
CollectionView	42
Features	42
Grouping	42-44
Incremental Loading	44-47
Sorting	47-48
FlexChart	48-49
Quick Start: Add Data to FlexChart	49-53

Chart Elements	53-54
Chart Types	54-58
Features	58
Axes	58-61
Customize Appearance	61-62
Data Binding	62-64
Data Labels	64-67
Header and Footer	67-68
Hit Test	68-70
Legend	70-71
Mixed Charts	71-72
Multiple Y Axes	72-75
Selection	75-76
Themes	76-78
Zones	78-82
FlexGrid	82-83
Quick Start: Add Data to FlexGrid	83-87
Features	87
Custom Cells	87-91
Custom Cell Editor	91
Customize Appearance	91-92
Clipboard and Keyboard Support	92-93
Data Mapping	93-95
Defining Columns	95-96
Editing	96
Inline Editing	96-97
Filtering	97
Search Box Filtering	97-99
Formatting Columns	99-101
Frozen Rows and Columns	101-102
Grouping	102-104
Merging Cells	104-106
Resizing Columns	106
Row Details	106-115
Sorting	115-118
Selecting Cells	118

Star and Auto Sizing	118-119
Wordwrap	119-120
FlexPie	120-121
Quick Start: Add data to FlexPie	121-124
Features	124
Customize Appearance	124-125
Data Binding	125-126
Data Labels	126-129
Donut Pie Chart	129-130
Exploded Pie Chart	130
Header and Footer	130-131
Legend	131-132
Themes	132-134
Gauge	134-135
Gauge Types	135-136
Quick Start: Add and Configure	136
BulletGraph Quick Start	136-139
LinearGauge Quick Start	139-141
RadialGauge Quick Start	141-145
Features	145
Animation	145
Customize Appearance	145-146
Data Binding	146-147
Direction	147-148
Range	148-149
User Scenario	149
Poll Results	149-153
Input	153
AutoComplete	153-154
Quick Start: Populating AutoComplete with Data	154-156
Features	157
Data Binding	157
Delay	157
Highlight Matches	157-158
CheckBox	158
ComboBox	158-159

Quick Start: Display a ComboBox Control	159-162
Features	162
Custom Appearance	162-163
Data Binding	163
Editing	163-164
DropDown	164
Creating a Custom Date Picker using DropDown	164-166
MaskedTextField	166
Quick Start: Display a MaskedTextField Control	166-168
Mask Symbols	168

## Getting Started with Xamarin Edition

**Xamarin Edition** is a collection of cross-platform UI controls developed by GrapeCity for Xamarin.iOS, Xamarin.Android, and Xamarin.Forms. Xamarin Edition has been optimized for Xamarin development, and provides native experiences in Android, iOS, and UWP. This beta release provides an opportunity for users to try out the enhanced architecture and provide feedback.

For existing Xuni users, the new architecture brings many new features:

- **UWP Support**  
The new architecture adds the ability to use our controls in Xamarin.Forms apps targeting UWP. This grows the number of potential platforms that you can target, and allows you to utilize Microsoft's newest technologies together.
- **Better Performance**  
The new controls should generally perform better than the old controls (sometimes doubling performance). By specifically focusing on the Xamarin architecture, the controls cut out some intermediary logic and are optimized for the platform. Since they're entirely in C#, so you can also expect a more consistent experience.
- **Designer Support**  
The new controls should also support Xamarin's designers for iOS and Android applications. This makes it much easier to construct your Android XML or iOS Storyboards using these controls.
- **New Control Features**  
The controls have been rethought for the new architecture with the combined experience of **Xuni**, **Wijmo**, as well as **ComponentOne** controls. Some controls have a number additional features (such as FlexGrid). The controls will continue to be in active development throughout the course of the beta.

## Breaking Changes for Xuni Users

### New Package Names

The packages have changed their prefix if you're coming from Xuni. For instance,

**Xuni.Android.Calendar** now corresponds to **C1.Android.Calendar**

**Xuni.iOS.Calendar** now corresponds to **C1.iOS.Calendar**

**Xuni.Forms.Calendar** now corresponds to **C1.Xamarin.Forms.Calendar**

We have also moved to a more consistent naming scheme for our controls based on the following pattern:

**C1.[Platform].[ControlName]**

For example, FlexGrid is available in **C1.Xamarin.Forms.Grid**

Additionally, FlexChart, FlexPie, and ChartCore have all been consolidated into one single package instead of three different packages. To use FlexChart or FlexPie, you now need to add a single package developed for the platform of your choice:

- C1.Android.Chart
- C1.iOS.Chart
- C1.Xamarin.Forms.Chart

### Namespace Changes

We've made some changes to the namespace of the current controls, which are in line with the changes in package names. For example, **Xuni.Forms.Calendar** now corresponds to **C1.Xamarin.Forms.Calendar**.

## Minor API Changes

There are some minor changes in API between ComponentOne Xamarin Edition and Xuni. These should mostly amount to additions, slight change in syntax, and use of prefix 'C1' instead of 'Xuni' in class and object names. For FlexChart, however, the control is very actively growing in terms of API, so missing features are intended to be added in the future.

## NuGet Packages

The following NuGet packages are available for download:

Name	Description
<b>C1.Android.Calendar</b>	Installing this NuGet package adds all the references that enable you to use the Calendar control in your Xamarin.Android application.
<b>C1.Android.Core</b>	This is the dependency package for the control NuGet packages and is automatically installed when any dependent package is installed.
<b>C1.Android.Chart</b>	Installing this NuGet package adds all the references that enable you to use the FlexChart and FlexPie controls in your Xamarin.Android application.
<b>C1.Android.Grid</b>	Installing this NuGet package adds all the references that enable you to use the FlexGrid control in your Xamarin.Android application.
<b>C1.Android.Gauge</b>	Installing this NuGet package adds all the references that enable you to use the Gauge control in your Xamarin.Android application.
<b>C1.Android.Input</b>	Installing this NuGet package adds all the references that enable you to use the Input controls in your Xamarin.Android application.
<b>C1.CollectionView</b>	This is the dependency package for the control NuGet packages and is automatically installed when any dependent package is installed.
<b>C1.Xamarin.Forms.Calendar</b>	Installing this NuGet package adds all the references that enable you to use the Calendar control in your project.
<b>C1.Xamarin.Forms.Core</b>	This is the dependency package for the control NuGet packages and is automatically installed when any dependent package is installed.
<b>C1.Xamarin.Forms.Chart</b>	Installing this NuGet package adds all the references that enable you to use the FlexChart and FlexPie controls in your project.
<b>C1.Xamarin.Forms.Grid</b>	Installing this NuGet package adds all the references that enable you to use the FlexGrid control in your project.
<b>C1.Xamarin.Forms.Gauge</b>	Installing this NuGet package adds all the references that enable you to use the Gauge control in your project.
<b>C1.Xamarin.Forms.Input</b>	Installing this NuGet package adds all the references that enable you to use the Input controls in your project.
<b>C1.iOS.Calendar</b>	Installing this NuGet package adds all the references that enable you to use the Calendar control in your Xamarin.iOS application.
<b>C1.iOS.Core</b>	This is the dependency package for the control NuGet packages and is automatically installed when any dependent package is installed.
<b>C1.iOS.Chart</b>	Installing this NuGet package adds all the references that enable you to use the



	FlexChart and FlexPie controls in your Xamarin.iOS application.
<b>C1.iOS.Grid</b>	Installing this NuGet package adds all the references that enable you to use the FlexGrid control as well as CollectionView interface in your Xamarin.iOS application.
<b>C1.iOS.Gauge</b>	Installing this NuGet package adds all the references that enable you to use the Gauge control in your Xamarin.iOS application.
<b>C1.iOS.Input</b>	Installing this NuGet package adds all the references that enable you to use the Input controls in your Xamarin.iOS application.

## Redistributable Files

**Xamarin Edition**, developed and published by GrapeCity, inc., can be used to develop applications in conjunction with Microsoft Visual Studio, Xamarin Studio or any other programming environment that enables the user to use and integrate controls. You may also distribute, free of royalties, the following redistributable files with any such application you develop to the extent that they are used separately on a single CPU on the client/workstation side of the network.

<b>Calendar</b>
<ul style="list-style-type: none"> <li>• C1.Android.Calendar.dll</li> <li>• C1.Xamarin.Forms.Calendar.dll</li> <li>• C1.Xamarin.Forms.Calendar.Platform.Android.dll</li> <li>• C1.Xamarin.Forms.Calendar.Platform.iOS.dll</li> <li>• C1.Xamarin.Forms.Calendar.Platform.UWP.dll</li> <li>• C1.iOS.Calendar.dll</li> <li>• C1.UWP.Calendar.dll</li> </ul>
<b>CollectionView</b>
<ul style="list-style-type: none"> <li>• C1.CollectionView.dll</li> </ul>
<b>Core</b>
<ul style="list-style-type: none"> <li>• C1.Android.Core.dll</li> <li>• C1.Xamarin.Forms.Core.dll</li> <li>• C1.Xamarin.Forms.Core.Platform.Android.dll</li> <li>• C1.Xamarin.Forms.Core.Platform.iOS.dll</li> <li>• C1.Xamarin.Forms.Core.Platform.UWP.dll</li> <li>• C1.iOS.Core.dll</li> <li>• C1.UWP.Core.dll</li> </ul>
<b>FlexChart</b>
<ul style="list-style-type: none"> <li>• C1.Android.Chart.dll</li> <li>• C1.Xamarin.Forms.Chart.dll</li> <li>• C1.Xamarin.Forms.Chart.Platform.Android.dll</li> <li>• C1.Xamarin.Forms.Chart.Platform.iOS.dll</li> <li>• C1.Xamarin.Forms.Chart.Platform.UWP.dll</li> <li>• C1.iOS.FlexChart.dll</li> <li>• C1.UWP.FlexChart.dll</li> </ul>

## FlexGrid

- C1.Android.Grid.dll
- C1.Xamarin.Forms.Grid.dll
- C1.Xamarin.Forms.Grid.Platform.Android.dll
- C1.Xamarin.Forms.Grid.Platform.iOS.dll
- C1.Xamarin.Forms.Grid.Platform.UWP.dll
- C1.iOS.Grid.dll
- C1.UWP.Grid.dll

## Gauge

- C1.Android.Gauge.dll
- C1.Xamarin.Forms.Gauge.dll
- C1.Xamarin.Forms.Gauge.Platform.Android.dll
- C1.Xamarin.Forms.Gauge.Platform.iOS.dll
- C1.Xamarin.Forms.Gauge.Platform.UWP.dll
- C1.iOS.Gauge.dll
- C1.UWP.Gauge.dll

## Input

- C1.Android.Input.dll
- C1.Xamarin.Forms.Input.dll
- C1.Xamarin.Forms.Input.Platform.Android.dll
- C1.Xamarin.Forms.Input.Platform.iOS.dll
- C1.Xamarin.Forms.Input.Platform.UWP.dll
- C1.iOS.Input.dll
- C1.UWP.Input.dll

## System Requirements

ComponentOne Xamarin can be used in applications written for the following mobile operating systems:

- Android 4.2 and above (API 17)
- iOS 10 and above recommended
- Windows 10 and Windows 10 Mobile

### Requirements

- Xamarin Platform 2.3.3.193 and above
- Visual Studio 2015 Update 3

### Windows System Requirements

- Windows 8.1 and above

### Mac System Requirements

- Xamarin Studio or Visual Studio for Mac
- MacOS 10.12
- Xcode 8 and above

## Additional requirements for UWP development

- Windows 10 operating system
- Visual Studio 2015 and above
- Windows 10 SDK
- Microsoft.NETCore.UniversalWindowsPlatform 5.2.2 or newer NuGet Package

## Licensing

**ComponentOne Xamarin Edition** contains runtime licensing, which means the library requires a unique key to be validated at runtime. The process is quick, requires minimal memory, and does not require network connection. Each application that uses ComponentOne Xamarin Edition requires a unique license key. This topic gives you in-depth instructions on how to license your app. For more information on GrapeCity licensing and subscription model, visit <https://www.componentone.com/Pages/Licensing/>.

To know the licensing process in details, see the following links

- [Licensing your app using GrapeCity License Manager Add-in](#)
- [Licensing you app using website](#)

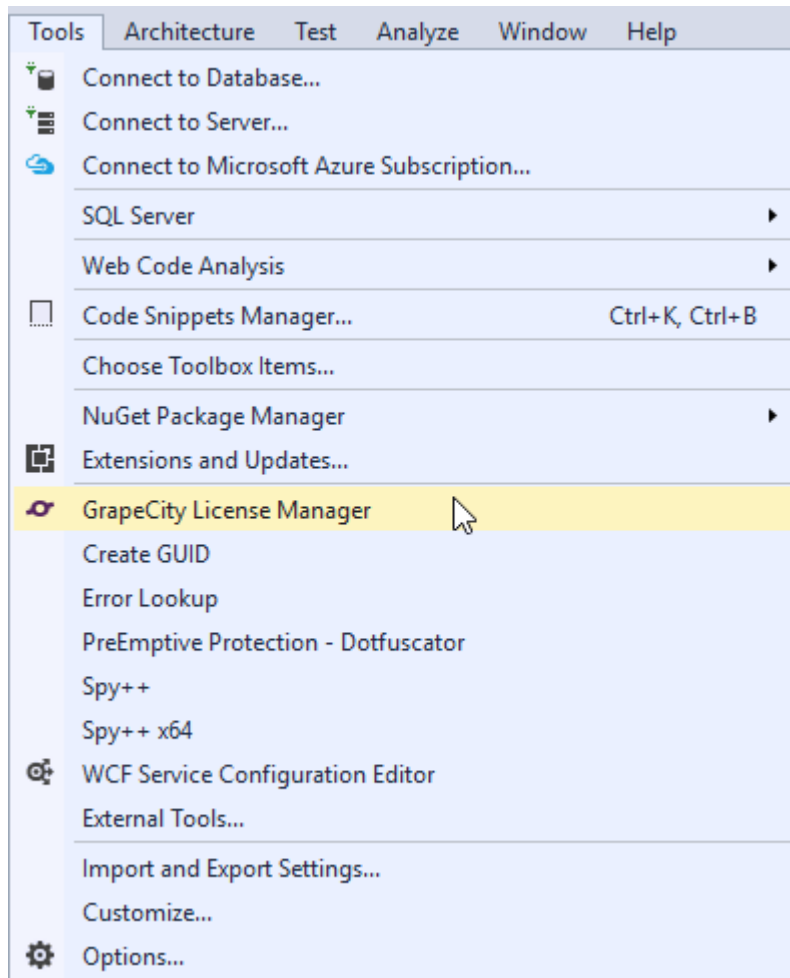
## Licensing your app using GrapeCity License Manager Add-in

If you are using ComponentOne Xamarin Edition 2017v2 and newer versions with Visual Studio, you can use the GrapeCity License Manager Add-in to license your apps. If you are using a version with Xamarin Studio or Visual Studio for Mac, follow the instructions given in [Licensing your app using website](#).

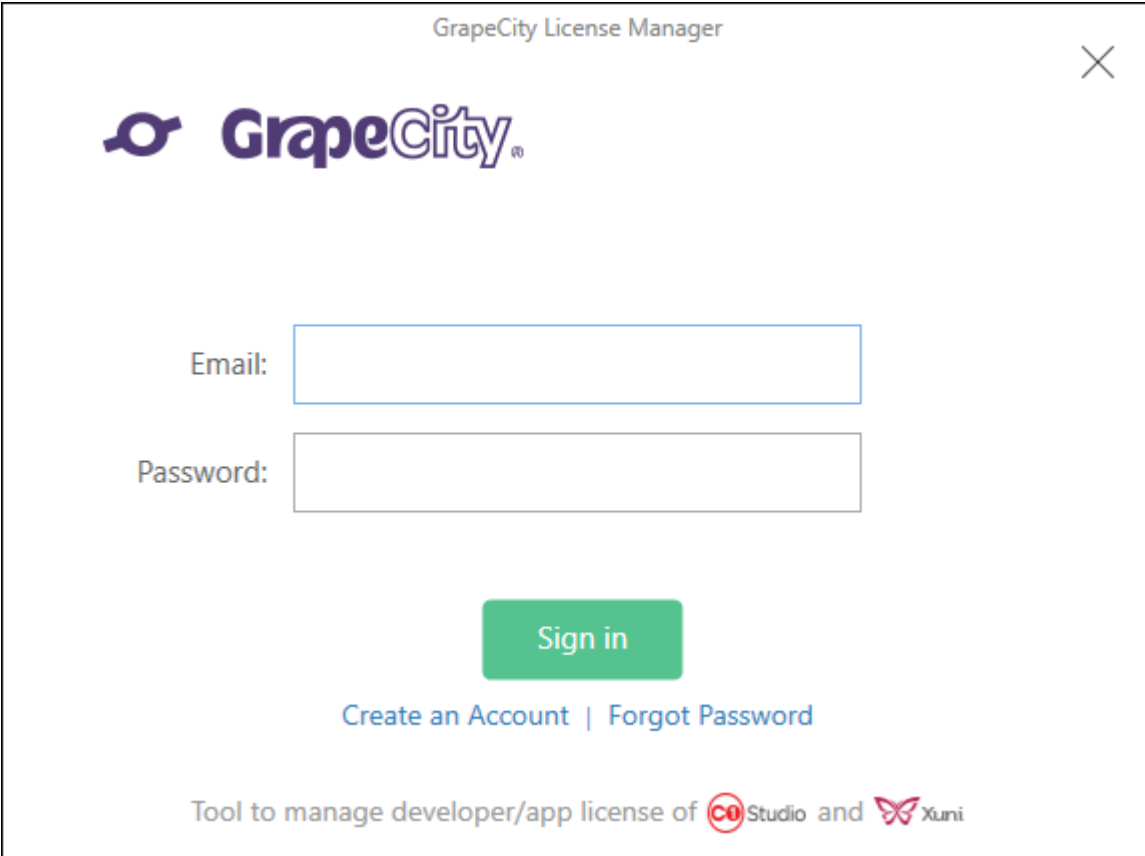
### GrapeCity License Manager Add-in for Visual Studio

The GrapeCity License Manager add-in generates XML keys to license your apps directly in Visual Studio. To license a Xamarin.Forms app, complete the steps as follows:

1. From the **Tools** menu in Visual Studio, select **GrapeCity License Manager** option.




2. Sign-in into the License Manager using your email address and password. If you have not created a **GrapeCity Account** in the past, you can create an account at this step. If you are already signed in, skip the screen.



The image shows a web application window titled "GrapeCity License Manager". At the top left is the GrapeCity logo, and at the top right is a close button (X). Below the logo, there are two input fields: "Email:" and "Password:". Below these fields is a green "Sign in" button. Under the button are two links: "Create an Account" and "Forgot Password". At the bottom, there is a line of text: "Tool to manage developer/app license of" followed by the CO Studio logo and the Xuni logo.

GrapeCity License Manager





Email:

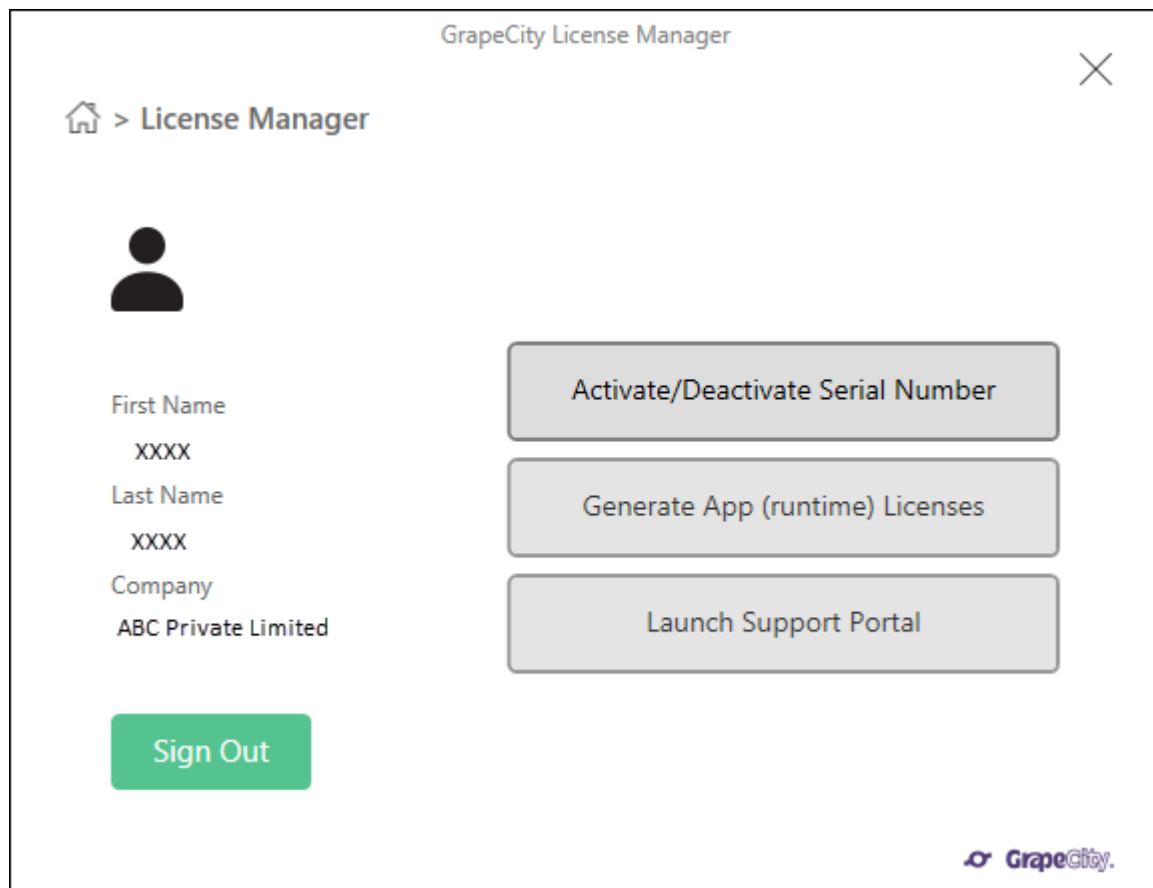
Password:

[Sign in](#)

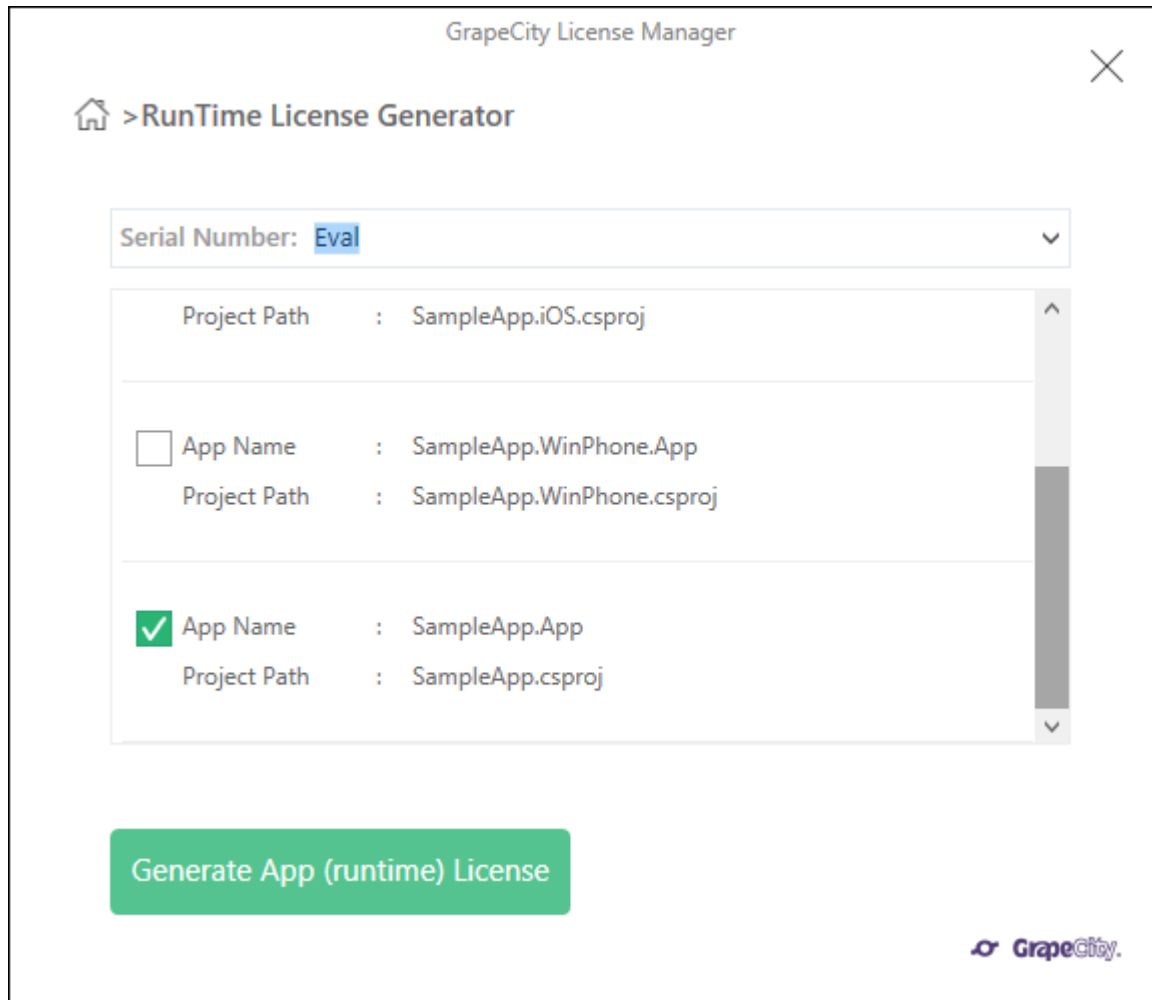
[Create an Account](#) | [Forgot Password](#)

Tool to manage developer/app license of  Studio and  Xuni

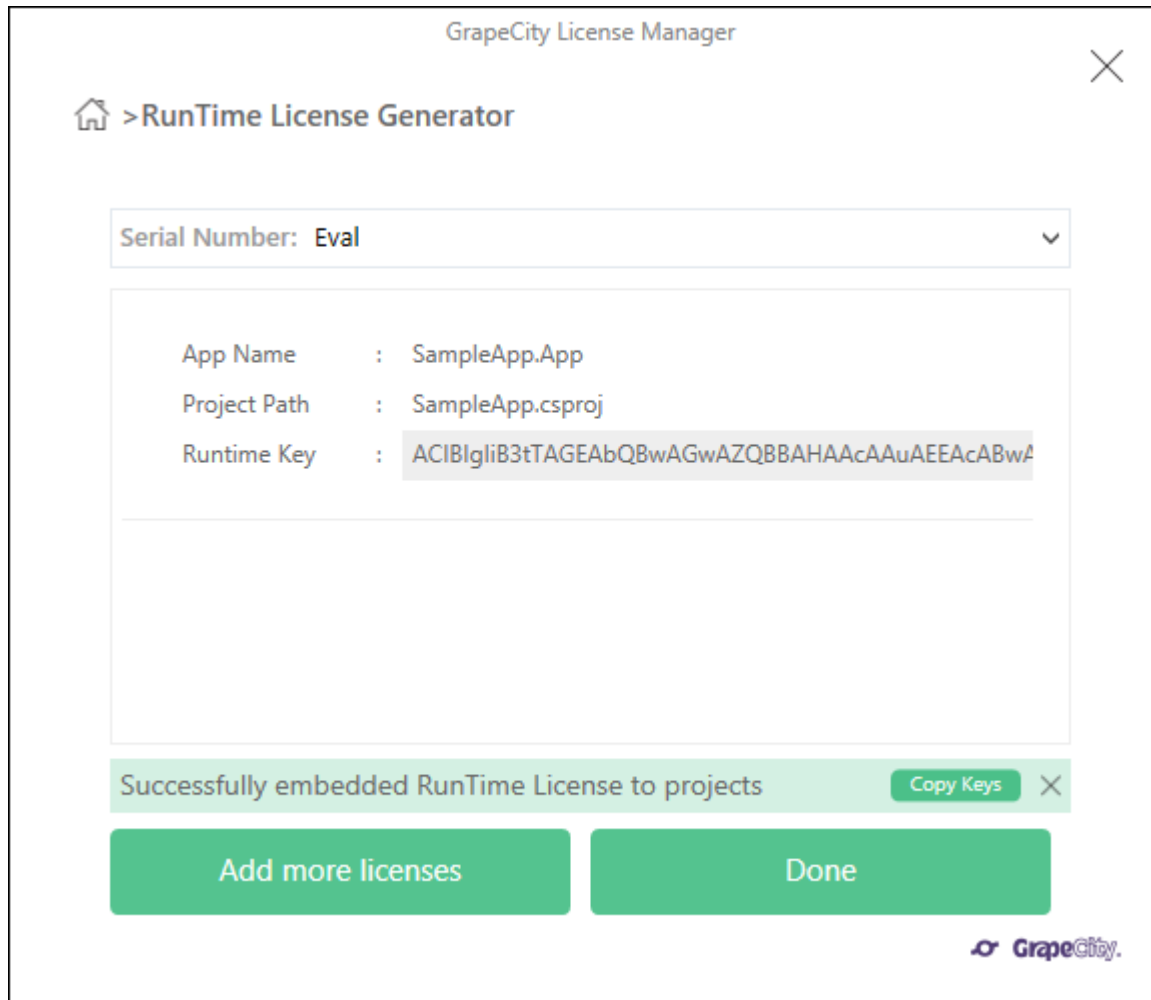
3. From the main License Manager Screen, you can activate or deactivate a serial number, generate a license key, or launch the support portal. To activate a full license serial key, click **Activate/Deactivate Serial Number**. To generate an app license using an already activated serial key or a trial key, click **Generate App (runtime) Licenses**.



4. Select your full license from the drop-down menu at the top. To generate a trial key, select **Eval**.
5. Click on the check box next to the PCL or shared project to be licensed.



6. Click **Generate App (runtime) License**.
7. Click **Done** to add the **GCDTLicenses.xml** file containing the license key to your PCL or shared project.




You can now build and run your licensed app.

## Licensing your app using website

ComponentOne Xamarin Edition users can license an app via the ComponentOne website. If you are using ComponentOne Xamarin Edition with Visual Studio on PC, you have the option to use the GrapeCity License Manager Add-in. For more information, see the topic [Licensing your app using GrapeCity License Manager Add-in](#).

### How to license your app using the website

1. Open a pre-existing mobile application or create a new mobile application.
2. Add the required Xamarin Edition NuGet packages to your application through the **NuGet Package Manager**.
3. Visit <https://www.componentone.com/Members/?ReturnUrl=%2fMyAccount%2fMyXuni.aspx>.

 **Note:** You must create a GrapeCity account and login to access this web page.

4. If you are generating a full license, select your serial number from the drop-down menu at the top of the page. If you are generating a trial license, leave it selected as **Evaluation**.
5. Select C# for the language.
6. In the **App Name** textbox, enter the name of your application. This name should match the Default Namespace of your PCL (shared project in your Xamarin.Forms application). See [Finding the Application Name](#) to know how to find the name of your application.
7. Click the **Generate** button. A runtime license will be generated in the form of a string contained within a class.



8. Copy the license and complete the following steps to add it to your application.
  1. Open your application in Visual Studio or Xamarin Studio.
  2. In the **Solution Explorer**, right click the project YourAppName (Portable or Shared).
  3. Select **Add | New**. The **Add New Item** dialog appears.
  4. Under installed templates, select **C# | Class**.
  5. Set the name of the class as `License.cs` and click **OK**.
  6. In the class `License.cs`, create a new string to store the runtime license, inside the constructor as shown below.

```
C#  
  
public static class License  
{  
    public const string Key = "Your Key";  
}
```

7. From the **Solution Explorer**, open `App.cs` and set the runtime license, inside the constructor `App()` method as shown below.

```
C#  
  
Cl.Xamarin.Forms.Core.LicenseManager.Key = License.Key;
```

If you are generating a trial license, your application is now ready to use for trial purposes. You can repeat this process for any number of applications. You must generate a new trial license for each app because they are unique to the application name.



The trial period is limited to 30 days, which begins when you generate your first runtime license. The controls will stop working after your 30 day trial has ended. You can extend your license by contacting our sales team.

## Finding the Application Name

ComponentOne Xamarin Edition licenses are unique to each application. Before you can generate a runtime license, you need to know the name of the application where the license will be used.

## Visual Studio

1. Open a pre-existing Mobile App.
2. In the **Solution Explorer**, right click the project YourAppName (Portable or Shared) and select **Properties**.
3. Open the **Library** tab.
4. The application name is the same as the **Default Namespace** displayed.

Library

Build

Build Events

Resources

Reference Paths

Signing

Code Analysis

Configuration: N/A Platform: N/A

General

Assembly name: App1

Default namespace: App1

Output type: Class Library


Assembly Information...

Targeting

Targets:

- .NET Framework 4.5
- Windows 8
- Windows Phone Silverlight 8
- Xamarin.Android
- Xamarin.iOS
- Xamarin.iOS (Classic)

Change...

 You need to generate a new runtime license in case you rename the assembly later.

## Xamarin Studio

1. Open a pre-existing Mobile App.
2. In the **Solution Explorer**, right click the project YourAppName (Portable or Shared) and select **Options**.
3. The application name is displayed on the **Main Settings** tab.

The screenshot shows the 'Main Settings' dialog box in Xamarin Studio. On the left is a sidebar with a tree view containing categories: General (with 'Main Settings' selected), Build (with sub-items: General, Custom Commands, Configurations, Compiler, Assembly Signing, Output), Run (with sub-items: General, Custom Commands), Source Code (with sub-items: .NET Naming Policies, Code Formatting, Standard Header, Name Conventions), and Version Control (with sub-item: Commit Message Style). The main area is titled 'Main Settings' and contains two sections. The 'Project Information' section has fields for 'Name' (App1), 'Version' (0.1) with a checkbox 'Get version from parent solution' checked, and a 'Description' text area. Below this is a 'Default Namespace' field (App1). The 'Location of Files' section has a 'Root directory' field containing 'C:\Users\UserName\Desktop\App1\App1\App1' and a 'Browse...' button. At the bottom right are 'Cancel' and 'OK' buttons.

## Creating a New Xamarin.Forms App

This topic demonstrates how to create a new cross platform app in Visual Studio or Xamarin Studio. See the [System Requirements](#) before proceeding. To download and install **Xamarin Studio**, visit <http://xamarin.com/download>.

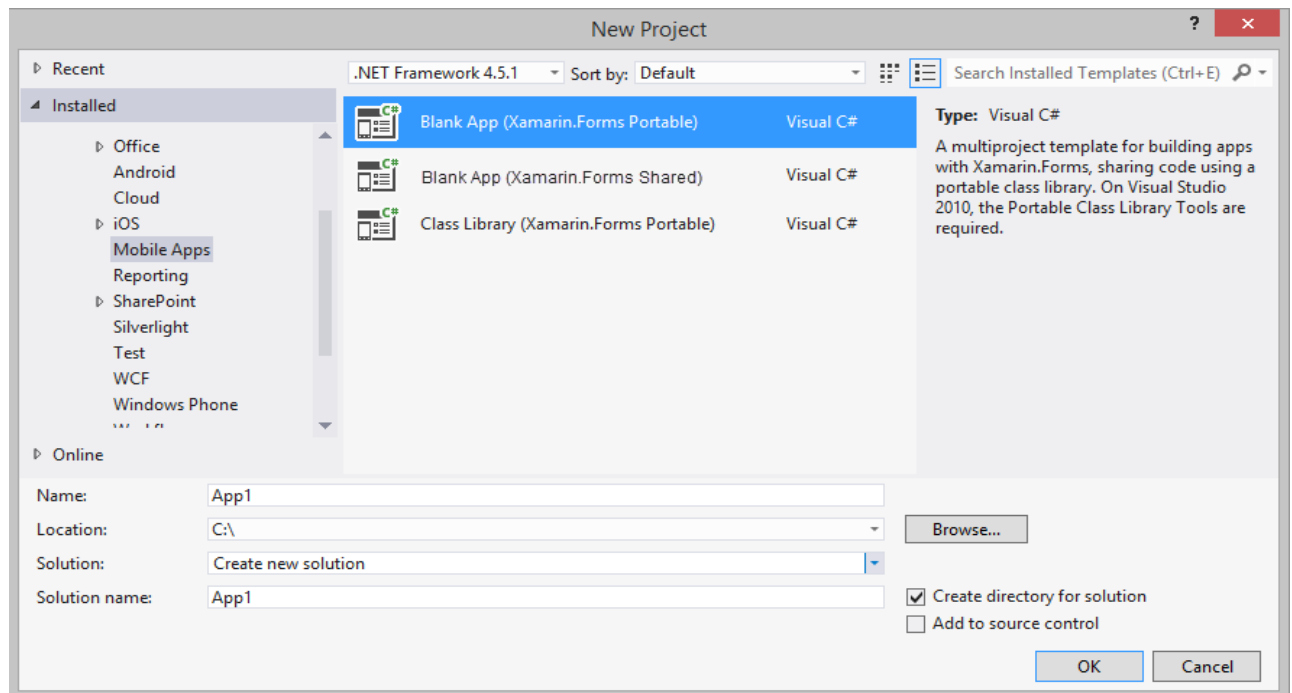
To know more about Sharing Code Options in Xamarin, visit:

[http://developer.xamarin.com/guides/cross-platform/application\\_fundamentals/building\\_cross\\_platform\\_applications/sharing\\_code\\_options/](http://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/sharing_code_options/)

Complete the following steps to create a new Xamarin.Forms Portable or Shared App:

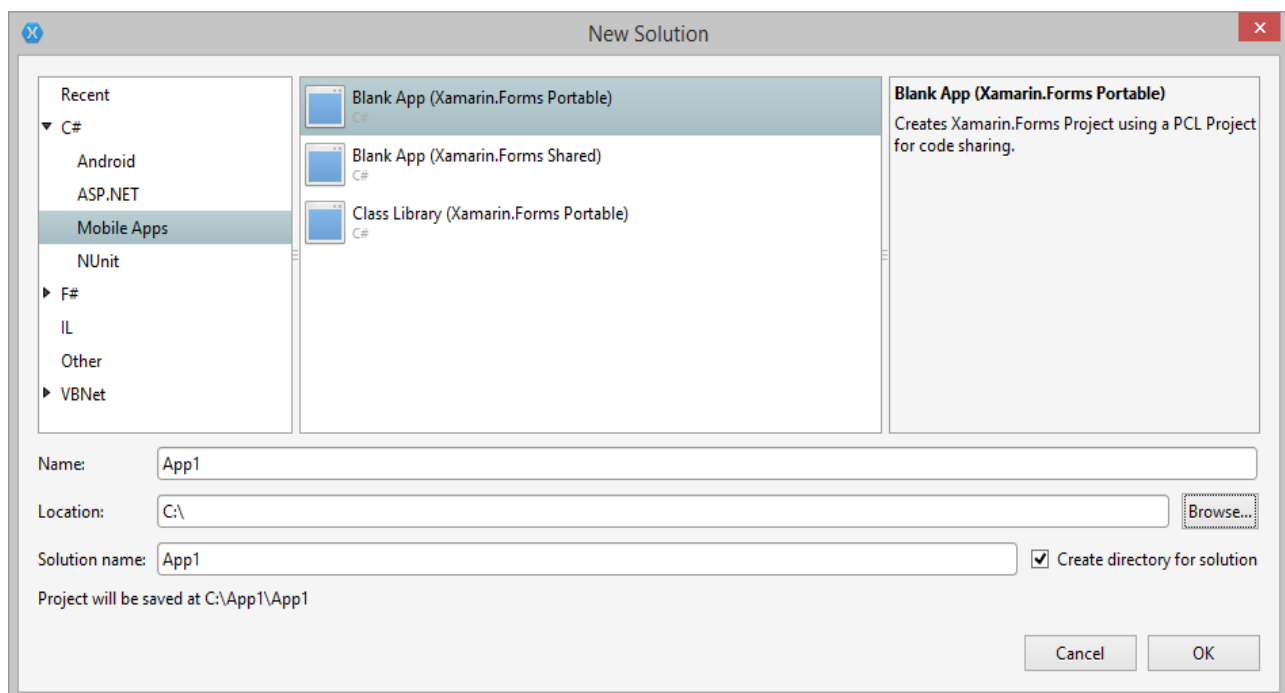
### Visual Studio


1. Select **File | New | Project**.
2. Under installed templates, select **Visual C# | Mobile Apps**.
3. In the right pane, select **Blank App (Xamarin.Forms Portable or Xamarin.Forms Shared)**.
4. Add a name for your app and select a location to save it.
5. Click **OK**.



## Xamarin Studio (Windows and MAC)

1. Select **File | New | Solution**.
2. Select **C# | Mobile Apps**.
3. In the right pane, select **Blank App (Xamarin.Forms Portable)** or **Xamarin.Forms Shared**
4. Add a name for your app and select a location to save it.
5. Click **OK**.



 A portable or shared solution comprises the following projects:

- **Portable or Shared:** A cross-platform application library containing the shared UI and code.
- **Android:** Android mobile application available on all development environments.

- **UWP:** Universal Windows application available only in projects created on Visual Studio.
- **iOS:**
  - Available on OS X.
  - A Mac PC is required to use Visual Studio as the development environment. See [Introduction to Xamarin.iOS for Visual Studio](#) to know how to setup your environment.

## Creating a New Xamarin.Forms App

This topic demonstrates how to create a new cross platform app in Visual Studio or Xamarin Studio. See the [System Requirements](#) before proceeding. To download and install **Xamarin Studio**, visit <http://xamarin.com/download>.

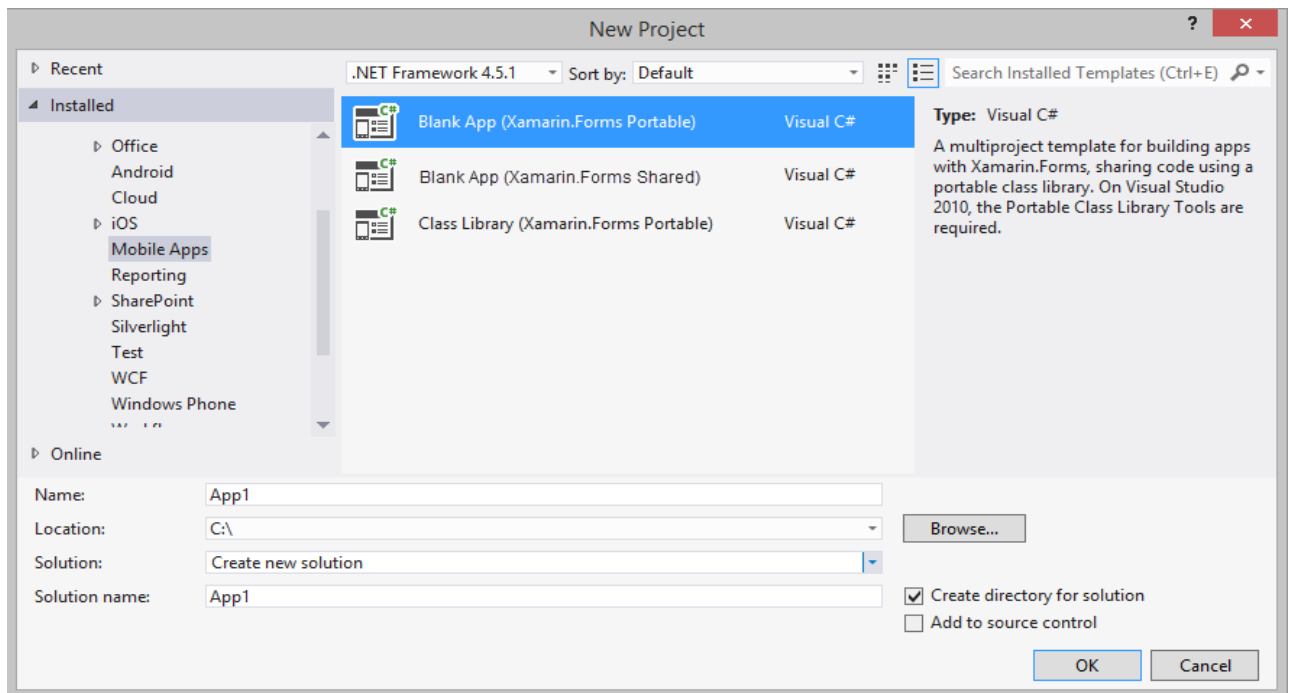
To know more about Sharing Code Options in Xamarin, visit:

[http://developer.xamarin.com/guides/cross-platform/application\\_fundamentals/building\\_cross\\_platform\\_applications/sharing\\_code\\_options/](http://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/sharing_code_options/)

Complete the following steps to create a new Xamarin.Forms Portable or Shared App:

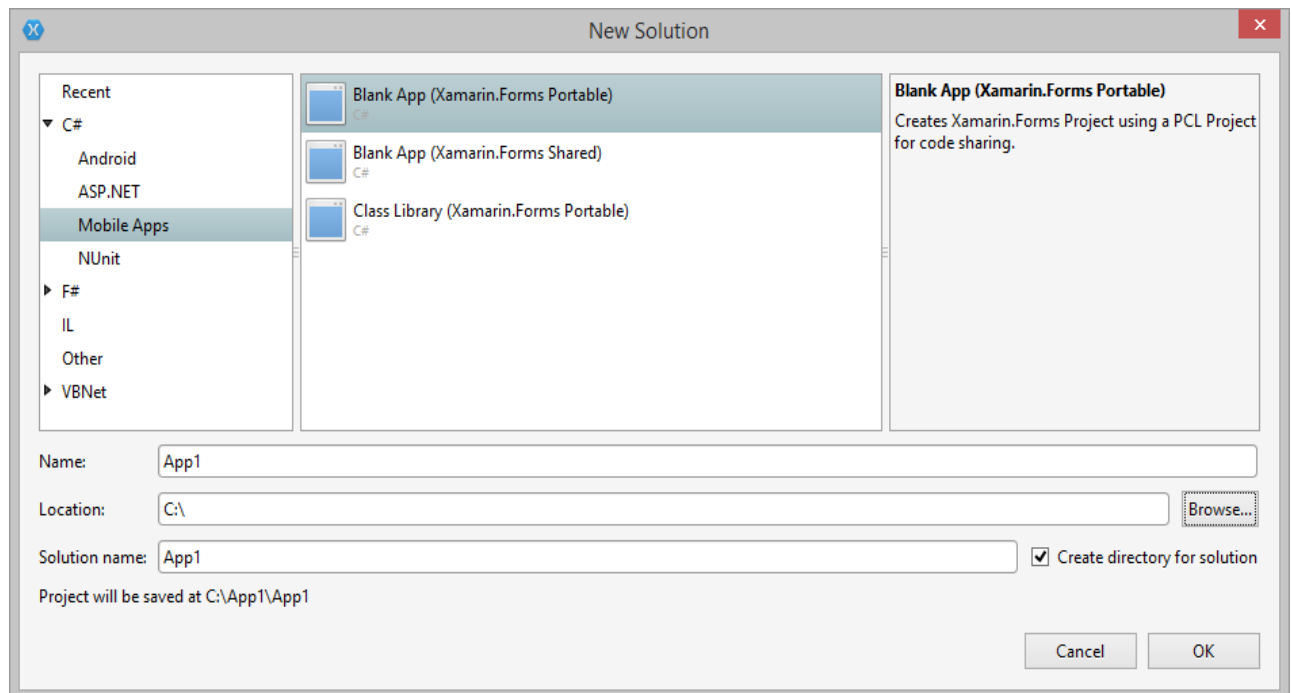
- **Visual Studio (Windows)**


1. Select **File | New | Project**.
2. Under installed templates, select **Visual C# | Mobile Apps**.
3. In the right pane, select **Blank App (Xamarin.Forms Portable or Xamarin.Forms Shared)**.
4. Add a name for your app and select a location to save it.
5. Click **OK**.



- **Xamarin Studio (Windows and MAC)**

1. Select **File | New | Solution**.
2. Select **C# | Mobile Apps**.
3. In the right pane, select **Blank App (Xamarin.Forms Portable or Xamarin.Forms Shared)**.
4. Add a name for your app and select a location to save it.
5. Click **OK**.



 A portable or shared solution comprises the following projects:

- **Portable or Shared:** A cross-platform application library containing the shared UI and code.
- **Android:** Android mobile application available on all development environments.
- **UWP:** Windows mobile application available only in projects created on Visual Studio.
- **iOS:**
  - Available on OS X.
  - A Mac PC is required to use Visual Studio as the development environment. See [Introduction to Xamarin.iOS for Visual Studio](#) to know how to setup your environment.

## Adding NuGet Packages to your App

### To Install NuGet

1. Go to <http://nuget.org/> and click **Install NuGet**.
2. Run the **NuGet.vsix** installer.
3. In the Visual Studio Extension Installer window click **Install**.
4. Once the installation is complete, click **Close**.

### To add Xamarin References to your App

In order to use Xamarin controls on the three platforms: Android, iOS and UWP, Xamarin references have to be added to all three projects for each of these platforms. Complete the following steps to add Xamarin references to your project.

## Visual Studio

1. Open a pre-existing Mobile App or create a new Mobile App (see [Creating a New Xamarin.Forms App](#)).
2. From the **Project** menu, select **Manage NuGet Packages**. The **Manage NuGet Packages** dialog box appears.
3. Click **Online** and then click **GrapeCity**.
4. Click **Install** next to **C1.Xamarin.Forms.ControlName** (for example C1.Xamarin.Forms.FlexChart). This adds the references for the Xamarin control.

5. Click **I Accept** to accept the license and then click **Close** in the Manage NuGet Packages dialog box.

## Xamarin Studio

1. Open a pre-existing Mobile App or create a new Mobile App (see [Creating a New Xamarin.Forms App](#)).
2. In the **Solution Explorer**, right click the project and select **Add | Add Packages**. The **Add Packages** dialog appears.
3. From the drop down menu in the top left corner, select **GrapeCity**. The available Xamarin packages are displayed.
4. Select the package **C1.Xamarin.Forms.ControlName** and click the **Add Package** button. This adds the references for the Xamarin control.

### To manually create a Xamarin Feed Source

Complete the following steps to manually add Xamarin NuGet feed URL to your NuGet settings in Visual Studio or Xamarin Studio and install Xamarin.

## Visual Studio

1. From the **Tools** menu, select **NuGet Package Manager | Package Manager Settings**. The Options dialog box appears.
2. In the left pane, select **Package Sources**.
3. Click the **Add** button in top right corner. A new source is added under **Available package sources**.
4. Set the **Name** of the new package source. Set the **Source** as <http://nuget.grapecity.com/nuget/>.
5. Click **OK**. The feed has now been added as another NuGet feed source.

### To install Xamarin using the new feed:

1. Open a pre-existing Mobile App or create a new Mobile App (see [Creating a New Xamarin.Forms App](#)).
2. Select **Project | Manage NuGet Packages**. The **Manage NuGet Packages** dialog box appears.
3. Click **Online** and then click **Xamarin**. The available packages are displayed in the right pane.
4. Click **Install** next to **C1.Xamarin.Forms.ControlName** (for example C1.Xamarin.Forms.Chart). This updates the references for the Xamarin control.
5. Click **I Accept** to accept the ComponentOne license for Xamarin and then click **Close** in the Manage NuGet Packages dialog box.

## Xamarin Studio

1. From the **Projects** menu, select **Add Packages**. The **Add Packages** dialog appears.
2. From the drop down menu on the top left corner, select **Configure Sources**. The **Preferences** dialog appears.
3. In the left pane, expand **Packages** and select **Sources**.
4. Click the **Add** button. The **Add Package Source** dialog appears.
5. Set the **Name** of the new package source. Set the **URL** as <http://nuget.grapecity.com/nuget/>.
6. Click the **Add Source** button. The Xamarin feed has now been added as another NuGet feed source.
7. Click **OK** to close the Preferences dialog.

### To install Xamarin using the new feed:

1. Open a pre-existing Mobile App or create a new Mobile App (see [Creating a New Xamarin.Forms App](#)).
2. In the **Solution Explorer**, right click the project and select **Add | Add Packages**. The **Add Packages** dialog appears.
3. From the drop down menu on the top left corner, select **Xamarin**. The available Xamarin packages are displayed.
4. Select the package **C1.[Platform].[ControlName]** and click the **Add Package** button. This adds the references

for the Xamarin control.

## Adding Xamarin Components using C#

This topic demonstrates how to add a Xamarin control to your app using C#. This is done in three steps:

- **Step 1: Add a new Class**
- **Step 2: Add the Control**
- **Step 3: Run the Program**

### Step 1: Add a new Class

1. In the **Solution Explorer**, right click the project YourAppName (Portable or Shared).
2. Select **Add | New**. The **Add New Item** dialog appears.
3. Under installed templates, select **C# | Class**.
4. Add a name for the class (for example Class1.cs) and click **OK**. A new class is added to your project.

### Step 2: Add the Control

1. In the **Solution Explorer**, double click Class1.cs to open it.
2. Include the necessary namespaces. For example, the following code includes the namespace for Gauge.

```
C#  
  
using Cl.Xamarin.Forms.Gauge;  
using Xamarin.Forms;
```

3. Declare a new method (for example ReturnMyControl()) with the control you want to add set as its return type.
4. In the method definition create an instance of the control and set its properties.

The following example shows how to create an instance of the LinearGauge control and initialize it in the ReturnMyControl() method definition.

```
C#  
  
public static ClLinearGauge ReturnMyControl()  
{  
    // Instantiate LinearGauge and set its properties  
    ClLinearGauge gauge = new ClLinearGauge();  
    gauge.HeightRequest = 50;  
    gauge.WidthRequest = 50;  
    gauge.Value = 35;  
    gauge.Thickness = 0.1;  
    gauge.Min = 0;  
    gauge.Max = 100;  
    gauge.Direction = LinearGaugeDirection.Right;  
  
    //Create Ranges  
    GaugeRange low = new GaugeRange();  
    GaugeRange med = new GaugeRange();  
    GaugeRange high = new GaugeRange();  
  
    //Customize Ranges  
    low.Color = Color.Red;  
    low.Min = 0;  
    low.Max = 40;
```



```
med.Color = Color.Yellow;
med.Min = 40;
med.Max = 80;
high.Color = Color.Green;
high.Min = 80;
high.Max = 100;

//Add Ranges to Gauge
gauge.Ranges.Add(low);
gauge.Ranges.Add(med);
gauge.Ranges.Add(high);

return gauge;
}
```

## Back to Top

### Step 3: Run the Program

1. In the **Solution Explorer**, double click App.cs to open it.
2. In the class constructor App(), set a new ContentPage as the MainPage and assign the control to the ContentPage's Content by invoking the method ReturnMyControl() defined in the previous procedure, **Step 2: Add a Control**.

The following code shows the class constructor App() after completing steps above.

```
C#
public App()
{
    // The root page of your application
    MainPage = new ContentPage
    {
        Content = Class1.ReturnMyControl()
    };
}
```

3. Few additional steps are required to run the iOS and UWP projects. For example, the following steps need to be performed in case of Gauges:
  - o **iOS Project:**
    1. In the **Solution Explorer**, double click AppDelegate.cs inside YourAppName.iOS project to open it.
    2. Add the following code to the FinishedLaunching() method.

```
C#
C1.Xamarin.Forms.Gauge.Platform.iOS.C1GaugeRenderer.Init();
```

- o **UWP Project:**
  1. In the **Solution Explorer**, expand MainPage.xaml.
  2. Double click MainPage.xaml.cs to open it.
  3. Add the following code to the class constructor.

```
C#
C1.Xamarin.Forms.Gauge.Platform.UWP.C1GaugeRenderer.Init();
```

4. Press **F5** to run the project.

[Back to Top](#)

## Adding Xamarin Components using XAML

This topic demonstrates how to add a Xamarin control to your app using XAML. This is done in three steps:

- **Step 1: Add a new Forms XAML Page**
- **Step 2: Add the Control**
- **Step 3: Run the Program**

### Step 1: Add a new Forms XAML Page

1. In the **Solution Explorer**, right click the project YourAppName (Portable or Shared).
2. Select **Add | New**. The **Add New Item** dialog appears.
3. Under installed templates, select **C# | Forms XAML Page**.
4. Add a name for the XAML page (for example Page1.xaml) and click **OK**. A new XAML page is added to your project.

### Step 2: Add the Control

1. In the **Solution Explorer**, double click Page1.xaml to open it.
2. Modify the <ContentPage> tag to include the following references:

XAML

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="YourAppName.Page1"
xmlns:cl="clr-namespace:C1.Xamarin.Forms.Gauge;assembly=C1.Xamarin.Forms.Gauge">
```

3. Initialize the control in between the <ContentPage> </ContentPage> tags and inside the <StackLayout> </StackLayout> tags.

The following code shows how to initialize a Gauge control.

XAML

```
<StackLayout>
<cl:C1LinearGauge Value="35" Min="0" Max="100" Thickness="0.1"
HeightRequest="50" WidthRequest="50" PointerColor="Blue" Direction="Right">
<cl:C1LinearGauge.Ranges>
<cl:GaugeRange Min="0" Max="40" Color="Red"/>
<cl:GaugeRange Min="40" Max="80" Color="Yellow"/>
<cl:GaugeRange Min="80" Max="100" Color="Green"/>
</cl:C1LinearGauge.Ranges>
</cl:C1LinearGauge>
</StackLayout>
```

[Back to Top](#)

### Step 3: Run the Program

1. In the **Solution Explorer**, double click App.cs to open it.
2. In the class constructor App(), set the Forms XAML Page Page1 as the MainPage.

The following code shows the class constructor App(), after completing this step.

```
C#  
  
public App()  
{  
    // The root page of your application  
    MainPage = new Page1();  
}
```

3. Few additional steps may be required for some controls. For example, the following steps need to be performed in case of Gauge to run an iOS app and a UWP app:

- **iOS App:**

1. In the **Solution Explorer**, double click AppDelegate.cs inside YourAppName.iOS project to open it.
2. Add the following code to the FinishedLaunching() method.

```
C#  
  
Cl.Xamarin.Forms.Gauge.Platform.iOS.ClGaugeRenderer.Init();
```

- **UWP App:**

1. In the **Solution Explorer**, expand MainPage.xaml.
2. Double click MainPage.xaml.cs to open it.
3. Add the following code to the class constructor.

```
C#  
  
Cl.Xamarin.Forms.Gauge.Platform.UWP.ClGaugeRenderer.Init();
```

4. Press **F5** to run the project.

**Back to Top**

## About this Documentation

### Acknowledgements

Microsoft, Windows, Windows Vista, Windows Server, and Visual Studio are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

### Contact Us

If you have any suggestions or ideas for new features or controls, please call us or write:

**ComponentOne, a division of GrapeCity**

201 South Highland Avenue, Third Floor  
Pittsburgh, PA 15206 • USA  
1.800.858.2739 | 412.681.4343  
412.681.4384 (Fax)

<http://www.componentone.com/>

## Technical Support

Xamarin Edition offers various support options. For a complete list and a description of each, visit the [ComponentOne website](#) to explore more.

Some methods for obtaining technical support include:

- **Online Resources**

ComponentOne provides customers with a comprehensive set of technical resources in the form of [Licensing FAQs](#), [samples](#), [demos](#), and [videos](#), [searchable online documentation](#) and more. We recommend this as the first place to look for answers to your technical questions.

- **Online Support**

The online support service provides you direct access to our Technical Support staff via [Submit a ticket](#). When you submit an incident, you immediately receive a response via e-mail confirming that the incident is created successfully. This email provides you with an Issue Reference ID. You will receive a response from us via an e-mail within two business days.

- **Product Forums**

[Forums](#) are available for users to share information, tips, and techniques regarding all the platforms supported by the ComponentOne Xamarin Edition, including Xamarin.Forms, Xamarin.iOS and Xamarin.Android. ComponentOne developers or community engineers will be available on the forums to share insider tips and technique and answer users' questions. Note that a user account is required to participate in the Forums.

- **Installation Issues**

Registered users can obtain help with problems installing Xamarin Edition on their systems. Contact technical support by using the online incident submission form or by phone (412.681.4738). Please note that this does not include issues related to distributing a product to end-users in an application.

- **Documentation**

[ComponentOne documentation](#) is available online for viewing. If you have suggestions on how we can improve our documentation, please send a [feedback](#) to the Documentation team. Note that the feedback sent to the Documentation team is for documentation related issues only. [Technical support](#) and [sales](#) issues should be sent directly to their respective departments.



**Note:** You must create a user account and register your product with a valid serial number to obtain support using some of the above methods.

## Controls

### Calendar

The Calendar control provides a calendar through which you can navigate to any date in any year. Calendar comes with an interactive date selection user interface (UI) with month, year and decade view modes. Users can view as well as select multiple dates on the calendar.

Calendar provides the ability to customize day slots so that users can visualize date information on the calendar. In addition, you can also customize the appearance of the calendar using your own content and style.

#### Key Features

- **Custom Day Content:** Customize the appearance of day slots by inserting custom content.
- **View Modes:** Tap header to switch from month mode to year and decade mode.
- **Appearance:** Easily style different parts of the control with heavy customizations.
- **Date Range Selection:** Simply tap two different dates to select all the dates in between.
- **Orientation:** Toggle the scroll orientation to either horizontal or vertical.

### Quick Start: Display a Calendar Control

This section describes how to add a Calendar control to your Xamarin application and select a date on the calendar at runtime. This topic comprises of two steps:

- **Step 1: Add a Calendar Control**
- **Step 2: Run the Project**

The following image shows how the Calendar appears after completing the above steps.



### Step 1: Add a Calendar Control

Complete the following steps to initialize a Calendar control in C# or XAML.

#### In Code

1. Add a new class (say QuickStart.cs) to your Portable or Shared project and include the following references.

C#

```
using Xamarin.Forms;
using Cl.Xamarin.Forms.Calendar;
```

2. Instantiate a Calendar control in a new method ReturnMyControl() as illustrated in the code below.

C#

```
public static ClCalendar ReturnMyControl()
{
    ClCalendar calendar = new ClCalendar();
    calendar.MaxSelectionCount = -1;
    calendar.HorizontalOptions = Xamarin.Forms.LayoutOptions.Center;
    calendar.FontSize = 30;
    return calendar;
}
```

#### In XAML

1. Add a new Forms XAML Page (say QuickStart.xaml) to your Portable or Shared project and modify the

<ContentPage> tag to include the following references:

#### XAML

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
xmlns:cl="clr-
namespace:C1.Xamarin.Forms.Calendar;assembly=C1.Xamarin.Forms.Calendar"
x:Class="CalendarQuickStart.QuickStart"
Padding="20">
```

2. Initialize a Calendar control by adding the following markup to the control between the <ContentPage> </ContentPage> tags inside the <Grid> </Grid> tags as illustrated below.

#### XAML

```
<Grid>
    <Label Text="{Binding MainText}" HorizontalOptions="Center" Font="Large" />
    <cl:C1Calendar x:Name="calendar" MaxSelectionCount="-1"/>
</Grid>
</ContentPage>
```

### Step 3: Run the Project

1. In the Solution Explorer, double-click App.cs file to open it.
2. Complete the following steps to display the Calendar control.
  - **To return a C# class:** In the class constructor App(), set a new ContentPage as the MainPage and assign the control to the ContentPage's Content by invoking the ReturnMyControl() method in **Step 2** above. The following code shows the class constructor App() after completing this step.

#### C#

```
public App ()
{
    // The root page of your application
    MainPage = new ContentPage
    {
        Content = QuickStart.ReturnMyControl ()
    };
}
```

- To return a Forms XAML Page: In the constructor App(), set the Forms XAML Page QuickStart as the MainPage. The following code shows the class constructor App(), after completing this step.

#### C#

```
public App ()
{
    // The root page of your application
    MainPage = new QuickStart ();
}
```

3. Some additional steps are required to run iOS and UWP apps:
  - **iOS App:**
    1. In the Solution Explorer, double click AppDelegate.cs inside YourAppName.iOS project to open it.
    2. Add the following code to the FinishedLaunching() method.

#### C#

```
C1.Xamarin.Forms.Calendar.Platform.iOS.C1CalendarRenderer.Init ();
```

- **UWP App:**

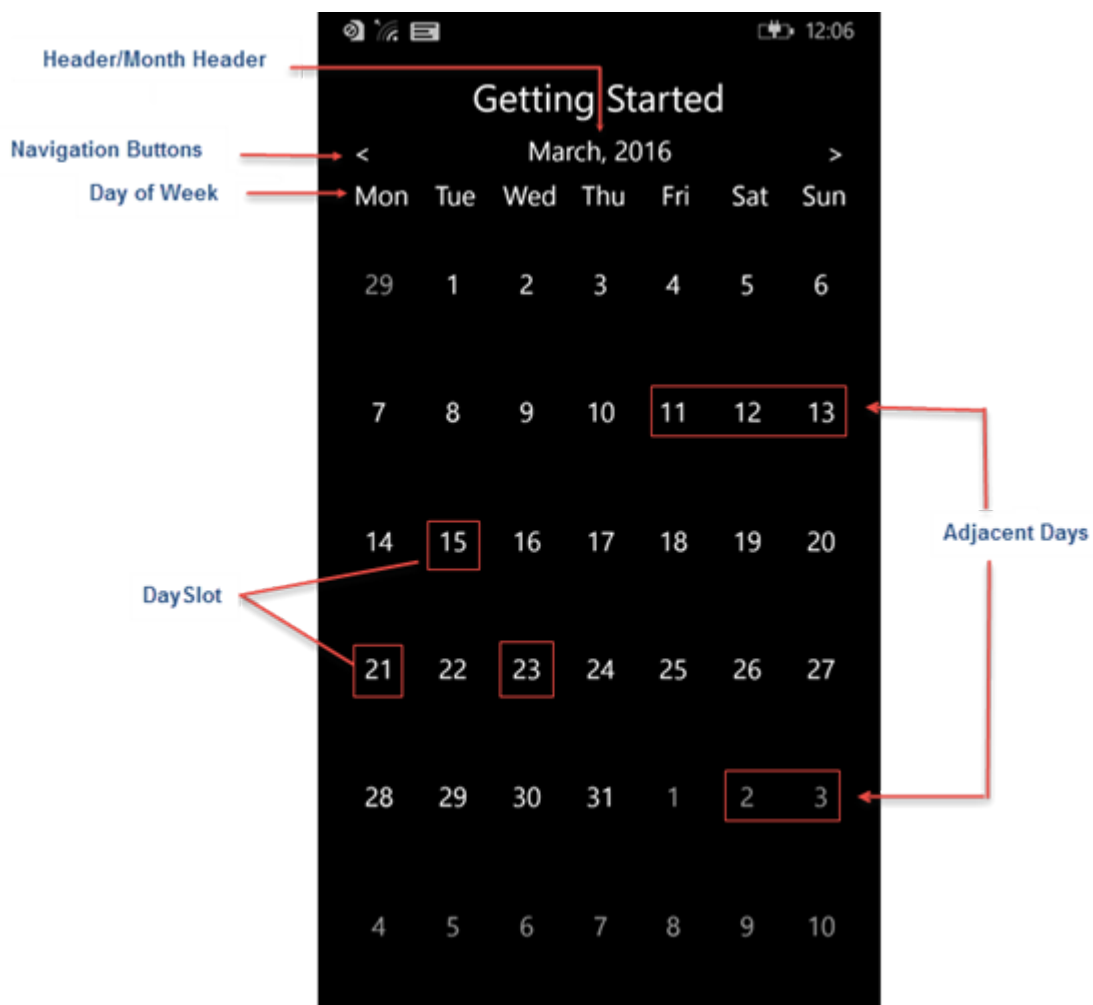
1. In the Solution Explorer, expand the MainPage.xaml inside YouAppName.UWP project.
2. Double click the MainPage.xaml.cs to open it and add the following code to the class constructor.

```
C#  
C1.Xamarin.Forms.Calendar.Platform.UWP.C1CalendarRenderer.Init();
```

## Interaction Guide

The Calendar control comprises of various functional parts such as Header, Day of Week, Day Slot, etc. These functional parts are illustrated below.

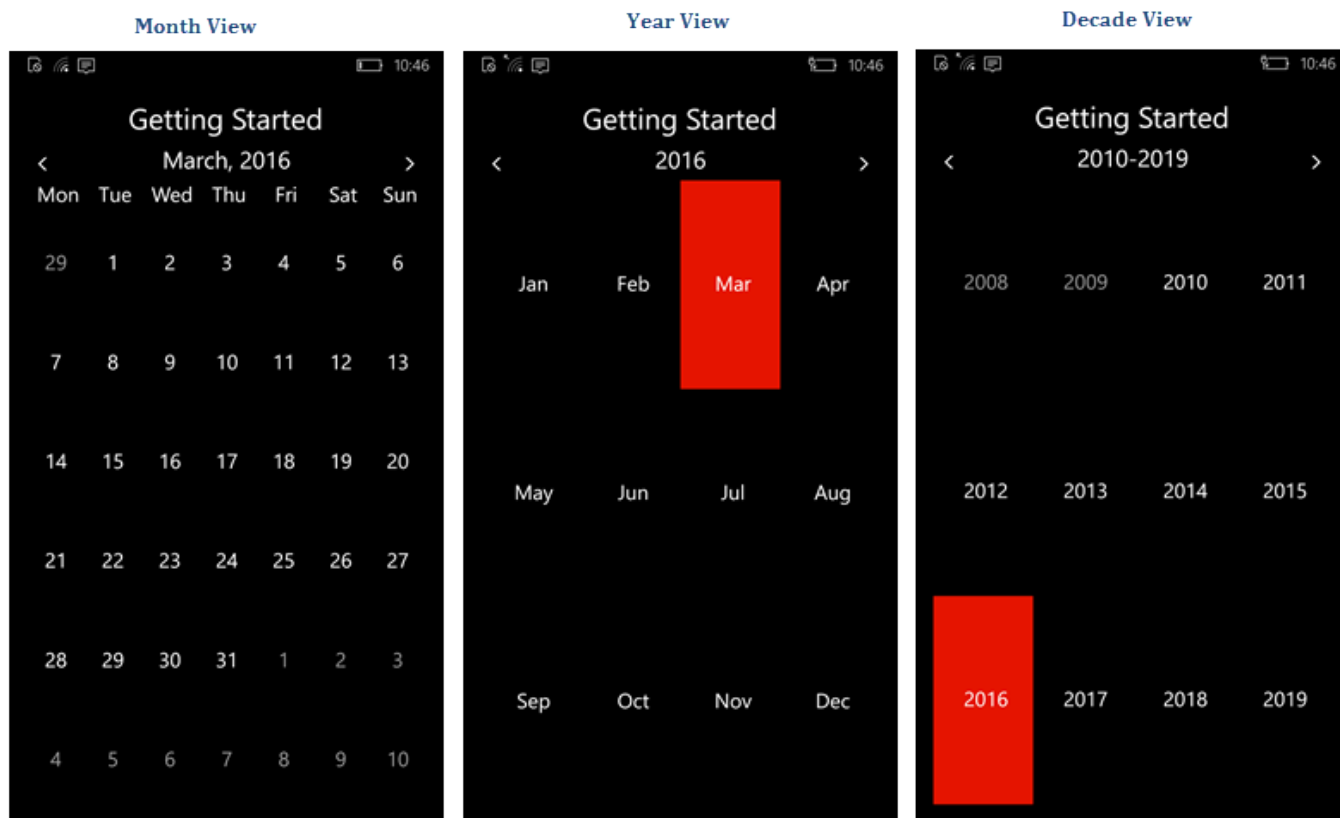
- **Header** - The Calendar comprises a header that displays the current month, year or decade along with navigation buttons. Users can hide the default header and create their own headers as illustrated in [Customizing Header](#).
- **Day Slot** - The Calendar features day slots which can be customized to display custom content. See [Customizing Day Content](#) to understand how day slots can be used to insert and display custom content.
- **Day of Week** - The user interface of Calendar displays seven days of week corresponding to respective dates.
- **Navigation Buttons** - The navigation buttons enables users to traverse the selected month or year, forward or backward.



### View Modes



The Calendar control supports month, year and decade views as shown in the image below. The calendar switches from month to year view when the user taps the month header. On tapping the year header, the calendar switches to decade view. The calendar switches back to the month view on tapping the header on the decade view, completing a full circle from Month>Year>Decade>Month view.



## Features

### Customizing Appearance

The Calendar control provides various built-in properties to customize calendar's appearance. You can use these properties to set calendar's background color, text color, header color, font size, header font size, selection background color, etc.

The image below shows a customized calendar after setting these properties.

< March, 2016 >						
M	T	W	T	F	S	S
29	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3
4	5	6	7	8	9	10

The following code example demonstrates how to set these properties in C# and XAML. This example uses the C# and XAML samples created in the [Quick start](#).

#### In Code

C#

```
//Customizing Appearance
calendar.BackgroundColor = Xamarin.Forms.Color.White;
calendar.TextColor = Xamarin.Forms.Color.Black;
calendar.DayBorderColor = Color.FromHex("#ABD0ED");
calendar.DayBorderWidth = 1;
calendar.FontFamily = "Segoe UI";
calendar.FontSize = 16;
calendar.BorderColor = Xamarin.Forms.Color.Black;
calendar.BorderWidth = 4;

calendar.DayOfWeekBackgroundColor = Color.FromHex("#FCC989");
calendar.DayOfWeekTextColor = Xamarin.Forms.Color.Black;
calendar.DayOfWeekFormat = "d";
calendar.DayOfWeekFontFamily = "Segoe UI";
calendar.DayOfWeekFontSize = 21;

calendar.HeaderBackgroundColor = Color.FromHex("#B1DCB6");
calendar.HeaderTextColor = Xamarin.Forms.Color.Black;
calendar.HeaderFontFamily = "Segoe UI";
calendar.HeaderFontSize = 21;
calendar.SelectionBackgroundColor = Xamarin.Forms.Color.Red;
calendar.TodayFontAttributes = FontAttributes.Italic;
```

## In XAML

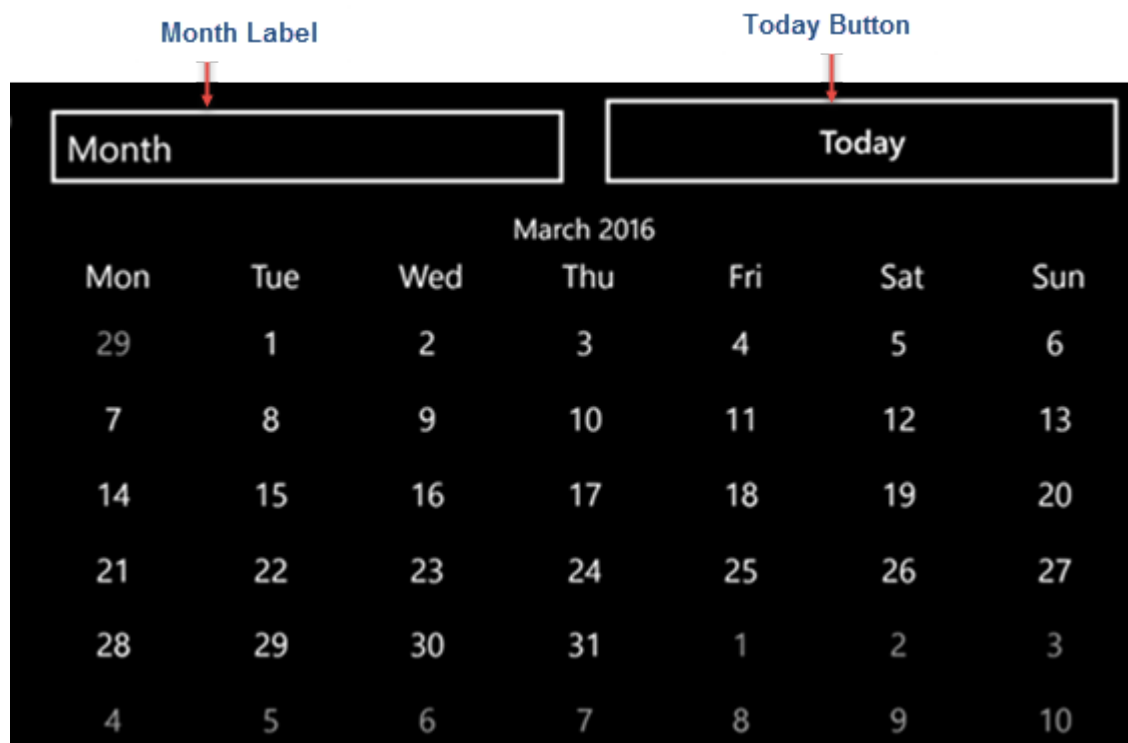
## XAML

```
<cl:ClCalendar x:Name="calendar" MaxSelectionCount="-1" BackgroundColor="White"
    TextColor="Black" AdjacentDayTextColor="#FFA5A5A3" DayBorderColor="#ABD0ED"
    DayBorderWidth="1"
    FontFamily="Segoe UI" FontSize="16" BorderColor="Black" BorderWidth="4"
    DayOfWeekBackgroundColor="#FCC989"
    DayOfWeekTextColor="Black" DayOfWeekFormat="d" DayOfWeekFontFamily="Segoe UI"
    DayOfWeekFontSize="21"
    HeaderBackgroundColor="#B1DCB6" HeaderTextColor="Black" HeaderFontFamily="Segoe UI"
    HeaderFontSize="21"
    SelectionBackgroundColor="Red" TodayFontAttributes="Italic"/>
```

## Customizing Header

The Calendar control shows a default header that displays the current month or year and navigation buttons. However, users can hide or remove the default header by setting the [ShowHeader](#) to **false** in XAML, and apply a custom header.

The following image shows a calendar with a custom header.



On tapping the **Month** label, the calendar provides option to switch to year or decade mode. The **Today** label navigates the calendar to the current day.

The following code example demonstrates how to create and apply a custom header in Calendar control in C# and XAML. This example uses the sample created in [Quick start](#).

1. Add a new Forms Xaml page, CustomizingHeader.xaml, to your project.
2. To initialize a Calendar control and applying a custom header in XAML, modify the entire XAML markup as shown below.

#### In XAML

##### XAML

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:cl="clr-
namespace:C1.Xamarin.Forms.Calendar;assembly=C1.Xamarin.Forms.Calendar"
             x:Class="C1CalendarCustomHeader.CustomizingHeader" x:Name="page">
    <Grid>
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <Picker x:Name="modePicker"/>
            <Button x:Name="todayButton" Text="Today" Grid.Column="1"/>
        </Grid>
        <Label x:Name="monthLabel" HorizontalOptions="Center"/>
        <cl:C1Calendar x:Name="calendar" ShowHeader="False"
VerticalOptions="FillAndExpand"/>
    </Grid>
</ContentPage>
```

3. In the Solution Explorer, expand the CustomizingHeader.xaml node and open CustomizingHeader.xaml.cs to open the C# code behind.
4. Add the following code in the CustomizingHeader class to apply a custom header and add functionality to the Month label and Today button.

#### In Code

##### C#

```
public partial class CustomizingHeader : ContentPage
{
    public CustomizingHeader()
    {
        InitializeComponent();
        //Title = AppResources.CustomHeaderTitle;

        modePicker.Items.Add(AppResources.MonthLabel);
        modePicker.Items.Add(AppResources.YearLabel);
        modePicker.Items.Add(AppResources.DecadeLabel);
        modePicker.SelectedIndex = 0;
        modePicker.SelectedIndexChanged += OnModeChanged;

        todayButton.Clicked += OnTodayClicked;
        calendar.ViewModeChanged += OnViewModeChanged;
        calendar.DisplayDateChanged += OnDisplayDateChanged;
```

```
        UpdateMonthLabel();
    }
    public string TodayLabel
    {
        get
        {
            return AppResources.TodayLabel;
        }
    }

    private void OnModeChanged(object sender, System.EventArgs e)
    {
        switch (modePicker.SelectedIndex)
        {
            case 0:
                calendar.ChangeViewModeAsync(CalendarViewMode.Month);
                break;
            case 1:
                calendar.ChangeViewModeAsync(CalendarViewMode.Year);
                break;
            case 2:
                calendar.ChangeViewModeAsync(CalendarViewMode.Decade);
                break;
        }
    }

    private void OnTodayClicked(object sender, System.EventArgs e)
    {
        calendar.ChangeViewModeAsync(CalendarViewMode.Month,
DateTime.Today);
    }

    private void OnViewModeChanged(object sender, EventArgs e)
    {
        switch (calendar.ViewMode)
        {
            case CalendarViewMode.Month:
                modePicker.SelectedIndex = 0;
                break;
            case CalendarViewMode.Year:
                modePicker.SelectedIndex = 1;
                break;
            case CalendarViewMode.Decade:
                modePicker.SelectedIndex = 2;
                break;
        }
    }

    private void OnDisplayDateChanged(object sender, EventArgs e)
    {
        UpdateMonthLabel();
    }
}
```

```

    }

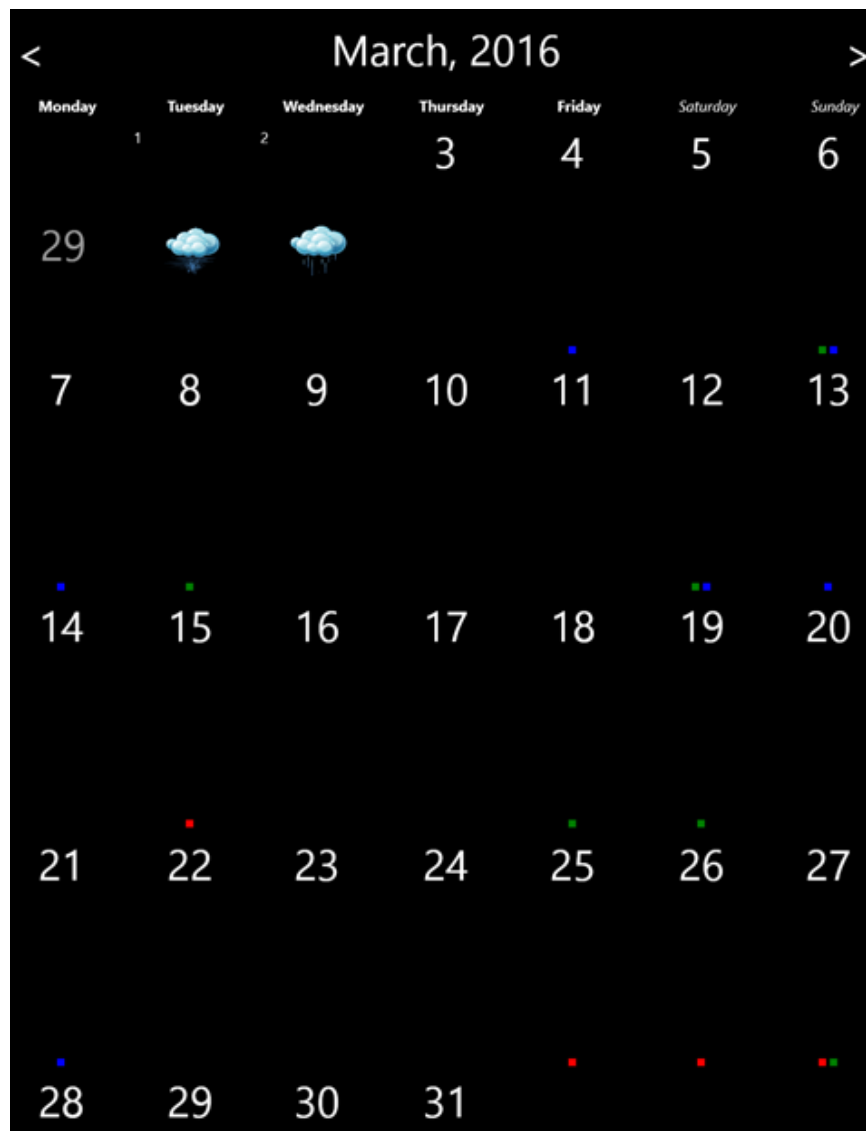
    private void UpdateMonthLabel()
    {
        monthLabel.Text = string.Format("{0:MMMM yy}",
calendar.DisplayDate);
        modePicker.SelectedIndex = 0;
    }
}

```

## Customizing Day Content

The Calendar control allows users to add custom content to day slot. For this, all you need to do is subscribe the [DaySlotLoading](#) event of the `CalendarViewDaySlot` class and apply custom content such as images in the background of these slots. This feature allows users to display weather related information on the calendar.

The image below shows a calendar after adding custom content to day slots. The calendar displays weather related information through various icons.



The following code example demonstrates how to add custom content to day slots in Calendar control in C# and XAML. This example uses the sample created in [Quick start](#).

1. Add a new Forms Xaml Page, Custom Day Content, to your portable project.
2. Add the following import statements in the CustomDayContent.xaml.cs file of your portable project.

```
C#
using Xamarin.Forms;
using C1.Xamarin.Forms.Calendar;
```

3. To initialize a calendar control and adding custom day content, modify the XAML markup as shown below.

## In XAML

```

XAML
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:cl="clr-
namespace:C1.Xamarin.Forms.Calendar;assembly=C1.Xamarin.Forms.Calendar"
             xmlns:c1="clr-namespace:Calendar101;assembly=Calendar.Xamarin"
             x:Class="CustomDayContent.DayContent" x:Name="page">

    <Grid>
        <c1:C1Calendar DayOfWeekFontSize="8" DayOfWeekFormat="dddd"
DayOfWeekFontAttributes="Italic"
            DaySlotLoading="OnDaySlotLoading" DayOfWeekSlotLoading="OnDayOfWeekSlotLoading"
            VerticalOptions="FillAndExpand">
            <c1:C1Calendar.DaySlotTemplate>
                <DataTemplate>
                    <StackLayout Padding="4">
                        <Label Text="{Binding Day}" VerticalOptions="FillAndExpand"/>
                        <StackLayout HorizontalOptions="Center"
Orientation="Horizontal" Spacing="2">
                            <Grid WidthRequest="4" HeightRequest="4"
BackgroundColor="Red"
                                IsVisible="{Binding RedDotVisible}"/>
                            <Grid WidthRequest="4" HeightRequest="4"
BackgroundColor="Green"
                                IsVisible="{Binding GreenDotVisible}"/>
                            <Grid WidthRequest="4" HeightRequest="4"
BackgroundColor="Blue"
                                IsVisible="{Binding BlueDotVisible}"/>
                        </StackLayout>
                    </StackLayout>
                </DataTemplate>
            </c1:C1Calendar.DaySlotTemplate>
            <c1:C1Calendar.AdjacentDaySlotTemplate>
                <DataTemplate>
                    <local:CalendarDaySlot DayText="{Binding Day}"
DayHorizontalAlignment="Center"
                        DayVerticalAlignment="Start"/>
                </DataTemplate>
            </c1:C1Calendar.AdjacentDaySlotTemplate>
        </c1:C1Calendar>
    </Grid>
</ContentPage>

```

4. In the **Solution Explorer**, expand the CustomDayContent node and open CustomDayContent.xaml.cs to open the C# code behind.
5. Add the following code in the CustomDayContent.xaml.cs to add custom content to day slots.

## In Code

C#

```

public partial class DayContent : ContentPage
{
    private List<ImageSource> _icons = new List<ImageSource>();
    private Random _rand = new Random();
    private Dictionary<DateTime, ImageSource> WeatherForecast = new
Dictionary<DateTime, ImageSource>();

    public DayContent()
    {
        InitializeComponent();

        _icons.Add(ImageSource.FromResource("CustomDayContent.Images.partly-cloudy-
day-icon.png"));
        _icons.Add(ImageSource.FromResource("CustomDayContent.Images.Sunny-
icon.png"));
        _icons.Add(ImageSource.FromResource("CustomDayContent.Images.rain-
icon.png"));
        _icons.Add(ImageSource.FromResource("CustomDayContent.Images.snow-
icon.png"));
        _icons.Add(ImageSource.FromResource("CustomDayContent.Images.thunder-
lightning-storm-icon.png"));
        _icons.Add(ImageSource.FromResource("CustomDayContent.Images.Overcast-
icon.png"));

        for (int i = 0; i > 10; i++)
        {
            WeatherForecast[DateTime.Today.AddDays(i)] = GetRandomIcon();
        }
    }

    public void OnDaySlotLoading(object sender, CalendarDaySlotLoadingEventArgs e)
    {
        if (!e.IsAdjacentDay)
        {
            if (WeatherForecast.ContainsKey(e.Date))
            {
                var daySlotWithImage = new CalendarImageDaySlot(e.Date);
                daySlotWithImage.DayText = e.Date.Day + "";
                daySlotWithImage.DayFontSize = 8;
                daySlotWithImage.ImageSource = WeatherForecast[e.Date];
                e.DaySlot = daySlotWithImage;
            }
            else
            {
                e.DaySlot.BindingContext = new MyDataContext(e.Date);
            }
        }
        else
        {
            e.DaySlot.BindingContext = new MyDataContext(e.Date);
        }
    }
}

```



```

        public void OnDayOfWeekSlotLoading(object sender,
CalendarDayOfWeekSlotLoadingEventArgs e)
        {
            if (!e.IsWeekend)
            {
                (e.DayOfWeekSlot as CalendarDayOfWeekSlot).DayOfWeekFontAttributes =
FontAttributes.Bold;
                (e.DayOfWeekSlot as CalendarDayOfWeekSlot).DayOfWeekFontSize = 8;
            }
            else
            {
                (e.DayOfWeekSlot as CalendarDayOfWeekSlot).DayOfWeekFontAttributes =
FontAttributes.Italic;
                (e.DayOfWeekSlot as CalendarDayOfWeekSlot).DayOfWeekFontSize = 8;
            }
        }

        private ImageSource GetRandomIcon()
        {
            return _icons[_rand.Next(0, _icons.Count - 1)];
        }
    }

    public class MyDataContext
    {
        private static Random _rand = new Random();
        public MyDataContext(DateTime date)
        {
            Day = date.Day;
            RedDotVisible = Day % 3 == 0;
            GreenDotVisible = Day % 3 == 0;
            BlueDotVisible = Day % 3 == 0;
        }

        public int Day { get; set; }
        public bool RedDotVisible { get; set; }
        public bool GreenDotVisible { get; set; }
        public bool BlueDotVisible { get; set; }
    }
}

```

## Orientation

The Calendar appears in default horizontal orientation. However, you can change the orientation of the calendar to Vertical by using the [Orientation](#) property. The [C1Calendar](#) class provides CalendarOrientation enumeration that can be set to set Vertical orientation as shown in the code below.

The following code example demonstrates how to set the orientation in C# and XAML. This code example uses the sample created in the [Quick start](#).

### In Code

C#

```
//Setting the Orientation
```

```
calendar.Orientation = CalendarOrientation.Vertical;
```

### In XAML

#### XAML

```
<Grid>
  <Label Text="{Binding MainText}" HorizontalOptions="Center" Font="Large" />
  <cl:C1Calendar x:Name="calendar" MaxSelectionCount="-1" DayOfWeekFontSize="21"
    HeaderBackgroundColor="#B1DCB6" HeaderTextColor="Black"
    HeaderFontFamily="Segoe UI"
    Orientation="Vertical"/>
</Grid>
```

## Selection

The Calendar control allows users to select a day on the calendar by tapping a date. However, you can set the number of days that you wish to select by using the [MaxSelectionCount](#) property in code. For instance, on setting the **MaxSelectionCount** property to 5, you can select a maximum of 5 days on the calendar as illustrated in the image below.



The following code examples illustrate how to set maximum selection in C# and XAML. The following examples uses the samples created in the [Quick Start](#).

### In Code

#### C#

```
// setting maximum selection
calendar.MaxSelectionCount = 5;
```

## In XAML

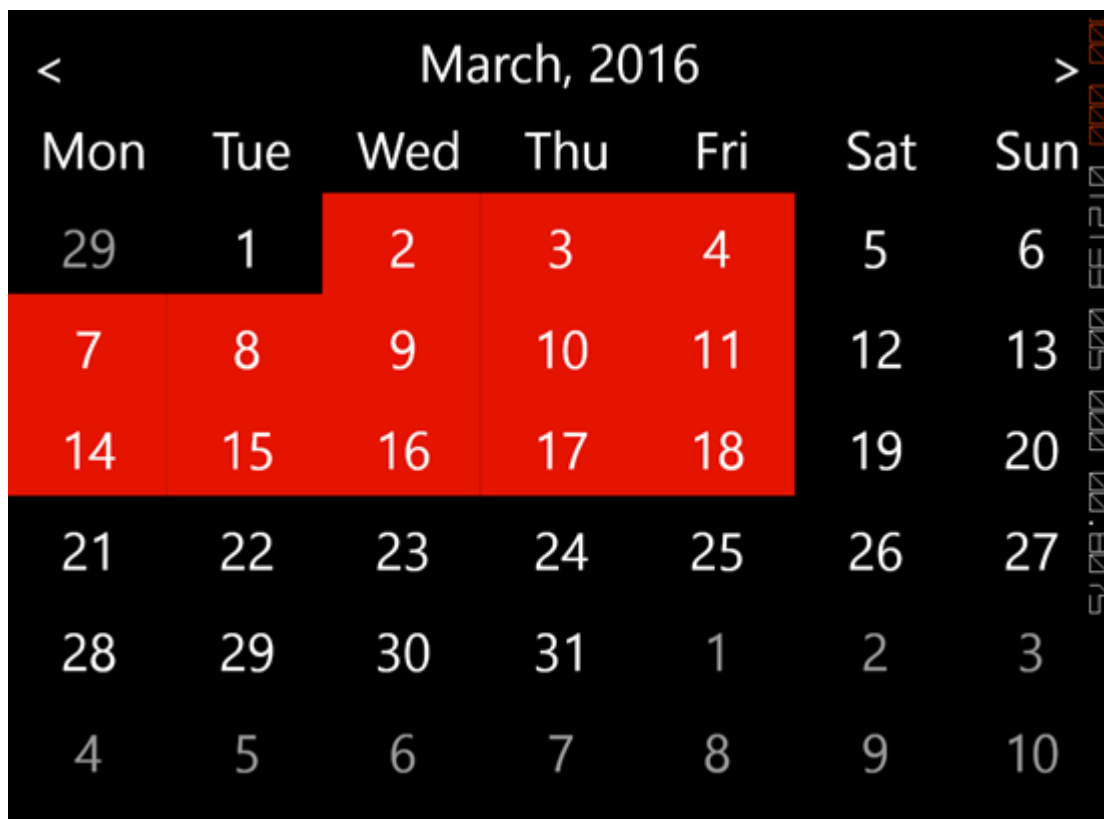
## XAML

```
<Grid>
  <Label Text="{Binding MainText}" HorizontalOptions="Center" Font="Large" />
  <cl:C1Calendar x:Name="calendar" MaxSelectionCount="5" />
</Grid>
```

## Customizing Selection

You can customize the default behavior of the calendar control to select specific dates. For instance, consider a scenario where you wish to select only weekdays on tapping two dates in different workweeks. For this, you simply need to subscribe the **OnSelectionChanging** event and apply selection condition in the handler.

The following image shows a calendar that only selects weekdays and deselects weekends on tapping two different dates in different workweeks.



The following code example demonstrates how to customize selection in C# and XAML. This code example uses the sample created in the [Quick start](#).

## In Code

1. Subscribe the SelectionChanging event in XAML between the <Grid></Grid> tags as depicted below.

## XAML

```
<Grid>
  <cl:C1Calendar SelectionChanging="OnSelectionChanging" MaxSelectionCount="-
1"/>
```

```
</Grid>
```

2. Switch to the code view and add the following code to select only weekdays between two dates in two different weeks.

C#

```
private void OnSelectionChanging(object sender,
CalendarSelectionChangingEventArgs e)
{
    foreach (var date in e.SelectedDates.ToArray())
    {
        if (date.DayOfWeek == DayOfWeek.Saturday || date.DayOfWeek ==
DayOfWeek.Sunday)
            e.SelectedDates.Remove(date);
    }
}
```

## CollectionView

**CollectionView** is a powerful data binding component that is designed to be used with data controls, such as **ListBox** and **FlexGrid**. **CollectionView** provides currency, filtering, grouping and sorting services for your data collection.

The **CollectionView** class implements the following interface:

- **ICollectionView**: provides current record management, custom sorting, filtering, and grouping.

### Key Features

- Provides **filtering**, **grouping** and **sorting** on a data set.
- Can be used with the data collection controls, such as **FlexGrid**.
- Provides **currency for master-detail** support for Xamarin applications.
- Based on the **.NET implementation** of **ICollectionView**.

## Features

### Grouping

The **CollectionView** interface supports grouping for data controls, such as **FlexGrid** and **ListBox**. To enable grouping, add one or more **GroupDescription** objects to the **GroupedCollectionView** property. **GroupDescription** objects are flexible, allowing you to group data based on value or on grouping functions.

The image below shows how the **FlexGrid** appears, after **Grouping** is applied to column **Country**.

	ID	Name	Country	CountryID
	▲ Key: Brazil (1 items)			
	0	Noah Jammers	Brazil	4
	▶ Key: Indonesia (1 items)			
	▶ Key: Vietnam (1 items)			
	▲ Key: Egypt (1 items)			
	3	Charlie Heath	Egypt	15
	▲ Key: Russia (2 items)			
	4	Oprah Stevens	Russia	8
	9	Noah Neiman	Russia	8

The following code example demonstrates how to apply Grouping in FlexGrid in C# and XAML. The example uses the data source, **Customer.cs**, created in the FlexGrid's [Quick start](#) section.

1. Add a new Forms XAML Page, **Grouping.xaml**, to your project.
2. To initialize a FlexGrid control and enabling grouping in XAML, modify the markup between the `<ContentPage>` `</ContentPage>` tags and inside the `<StackLayout>` `</StackLayout>` tags, as shown below.

#### In XAML

```
XAML

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="Appl.Grouping"
xmlns:cl="clr-namespace:C1.Xamarin.Forms.Grid;assembly=C1.Xamarin.Forms.Grid">
    <ContentPage.ToolbarItems>
        <ToolbarItem x:Name="tb_Collapse" Text="Collapse"></ToolbarItem>
    </ContentPage.ToolbarItems>
    <StackLayout>
        <Grid VerticalOptions="FillAndExpand">
            <cl:Grid x:Name="grid" AutoGenerateColumns="True" IsGroupingEnabled="True"
/>
        </Grid>
    </StackLayout>
</ContentPage>
```

3. In the **Solution Explorer**, expand the Grouping.xaml node and open Grouping.xaml.cs to open the C# code.
4. Add the following code in the Grouping class constructor to apply grouping to the column **Country** in the FlexGrid:

#### In Code

C#

```
C1CollectionView<Customer> _collectionView;

public Grouping()
{
    InitializeComponent();

    this.Title = AppResources.GroupingTitle;
    grid.SelectionChanging += OnSelectionChanging;

    var task = UpdateVideos();
}

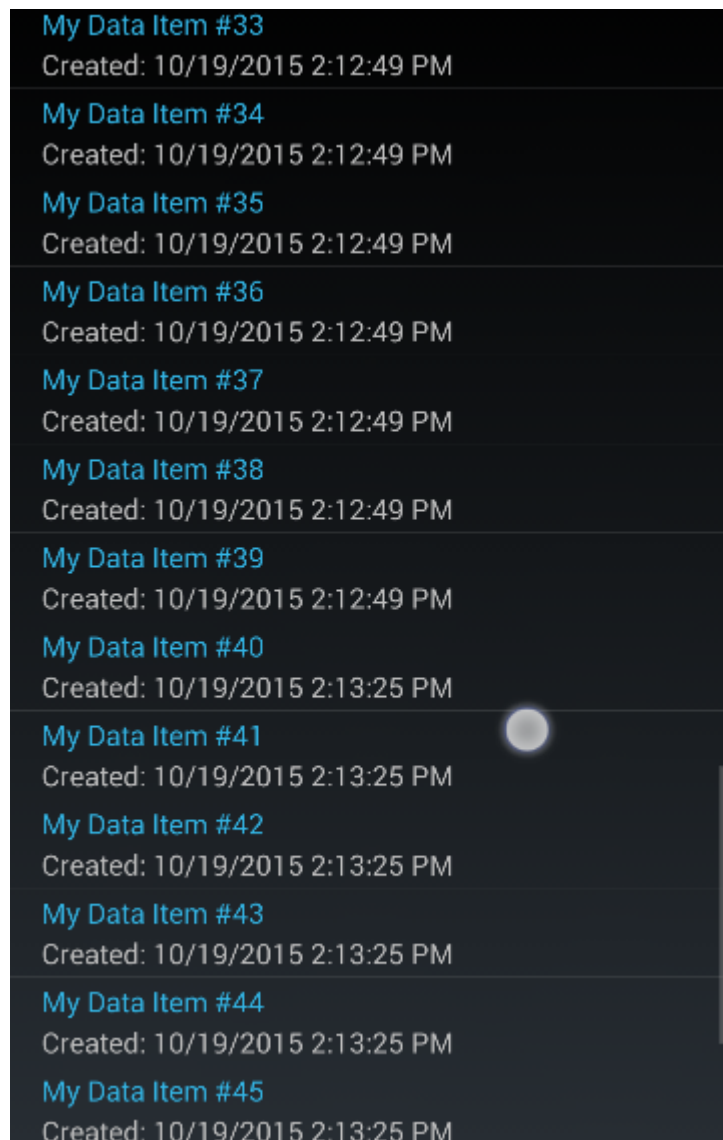
private async Task UpdateVideos()
{
    var data = Customer.GetCustomerList(100);
    _collectionView = new C1CollectionView<Customer>(data);
    await _collectionView.GroupAsync(c => c.Country);
    grid.ItemsSource = _collectionView;
}

public void OnSelectionChanging(object sender, GridCellRangeEventArgs e)
{
    if (e.CellType == GridCellType.Cell || e.CellType ==
GridCellType.RowHeader)
    {
        var row = grid.Rows[e.CellRange.Row] as GridGroupRow;
        if (row != null)
            e.Cancel = true;
    }
}
```

## Incremental Loading

Incremental loading (on demand loading) is a powerful feature for any mobile application. CollectionView supports incremental loading for data bound controls, such as FlexGrid, ListView. To apply incremental loading on your Xamarin.Forms application, first you need to implement collection view class, which extends a cursor CollectionView and overrides **GetPageAsync**. Once you have implemented this, you need to add the logic that loads the data in pages or chunks. You can also set the number of pages to be loaded at a time.

The image below shows how the ListBox appears when incremental loading is applied on it.



The following code example demonstrates how to implement on demand loading for a simple ListView control using CollectionView.

1. Add a new Forms XAML Page, **IncrementalLoading.xaml** to your project.
2. To initialize a ListBox control and enabling incremental loading, modify the markup between the `<ContentPage>``</ContentPage>` as shown below.

#### In XAML

##### XAML

```
<ListView x:Name="list">
    <ListView.ItemTemplate>
        <DataTemplate>
            <TextCell Text="{Binding ItemName}" Detail="{Binding ItemDateTime,
                StringFormat='Created: {0}'}" />
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

3. In the **Solution Explorer**, expand the `IncrementalLoading.xaml` node and open `IncrementalLoading.xaml.cs` to open the C# code behind.
4. Add the following code in the `IncrementalLoading` class constructor to implement on demand loading in the `LostBox` control:

#### In Code

C#

```
public partial class SimpleOnDemand : ContentPage
{
    public SimpleOnDemand()
    {
        InitializeComponent();

        Title = AppResources.SimpleOnDemandTitle;

        // instantiate our on demand collection view
        var myCollectionView = new SimpleOnDemandCollectionView();
        list.ItemsSource = myCollectionView;

        // start on demand loading
        list.LoadItemsOnDemand(myCollectionView);
    }
}

public class SimpleOnDemandCollectionView :
ClCursorCollectionView<MyDataItem>
{
    public SimpleOnDemandCollectionView()
    {
        PageSize = 10;
    }

    public int PageSize { get; set; }
    protected override async Task<Tuple<string, IReadOnlyList<MyDataItem>>>
        GetPageAsync(string pageToken, int? count = null)
    {
        // create new page of items
        var newItems = new List<MyDataItem>();
        for (int i = 0; i < this.PageSize; i++)
        {
            newItems.Add(new MyDataItem(this.Count + i));
        }

        return new Tuple<string, IReadOnlyList<MyDataItem>>("token not
used", newItems);
    }
}

public class MyDataItem
{

```



```
public MyDataItem(int index)
{
    this.ItemName = "My Data Item #" + index.ToString();
    this.ItemDateTime = DateTime.Now;
}
public string ItemName { get; set; }
public DateTime ItemDateTime { get; set; }
}
```

## Sorting


CollectionView interface supports ascending and descending sorting for data controls. To enable sorting, add one or more [SortDescription](#) objects to the CollectionView's [SortDescriptions](#) property. To sort columns at runtime, You can simply tap the header of the list to sort data.

SortDescription objects are flexible, they allow you to add objects for individual columns, and set their sorting order to ascending or descending.

The image below shows how the FlexGrid appears after sorting is applied to the column name.

	ID	Name	Country	CountryID
	0	Ben Cole	United States	2
	3	Ben Orsted	Russia	8
	8	Herb Heath	Mexico	10
	1	Herb Stevens	Brazil	4
	6	Jack Heath	Thailand	20
	2	Mark Jammers	Vietnam	12
	4	Noah Heath	Egypt	15

The following code example demonstrates how to sort a FlexGrid control in C# and XAML. This example uses the sample created in the FlexGrid's [Quick Start](#) section.

 Import the following references in the class:

```
using Xamarin.Forms;
using C1.CollectionView;
using C1.Xamarin.Forms.Grid;
```

### In Code

C#

```
public static FlexGrid GetGrid()
{
```

```

    var dataCollection = Customer.GetCustomerList(10);
    ClCollectionView<Customer> cv = new ClCollectionView<Customer>
(dataCollection);
    var sort = cv.SortDescriptions.FirstOrDefault(sd => sd.SortPath == "Name");
    var direction = sort != null ? sort.Direction : SortDirection.Descending;
    cv.SortAsync(x => x.Name, direction == SortDirection.Ascending ? _
SortDirection.Descending : SortDirection.Ascending);
    FlexGrid _grid = new FlexGrid();
    _grid.ItemsSource = cv;
    _grid.VerticalOptions = LayoutOptions.FillAndExpand;
    return _grid;
}

```

### In XAML

#### XAML

```
<Cl.Xamarin.Forms:Grid AllowSorting="True">
```

## FlexChart

**FlexChart** allows you to represent data visually in mobile applications. Depending on the type of data you need to display, you can represent your data as bars, columns, bubbles, candlesticks, lines, scattered points or even display them in multiple chart types.

FlexChart manages the underlying complexities inherent in a chart control completely, allowing developers to concentrate on important application specific tasks.



### Key Features

- **Chart Type:** Change a line chart to a bar chart or any other chart type by setting a single property. FlexChart supports more than ten different chart types.
- **Touch Based Labels:** Display chart values using touch based labels.
- **Multiple Series:** Add multiple series on a single chart.

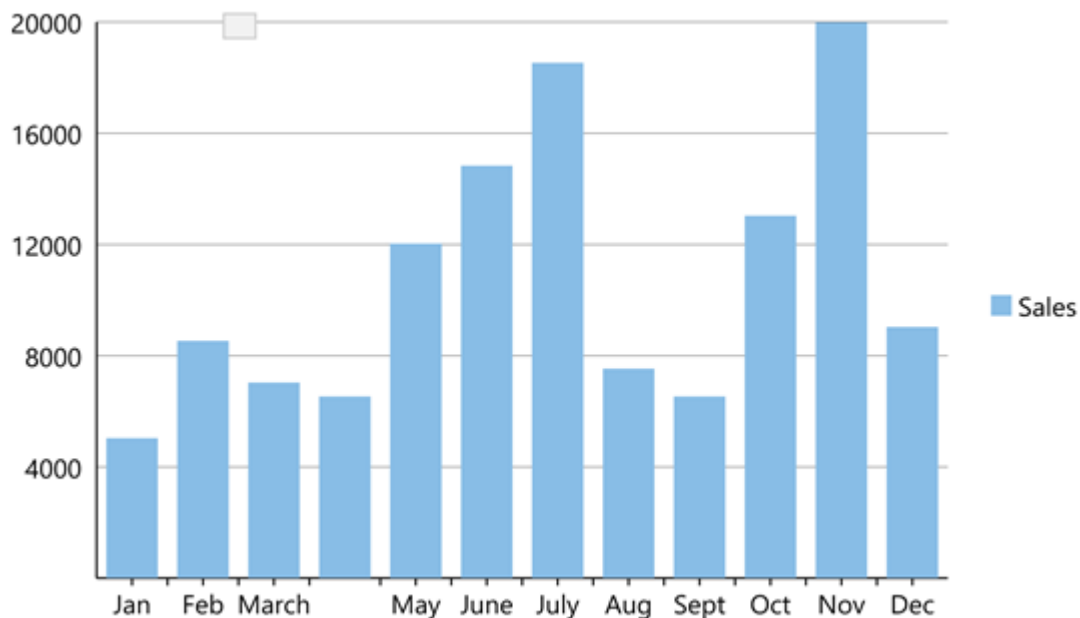
## Quick Start: Add Data to FlexChart

This section describes how to add a FlexChart control to your portable or shared app and add data to it. For information on how to add Xamarin components in C# or XAML, see [Adding Xamarin Components using C#](#) or [Adding Xamarin Components using XAML](#).

This topic comprises of three steps:

- **Step 1: Create a Data source for FlexChart**
- **Step 2: Add a FlexChart control**
- **Step 3: Run the Project**

The following image shows how the FlexChart appears after completing the steps above.



### Step 1: Create a Data source for FlexChart

The following classes serve as a data source for the FlexChart control.

C#

```
public class FlexChartDataSource
{
    private List<Month> appData;

    public List<Month> Data
    {
        get { return appData; }
    }

    public FlexChartDataSource()
    {

```

```
// appData
appData = new List<Month>();
var monthNames =
"Jan, Feb, March, April, May, June, July, Aug, Sept, Oct, Nov, Dec".Split(',');
var salesData = new[] { 5000, 8500, 7000, 6500, 12000, 14800, 18500, 7500,
6500, 13000, 20000, 9000 };
var downloadsData = new[] { 6000, 7500, 12000, 5800, 11000, 7000, 16000,
17500, 19500, 13250, 13800, 19000 };
var expensesData = new[] { 15000, 18000, 15500, 18500, 11000, 16000, 8000,
7500, 6500, 6000, 13500, 5000 };
for (int i = 0; i < 12; i++)
{
    Month tempMonth = new Month();
    tempMonth.Name = monthNames[i];
    tempMonth.Sales = salesData[i];
    tempMonth.Downloads = downloadsData[i];
    tempMonth.Expenses = expensesData[i];
    appData.Add(tempMonth);
}
}

public class Month
{
    string _name;
    long _sales, _downloads, _expenses;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public long Sales
    {
        get { return _sales; }
        set { _sales = value; }
    }

    public long Downloads
    {
        get { return _downloads; }
        set { _downloads = value; }
    }

    public long Expenses
    {
        get { return _expenses; }
        set { _expenses = value; }
    }
}
```

[Back to Top](#)**Step 2: Add a FlexChart control**

Complete the following steps to initialize a FlexChart control in C# or XAML.

**In Code**

1. Add a new class (for example QuickStart.cs) to your Portable or Shared project and include the following references:

**C#**

```
using Xamarin.Forms;
using Cl.Xamarin.Forms.Chart;
```

2. Instantiate a FlexChart control in a new method GetChartControl().

**C#**

```
public static FlexChart GetChartControl()
{
    FlexChart chart = new FlexChart();

    FlexChartDataSource ds = new FlexChartDataSource();
    chart.ItemsSource = ds.Data;
    chart.BindingX = "Name";

    ChartSeries series = new ChartSeries();
    series.SeriesName = "Sales";
    series.Binding = "Sales";
    series.ChartType = ChartType.Column;
    chart.Series.Add(series);

    return chart;
}
```

**In XAML**

1. Add a new Forms XAML Page (for example QuickStart.xaml) to your Portable or Shared project and modify the <ContentPage> tag to include the following references:

**XAML**

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="Test_XAML.QuickStart"
xmlns:cl="clr-namespace:Cl.Xamarin.Forms.Chart;assembly=Cl.Xamarin.Forms.Chart">
```

2. Initialize a FlexChart control by adding the markup for the control between the <ContentPage> </ContentPage> tags and inside the <StackLayout></StackLayout> tags, as shown below.

**XAML**

```
<StackLayout>
    <cl:FlexChart x:Name="chart" ItemsSource="{Binding Data}" BindingX="Name"
    ChartType="Column"
        Grid.Row="1" Grid.ColumnSpan="2" VericalOptions="FillAndExpand">
        <cl:FlexChart.Series>
```

```

        <cl:ChartSeries x:Name="Sales2015" Name ="Sales" Binding="Sales" >
    </cl:ChartSeries>
    </cl:FlexChart.Series>
</cl:FlexChart>
</StackLayout>

```

3. In the **Solution Explorer**, expand the QuickStart.xaml node and open QuickStart.xaml.cs to view the C# code.
4. In the QuickStart() class constructor, set the BindingContext for the FlexChart.

The following code shows what the QuickStart() class constructor looks like after completing this step.

```

C#
public QuickStart()
{
    InitializeComponent();
    chart.BindingContext = new FlexChartDataSource();
}

```

## Back to Top

### Step 3: Run the Project

1. In the **Solution Explorer**, double click App.cs to open it.
2. Complete the following steps to display the FlexChart control.
  - **To return a C# class:** In the class constructor App(), set a new ContentPage as the MainPage and assign the control to the ContentPage's Content by invoking the method GetChartControl() defined in the previous procedure, **Step 2: Add a FlexChart Control**.

The following code shows the class constructor App() after completing steps above.

```

C#
public App()
{
    // The root page of your application
    MainPage = new ContentPage
    {
        Content = QuickStart.GetChartControl()
    };
}

```

- **To return a Forms XAML Page:** In the class constructor App(), set the Forms XAML Page QuickStart as the MainPage.

The following code shows the class constructor App(), after completing this step.

```

C#
public App()
{
    // The root page of your application
    MainPage = new QuickStart();
}

```

3. Some additional steps are required to run iOS and UWP apps:
  - **iOS App:**

1. In the **Solution Explorer**, double click `AppDelegate.cs` inside `YourAppName.iOS` project to open it.
2. Add the following code to the `FinishedLaunching()` method.

```
C#
C1.Xamarin.Forms.Chart.Platform.iOS.Forms.Init();
```

o **UWP App:**

1. In the **Solution Explorer**, expand `MainPage.xaml`.
2. Double click `MainPage.xaml.cs` to open it.
3. Add the following code to the class constructor.

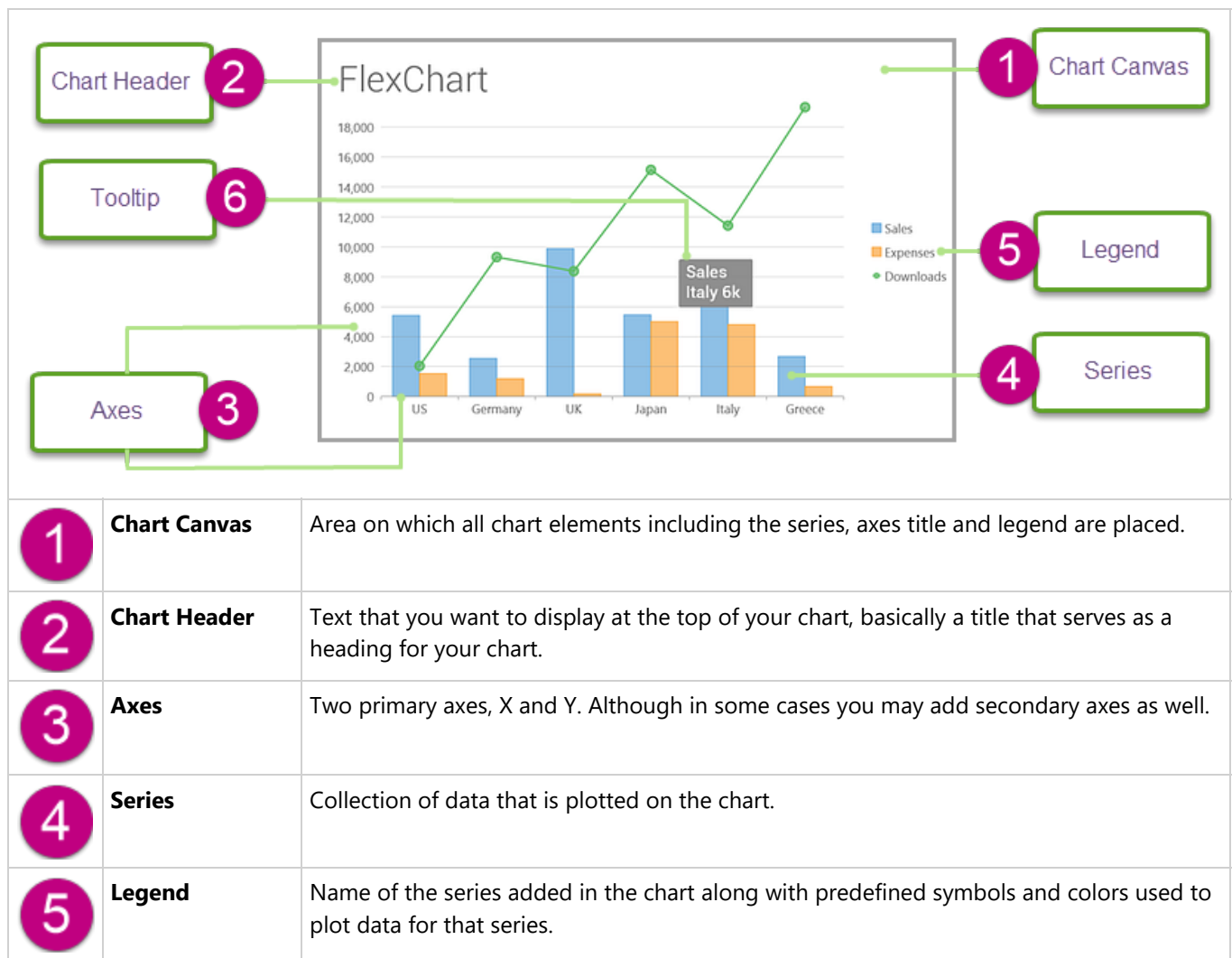
```
C#
C1.Xamarin.Forms.Chart.Platform.UWP.FlexChartRenderer.Init();
```

4. Press **F5** to run the project.

**Back to Top**

## Chart Elements

FlexChart is composed of several elements as shown below:



**Tooltip**

Tooltips or labels that appear when you hover on a series.

## Chart Types

You can change the type of the FlexChart control depending on your requirement. Chart type can be changed by setting the [ChartType](#) property of the FlexChart control. In this case, if multiple series are added to the FlexChart, all of them are of the same chart type. To know how to add multiple series and to set a different ChartType for each series, see [Mixed charts](#). FlexChart supports various chart types including Line and LineSymbol chart, Area chart, Bar and Column chart, Bubble chart, Scatter chart, Candlestick chart, etc.

### In Code

C#

```
chart.ChartType = ChartType.LineSymbols;
```

### In XAML

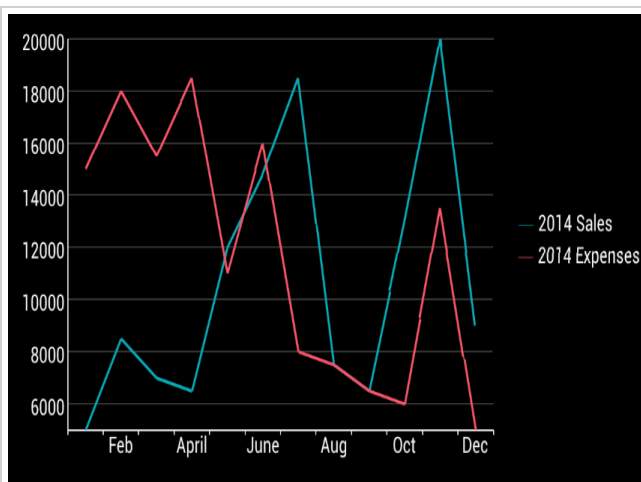
XAML

```
<cl:FlexChart x:Name="chart" ChartType="Bar" ItemsSource="{Binding Data}" BindingX="Name"
  Grid.Row="1" Grid.ColumnSpan="2">
  <cl:FlexChart.Series>
    <cl:ChartSeries x:Name="Sales2014" Binding="Sales" ></cl:ChartSeries>
  </cl:FlexChart.Series>
</cl:FlexChart>
```

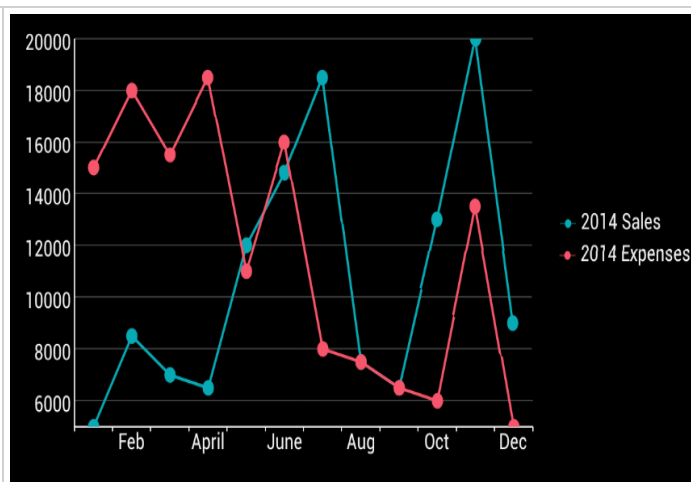
### Line and LineSymbol chart

A Line chart draws each series as connected points of data, similar to area chart except that the area below the connected points is not filled. The series can be drawn independently or stacked. It is the most effective way of denoting changes in value between different groups of data. A LineSymbol chart is similar to line chart except that it represents data points using symbols.

These charts are commonly used to show trends and performance over time.



Line Chart

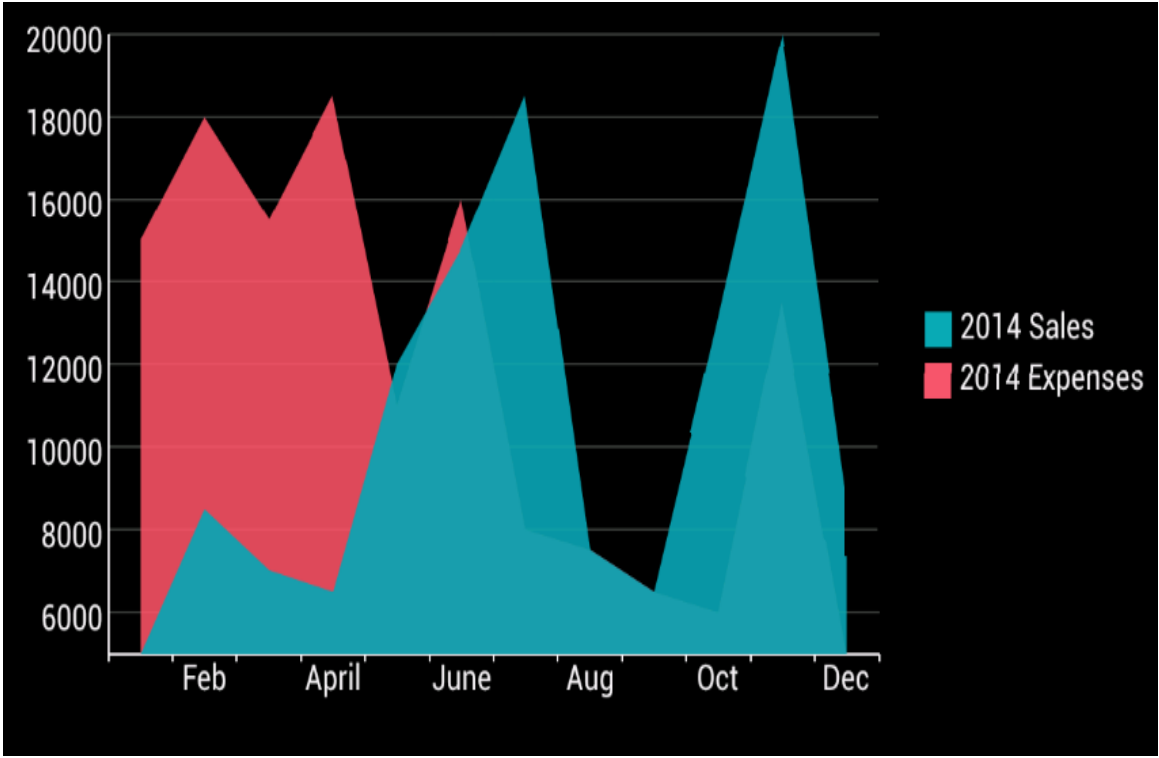


LineSymbol Chart



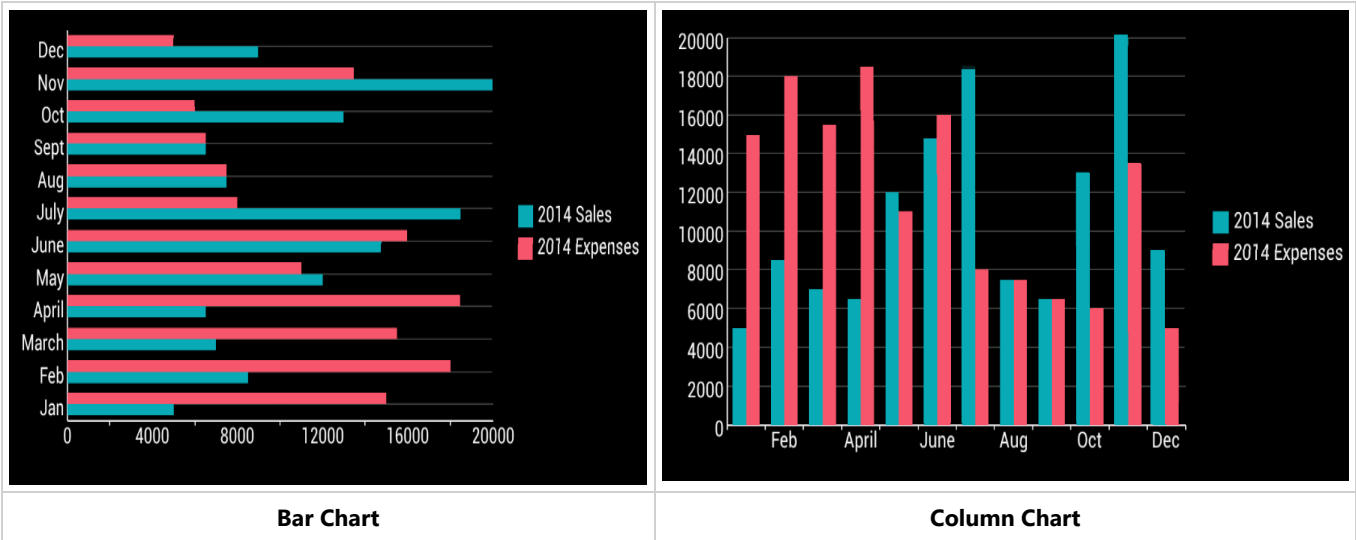
Area chart

An Area chart draws each series as connected points of data and the area below the connected points is filled with color to denote volume. Each new series is drawn on top of the preceding series. The series can either be drawn independently or stacked. These charts are commonly used to show trends between associated attributes over time.



Bar and Column chart

A Bar chart or a Column chart represents each series in the form of bars of the same color and width, whose length is determined by its value. Each new series is plotted in the form of bars next to the bars of the preceding series. When the bars are arranged horizontally, the chart is called a bar chart and when the bars are arranged vertically, the chart is called column chart. Bar charts and Column charts can be either grouped or stacked. These charts are commonly used to visually represent data that is grouped into discrete categories, for example age groups, months, etc.



Bubble chart

A Bubble chart represents three dimensions of data. The X and Y values denote two of the data dimensions. The third dimension is denoted by the size of the bubble.

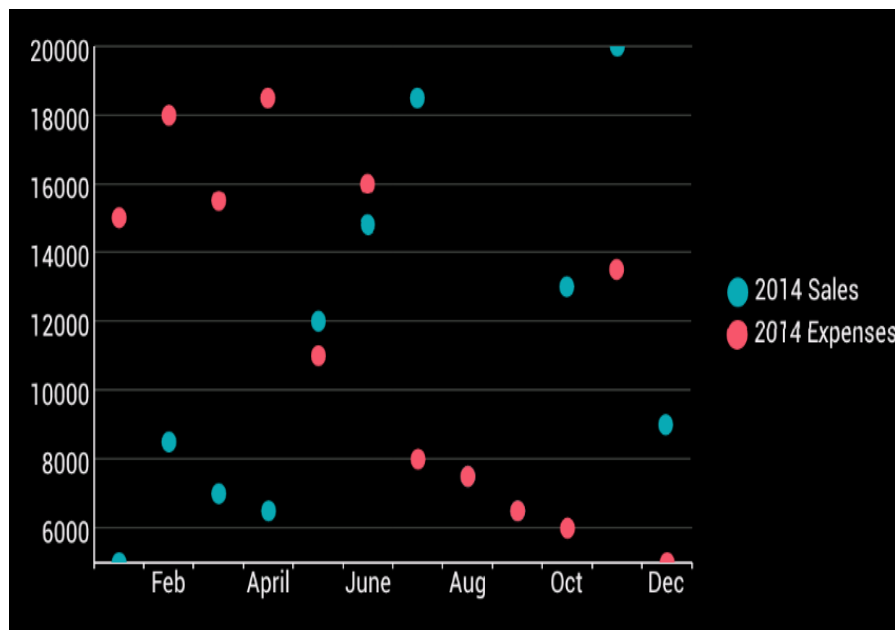
These charts are used to compare entities based on their relative positions on the axis as well as their size.



## Scatter

A Scatter chart represents a series in the form of points plotted using their X and Y axis coordinates. The X and Y axis coordinates are combined into single data points and displayed in uneven intervals or clusters.

These charts are commonly used to determine the variation in data point density with varying x and y coordinates.



## Candlestick chart

A Candlestick chart is a financial chart that shows the opening, closing, high and low prices of a given stock. It is a special type of HiLoOpenClose chart that is used to show the relationship between open and close as well as high and low. Candle chart uses price data (high, low, open, and close values) and it includes a thick candle-like body that uses the color and size of the body to reveal additional information about the relationship between the open and close values. For example, long transparent candles show buying pressure and long filled candles show selling pressure.

### Elements of a Candlestick chart

The Candlestick chart is made up of the following elements: **candle**, **wick**, and **tail**.

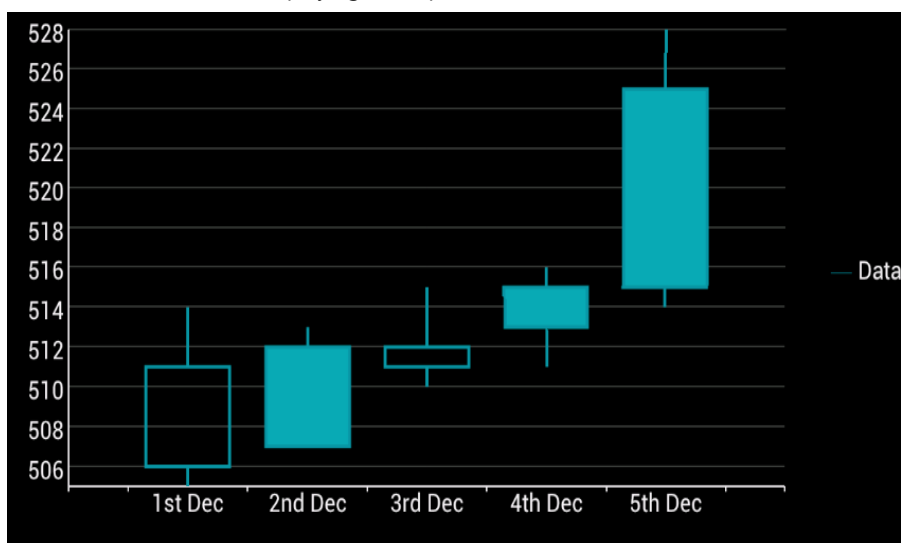
- **Candle:** The candle or the body (the solid bar between the opening and closing values) represents the change in stock price from opening to closing.

- **Wick and Tail:** The thin lines, wick and tail, above and below the candle depict the high/low range.
- **Hollow Body:** A hollow candle or transparent candle indicates a rising stock price (close was higher than open). In a hollow candle, the bottom of the body represents the opening price and the top of the body represents the closing price.
- **Filled Body:** A filled candle indicates a falling stock price (open was higher than close). In a filled candle the top of the body represents the opening price and the bottom of the body represents the closing price.

In a Candlestick there are five values for each data point in the series.

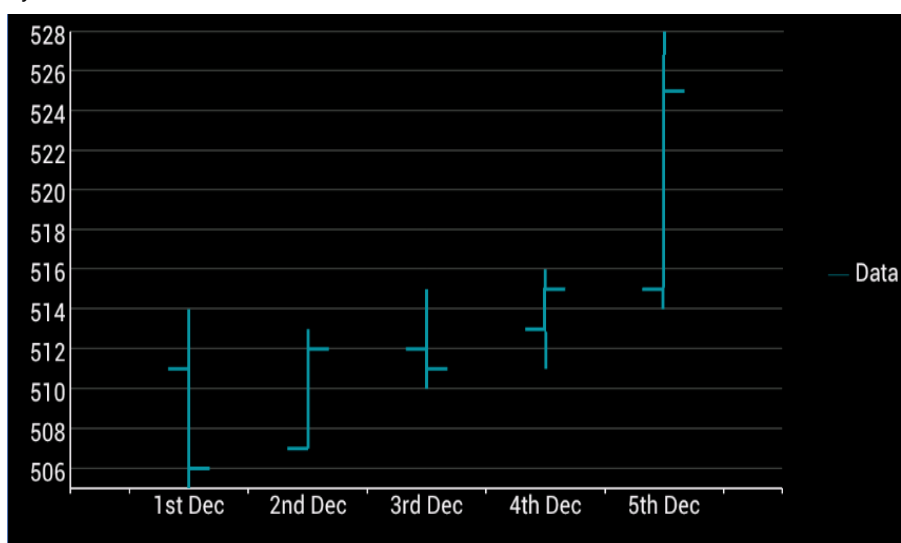
- **x:** Determines the date position along the x axis.
- **high:** Determines the highest price for the day, and plots it as the top of the candle along the y axis.
- **low:** Determines the lowest price for the day, and plots it as the bottom of the candle along the y axis.
- **open:** Determines the opening price for the day.
- **close:** Determines the closing price for the day.

The following image shows a candlestick chart displaying stock prices.



### High Low Open Close chart

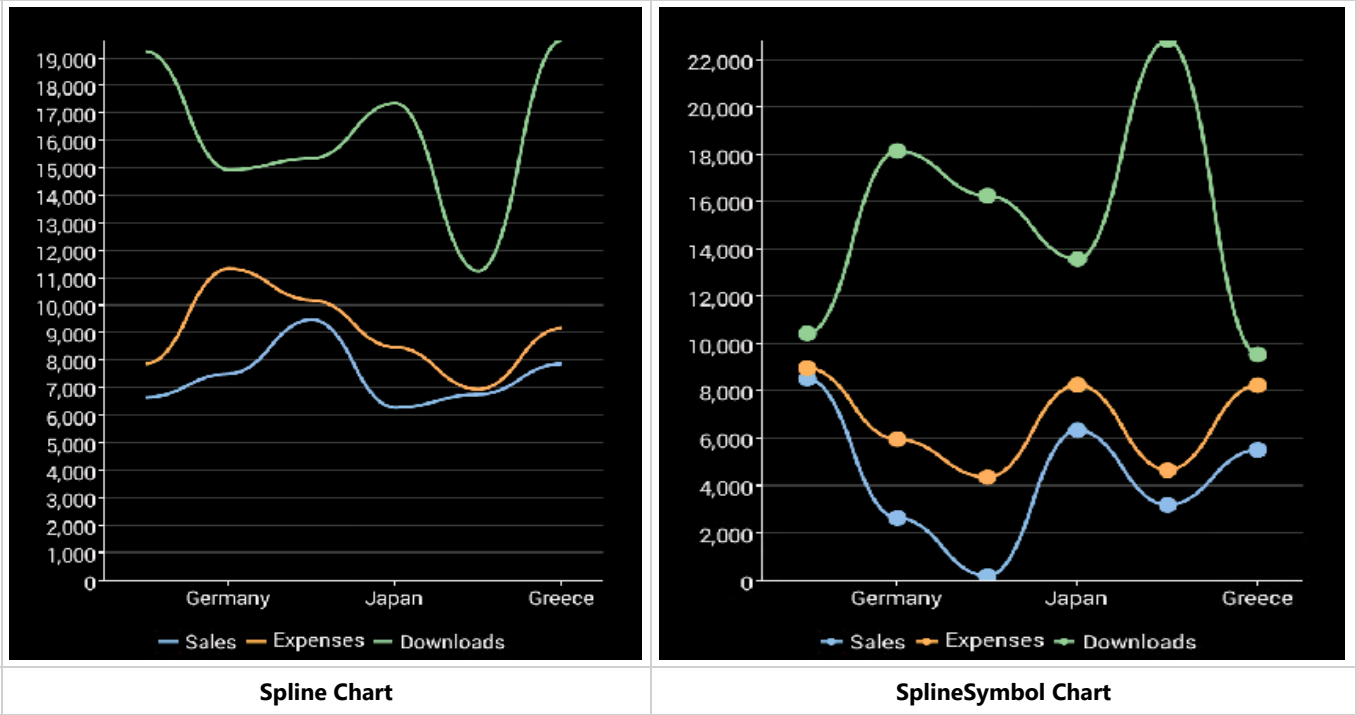
HiLoOpenClose are financial charts that combine four independent values to supply high, low, open and close data for a point in a series. In addition to showing the high and low value of a stock, the Y2 and Y3 array elements represent the stock's opening and closing price respectively.



### Spline and SplineSymbol chart

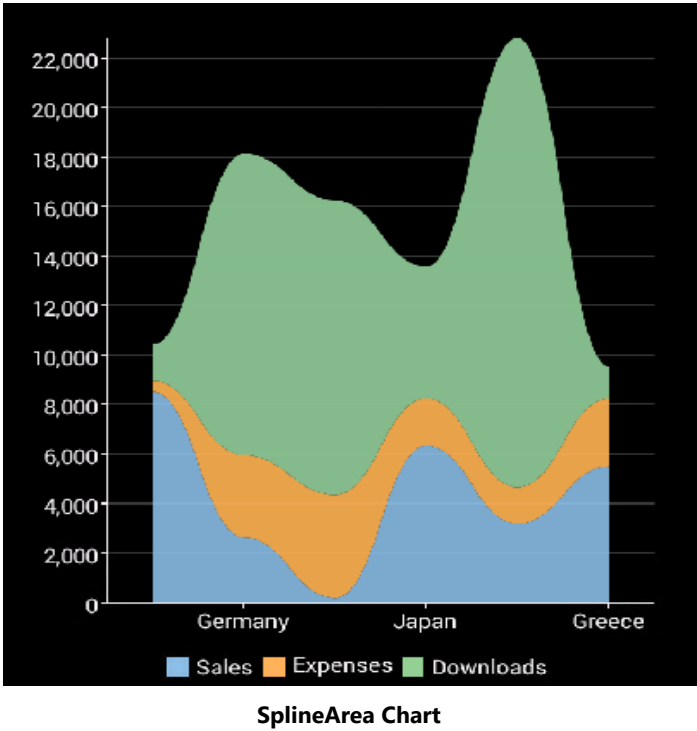
A Spline chart is a combination of line and area charts. It draws a fitted curve through each data point and its series can be drawn independently or stacked. It is the most effective way of representing data that uses curve fittings to show difference of values. A

SplineSymbol chart is similar to Spline chart except that it represents data points using symbols. These charts are commonly used to show trends and performance over time, such as product life-cycle.



SplineArea chart

SplineArea charts are spline charts that display the area below the spline filled with color. SplineArea chart is similar to Area chart as both the charts show area, except that SplineArea chart uses splines and Area chart uses lines to connect data points.




Features

## Axes

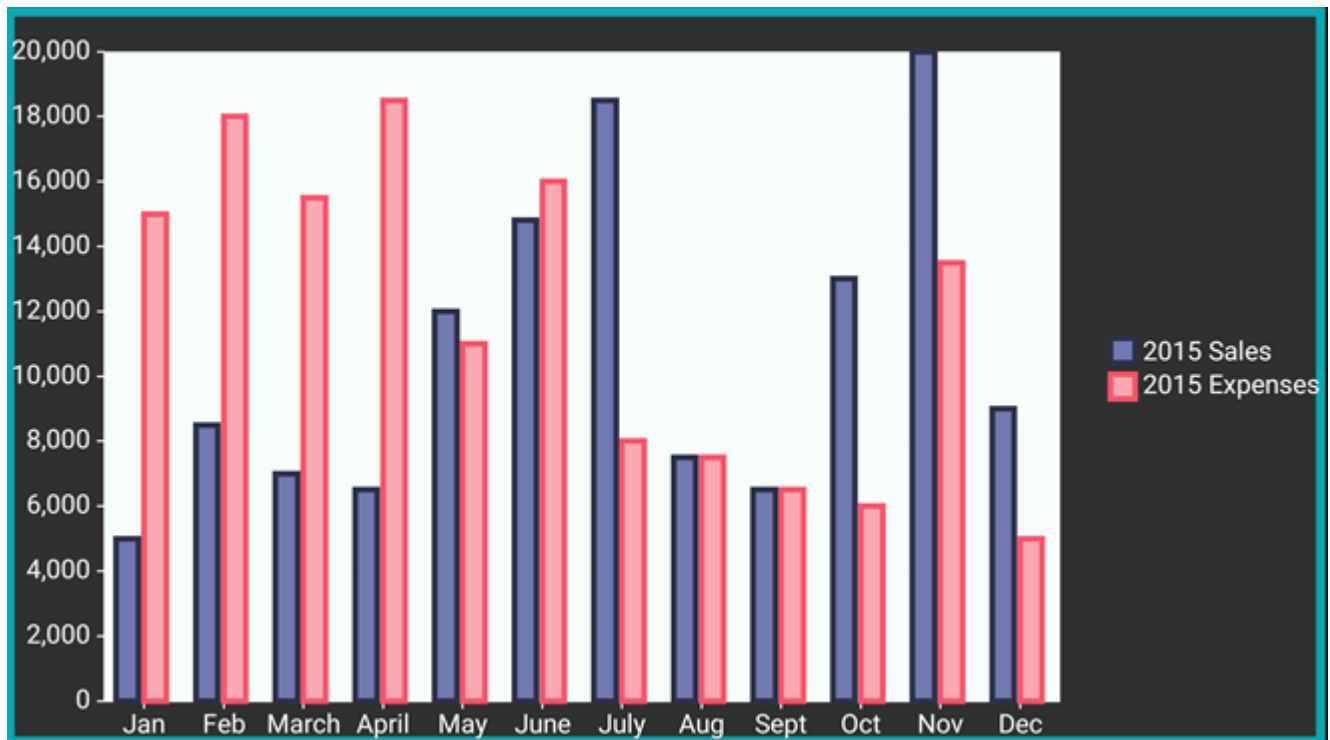
An axis is composed of several elements, such as formats, lines, tick marks and titles. There are several properties available in FlexChart that let you customize these elements, for both X and Y axes. Some of the important properties are listed below.

- **Format** - Lets you select the format string used for axis labels.
- **AxisLine** - Let you display axis lines.
- **MajorUnit** - Lets you set the major unit of the axis.
- **Origin** - Lets you set the origin.
- **Title** - Lets you add a title to the axis.

See [https://msdn.microsoft.com/ja-jp/library/dwhawy9k\(v=vs.110\).aspx](https://msdn.microsoft.com/ja-jp/library/dwhawy9k(v=vs.110).aspx) for information on standard format strings available in .Net.

 Axis line for Y axis and grid lines on the X axis are disabled by default. To enable the axis lines and grid lines, set the [AxisLine](#) and [MajorGrid](#) properties to true.

The image below shows a FlexChart with customized axes.



The following code examples demonstrate how to customize the axes in C# and XAML. These examples use the sample created in the [Customize Appearance](#) section.

### In Code

C#

```
//Customizing X-axis
chart.AxisY.AxisLine = true;
chart.AxisY.MajorGrid = true;
chart.AxisY.Title = "Sales and Expenses (in Millions)";
chart.AxisY.MajorGrid = true;
```

```
chart.AxisY.MajorUnit = 2000;
chart.AxisY.Format = "D";

//Customizing Y-axis
chart.AxisX.AxisLine = true;
chart.AxisX.MajorGrid = true;
chart.AxisX.Title = "Month";
```

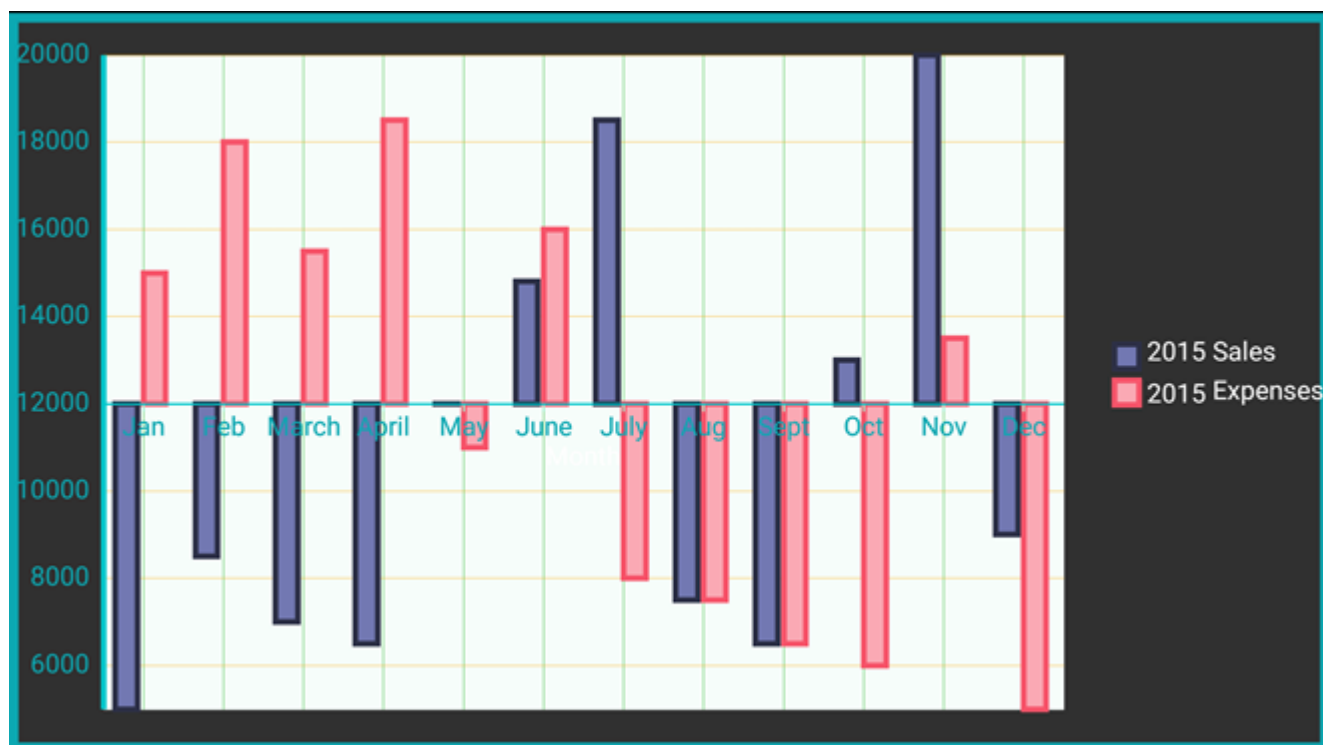
## In XAML

### XAML

```
<cl:FlexChart x:Name="chart" ChartType="Column" ItemsSource="{Binding Data}"
BindingX="Name"
PlotAreaBackgroundColor="#F6FDFA" >
    <cl:FlexChart.Series>
        <cl:ChartSeries x:Name="Sales2015" Name="2015 Sales" Binding="Sales">
            <cl:ChartSeries.Style>
                <cl:ChartStyle Fill="#7278B2" Stroke="#2D3047"
StrokeThickness="3" />
            </cl:ChartSeries.Style>
        </cl:ChartSeries>
        <cl:ChartSeries x:Name="Expenses2015" Name="2015 Expenses" Binding="Expenses">
            <cl:ChartSeries.Style>
                <cl:ChartStyle Fill="#FAA9B4" Stroke="#F6546A"
StrokeThickness="3" />
            </cl:ChartSeries.Style>
        </cl:ChartSeries>
    </cl:FlexChart.Series>
    <cl:FlexChart.AxisY >
        <cl:ChartAxis MajorGrid="true" AxisLine="true" Title="Sales and Expenses (in
Millions)"
        MajorUnit = "2000" Format="D">
        </cl:ChartAxis>
    </cl:FlexChart.AxisY>
    <cl:FlexChart.AxisX>
        <cl:ChartAxis AxisLine="true" Title= "Month" MajorGrid="true">
        </cl:ChartAxis>
    </cl:FlexChart.AxisX>
</cl:FlexChart>
```

## Customizing Axis Origin

The FlexChart control allows users to customize the origin for plotting data points in two quadrants. You can use [Origin](#) property of the [ChartAxis](#) class to set the origin for both the axes. The following image shows the origin of X-axis set to 12,000.



### In Code

The following code example illustrates how to customize the X-axis origin for FlexChart control. This example uses the sample created for [Quick Start](#) section. You can also set Y-axis origin in code in a way similar to that given in the following code.

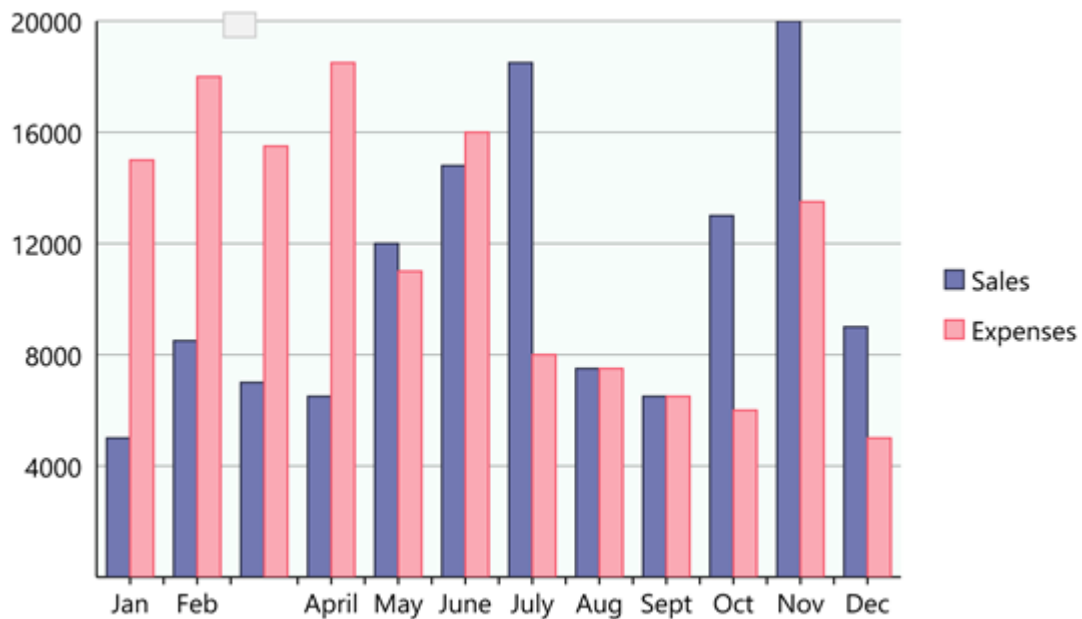
C#

```
chart.AxisX.Origin = 12000;
```

## Customize Appearance

Although, Xamarin controls match the native controls on all three platforms by default and are designed to work with both: light and dark themes available on all platforms. But, there are several properties to customize the appearance of the FlexChart control. You can change the background color of the chart plot area, set the color of the series, add colored borders of specified thickness to charts as well as series and do much more to enhance the appearance of the control.

The image below shows a customized FlexChart control.



### In Code

The following code example demonstrates how to customize FlexChart and its series. This examples uses the sample created in the [Quick Start](#) section, with multiple series added to the chart. See [Mixed Charts](#) to know how to add multiple series to a FlexChart.

C#

```
//Customize chart series
ChartSeries series = new ChartSeries();
series.SeriesName = "Sales";
series.Binding = "Sales";
series.ChartType = ChartType.Column;
series.Style.Fill = Color.FromHex("#7278B2");
series.Style.Stroke = Color.FromHex("#2D3047");
chart.Series.Add(series);

ChartSeries series1 = new ChartSeries();
series1.SeriesName = "Expenses";
series1.Binding = "Expenses";
series1.ChartType = ChartType.Column;
series1.Style.Fill = Color.FromHex("#FAA9B4");
series1.Style.Stroke = Color.FromHex("#F6546A");
chart.Series.Add(series1);

//Customize chart plot area
ChartStyle s = new ChartStyle();
s.Fill = Color.FromHex("#F6FDFA");
s.StrokeThickness = 0;
chart.PlotStyle = s;
```

## Data Binding

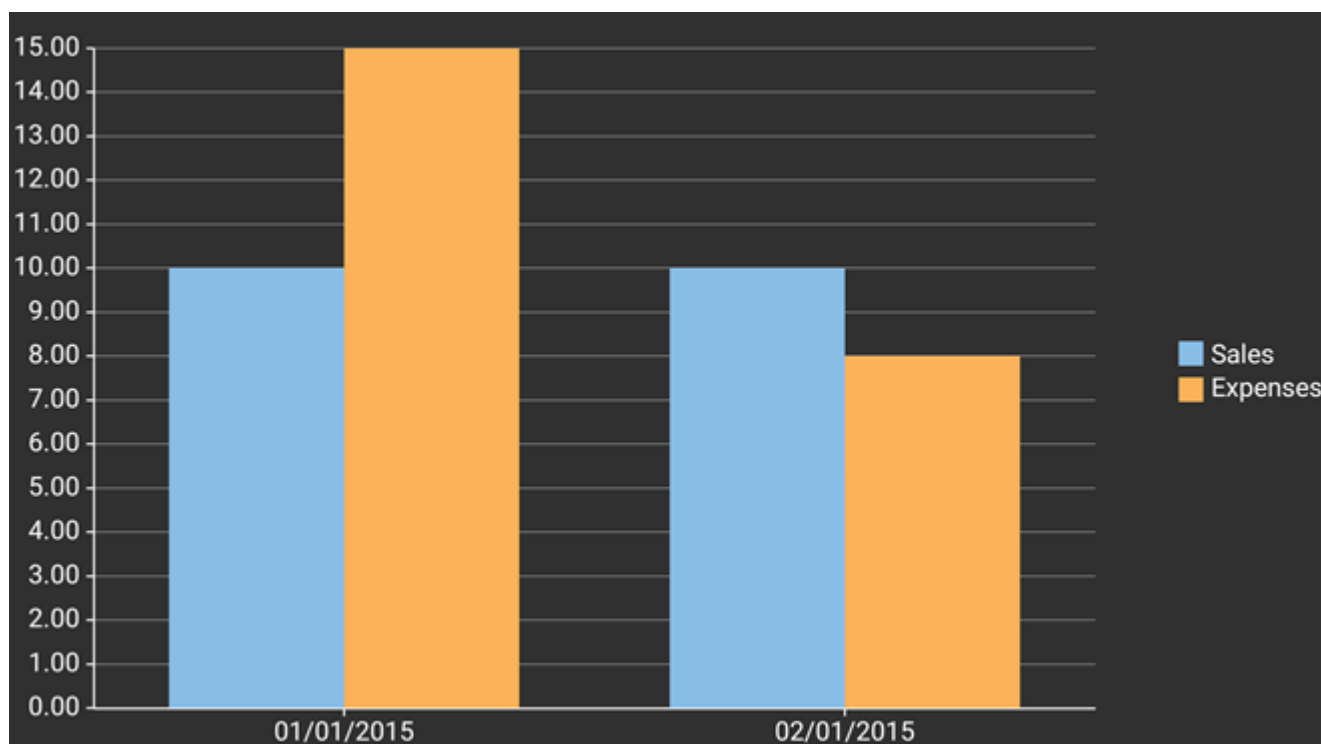


You can bind the FlexChart control to data by setting its [BindingX](#) property (for X-axis values), and [Binding](#) property (for Y-axis values) on each Series that you add to the control. The FlexChart control and the [ChartSeries](#) class provides the same set of properties for data binding as mentioned in the following table.

Property Name	Description
<a href="#">Binding</a>	Property for binding Y-axis values
<a href="#">BindingX</a>	Property for binding X-axis values
<a href="#">ItemsSource</a>	Property for binding with collection of items

Users can set the ItemsSource and BindingX properties on the FlexChart control, and the Binding property on each series. The ChartSeries uses parent chart values in case these properties are not specified on the series level.

The image given below shows Data Binding in the FlexChart control. The X-axis is bound to DateTime while the Y-axis is bound to Sales and Expenses values for the corresponding DateTime.



To implement and observe this feature, we use a new data source file and make some changes in the QuickStart class that we used earlier.

### In Code

The following code examples illustrate how to set Data Binding in FlexChart control. The example uses the sample created in the Quick Start section with slight changes as described below.

1. Replace the code in the data source file, that is FlexChartDataSource, with the following code.

```
C#  
  
public class FlexChartDataSource  
{  
    public string Name { get; set; }  
    public double Sales { get; set; }  
    public double Expenses { get; set; }  
}
```

```

public double Downloads { get; set; }
public DateTime Date { get; set; }
public FlexChartDataSource()
{
    this.Name = string.Empty;
    this.Sales = 0;
    this.Expenses = 0;
    this.Downloads = 0;
    this.Date = DateTime.Now;
}
public FlexChartDataSource(string name, double sales, double expenses, double
downloads, DateTime date)
{
    this.Name = name;
    this.Sales = sales;
    this.Expenses = expenses;
    this.Downloads = downloads;
    this.Date = date;
}
}

```

2. Set the `BindingX` to "Time" within `GetChartControl ()` method in `QuickStart` class file as illustrated in the following code.

C#

```

chart.BindingX = "Time";
chart.Series.Add(new ChartSeries() { Binding = "Sales", Name = "Sales" });
chart.Series.Add(new ChartSeries() { Binding = "Expenses", Name = "Expenses" });
chart.ItemsSource = new object[]
{
    new {Time=new DateTime(2015,1,1), Sales=10, Expenses = 15},
    new {Time=new DateTime(2015,2,1), Sales=10, Expenses=8}
};

```

## In XAML

You can also set the Binding through XAML as mentioned in the following code snippet.

XAML

```

<cl:FlexChart x:Name="chart" ItemsSource="{Binding FlexChartDataSource}"
BindingX="Time">
    <cl:FlexChart.Series>
        <cl:ChartSeries Binding="Sales" Name="Sales" />
        <cl:ChartSeries Binding="Expenses" Name="Expenses" />
    </cl:FlexChart.Series>
</cl:FlexChart>

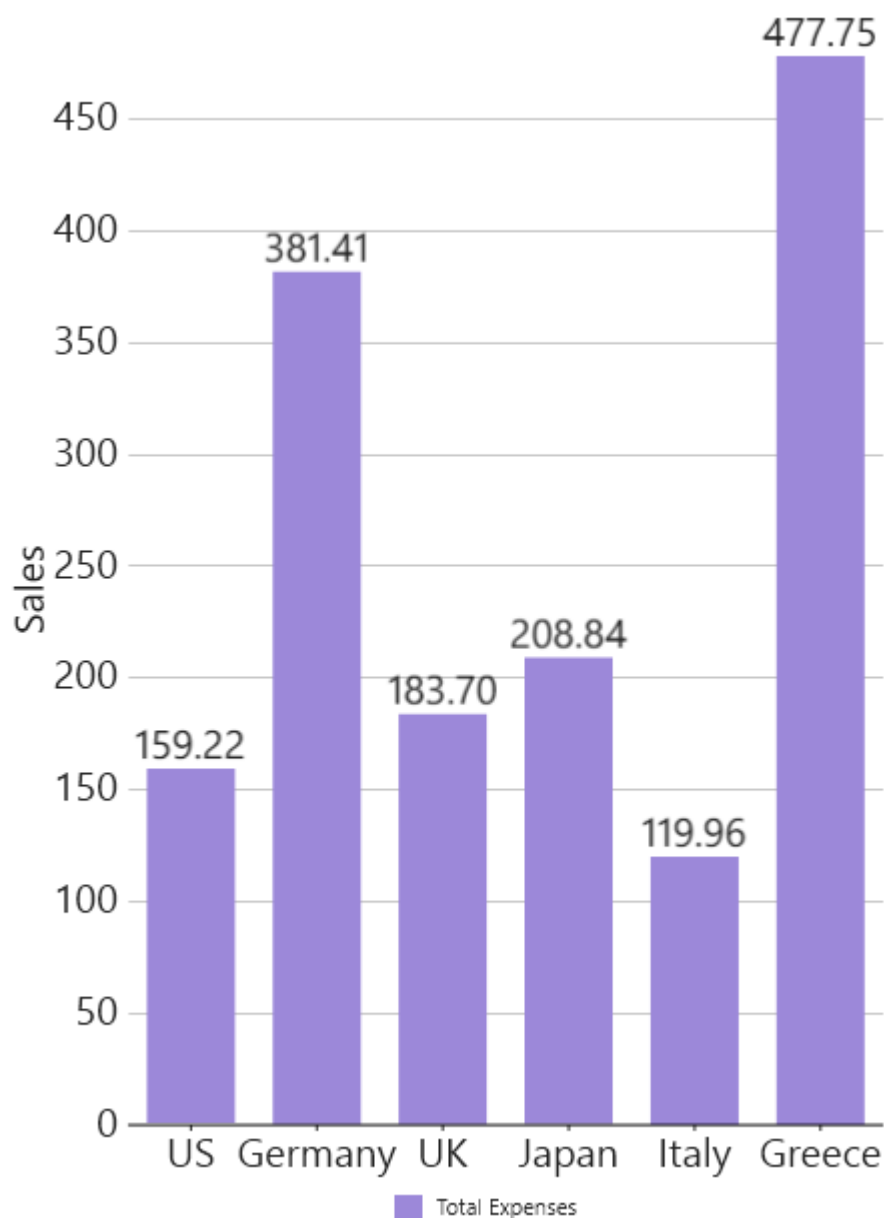
```

## Data Labels

You can add data labels in the `FlexChart` control to show the exact values corresponding to a particular column, bar or point in the plot area. You can display the data labels at the following positions relative to each plot element.

- **Top** - on the top of the column, bar or point in the chart area.
- **Bottom** - below the edge of the column, bar or point in the chart area.
- **Right** - to the right of the column, bar or point in the chart area.
- **Left** - to the left of the column, bar or point in the chart area.
- **Center** - in the center of the column, bar or point in the chart area.
- **None** - no labels are visible.

The following image shows a column chart with data labels positioned on the top.



A data label can contain any amount of text and other UI elements. To display data labels in a FlexChart control, you must set the [ChartDataLabel.Position](#) property and define the content template to display as the label.

The table below lists the predefined parameters applicable for data label content customization.

Parameter	Description
x	Shows the X value of the data point.
y	Shows the Y value of the data point.

value	Shows the Y value of the data point.
name	Shows the X value of the data point.
seriesName	Shows the name of the series.
pointIndex	Shows the index of the data point.
Percentage	Shows the percentage of a pie slice to the whole pie chart.

### In XAML

To set the position and define the content template in XAML, add the following markup between the FlexChart control tags.

#### XAML

```
<cl:FlexChart.DataLabel>
    <cl:ChartDataLabel Position="Top" Border="True" Content="{{value:F2}}"/>
</cl:FlexChart.DataLabel>
```

### In Code

To change the label position in code, add the following code.

#### C#

```
//Setting the data labels
chart.DataLabel.Position = ChartLabelPosition.Top;
```

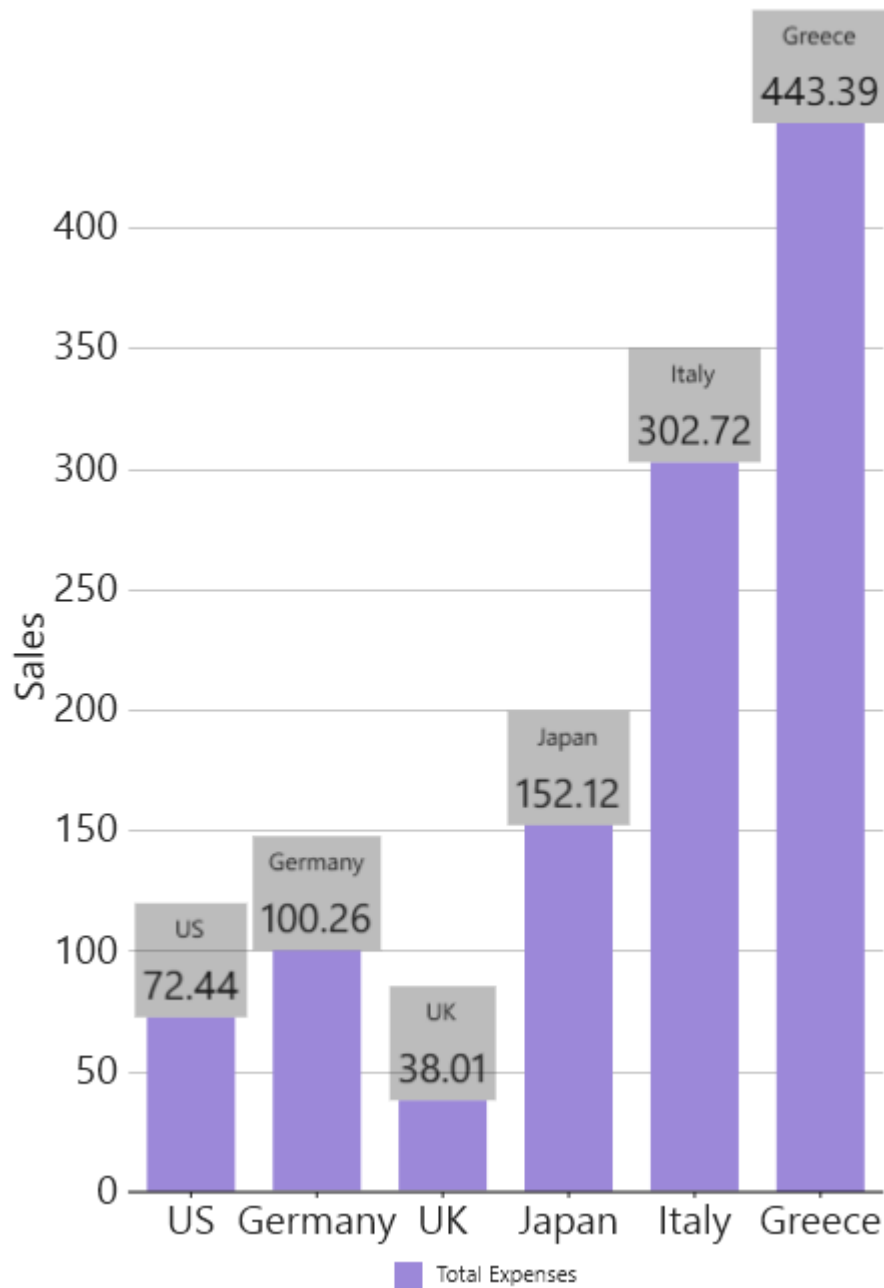
To display the X and Y values in a single label you would add two Labels to the data label template as shown below.

### In XAML

#### XAML

```
<cl:FlexChart.DataLabel>
    <cl:ChartDataLabel Border="True" Position="Top"
Content="{{x}}{value:F2}}">
        <cl:ChartDataLabel.Style>
            <cl:ChartStyle FontSize="14" Fill="LightGray"/></cl:ChartStyle>
        </cl:ChartDataLabel.Style>
    </cl:ChartDataLabel>
</cl:FlexChart.DataLabel>
```

The image below show how the X and Y values appear in a single label.

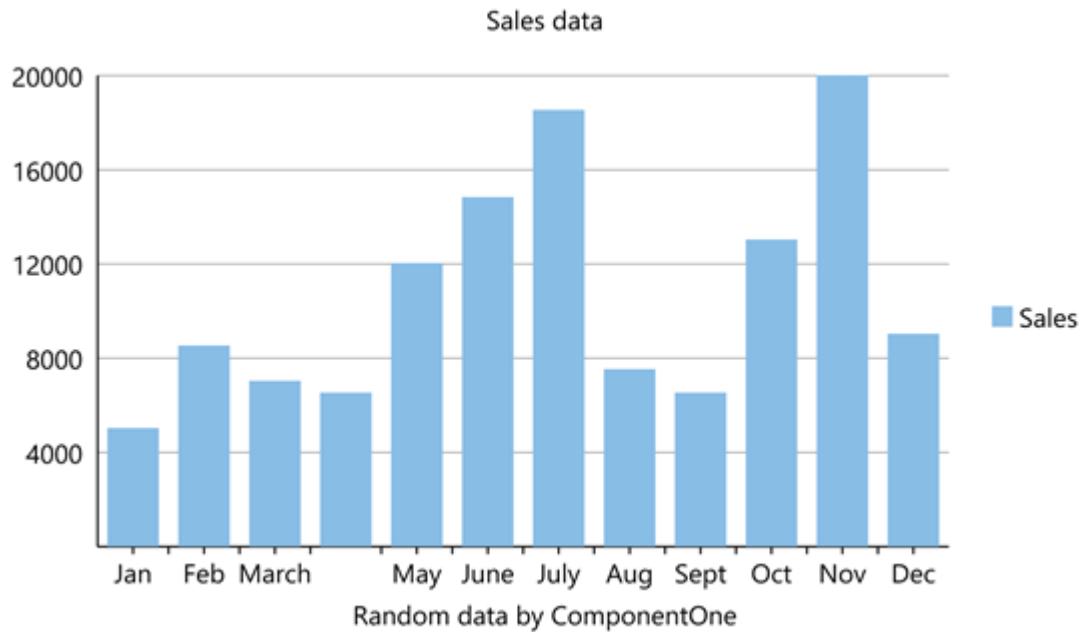


## Header Footer

You can add a title to the FlexChart control by setting its [Header](#) property. Besides a title, you may also set a footer for the chart by setting the [Footer](#) property. There are also some additional properties to customize header and footer text in a FlexChart.

- [HeaderAlignment](#) - Lets you set the alignment for header text.
- [FooterAlignment](#) - Lets you set the alignment for footer text.

The image below shows how the FlexChart appears, after these properties have been set.



### In Code

The following code example demonstrates how to set header and footer properties in C#. This example uses the sample created in the [Quick Start](#) section.

```
C#
//Set header and footer
chart.Header = "Sales data";
chart.HeaderAlignment = LayoutAlignment.Fill;

chart.Footer = "Random data by ComponentOne";
chart.FooterAlignment = LayoutAlignment.Fill;
```

## Hit Test

The [HitTest](#) method is used to determine X and Y coordinates, as well as the index of a point on the FlexChart where the user taps. This method is helpful in scenarios such as displaying tooltips that lie outside the series of the FlexChart.

The following code examples demonstrate how to define the chart\_Tapped event. This event invokes the **HitTest()** method to retrieve the information of the tapped point in the FlexChart region and displays it in the chart footer.

### In Code

1. In the Solution Explorer, right-click your portable application and select **Add | New Item**.
2. In the New Item dialog, select **Forms Xaml Page** and provide a name to it, **HitTest**.
3. Click **OK** to add the XAML page to your project.
4. In the HitTest.xaml page, replace the existing code with following code.

#### XAML

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
              xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
              xmlns:cl="clr-namespace:Cl.Xamarin.Forms.Chart;"
              assembly="Cl.Xamarin.Forms.Chart"
```

```

        x:Class="QuickstartChart.HitTest"
        Padding="10">
<StackLayout>
    <Grid VerticalOptions="FillAndExpand">
        <cl:FlexChart x:Name="flexChart" Tapped="flexChart_Tapped"
            Header="Trigonometric functions" Footer="Cartesian coordinates"
            BindingX="X" ChartType="LineSymbols" LegendPosition="Bottom" >
            <cl:FlexChart.Series>
                <cl:ChartSeries x:Name="seriesCosX" Binding="Y" SeriesName="cos(x)"
/>
                <cl:ChartSeries x:Name="seriesSinX" Binding="Y" SeriesName="sin(x)"
/>
            </cl:FlexChart.Series>
        </cl:FlexChart>
    </Grid>
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <StackLayout x:Name="stackHitTest" VerticalOptions="Fill">
        <StackLayout Orientation="Horizontal">
            <Label x:Name="lblChartElement" />
            <Label Text="{Binding ChartElement}" />
        </StackLayout>
        <StackLayout x:Name="stackSeries">
            <StackLayout Orientation="Horizontal">
                <Label x:Name="lblSeriesName" />
                <Label Text="{Binding Series.SeriesName}" />
            </StackLayout>
            <StackLayout x:Name="stackData">
                <StackLayout Orientation="Horizontal">
                    <Label x:Name="lblPointIdx" />
                    <Label Text="{Binding PointIndex}" />
                </StackLayout>
                <StackLayout Orientation="Horizontal" >
                    <Label Text="{Binding X, StringFormat='X:{0:F2}}'" />
                    <Label Text="{Binding Y, StringFormat='Y:{0:F2}}'" />
                </StackLayout>
            </StackLayout>
        </StackLayout>
    </StackLayout>-->
</Grid>
</StackLayout>

```

5. Open the HitTest.xaml.cs page from the Solution Explorer and add the following code to implement HitTest functionality at runtime.

C#

```

public partial class HitTest : ContentPage
{
    public HitTest()
    {
        InitializeComponent();
    }
}

```

```
int len = 40;
List<Point> listCosTuple = new List<Point>();
List<Point> listSinTuple = new List<Point>();

for (int i = 0; i < len; i++)
{
    listCosTuple.Add(new Point(i, Math.Cos(0.12 * i)));
    listSinTuple.Add(new Point(i, Math.Sin(0.12 * i)));
}

this.flexChart.AxisY.Format = "n2";

this.seriesCosX.ItemsSource = listCosTuple;
this.seriesSinX.ItemsSource = listSinTuple;
}

void flexChart_Tapped(object sender,
Cl.Xamarin.Forms.Core.ClTappedEventArgs e)
{
    var hitTest = this.flexChart.HitTest(e.HitPoint);

    this.stackHitTest.BindingContext = hitTest;
    this.stackData.BindingContext = hitTest;

    this.stackSeries.IsVisible = hitTest != null && hitTest.Series !=
null;
    this.stackData.IsVisible = hitTest != null && hitTest.PointIndex !=
-1;
}
}
```

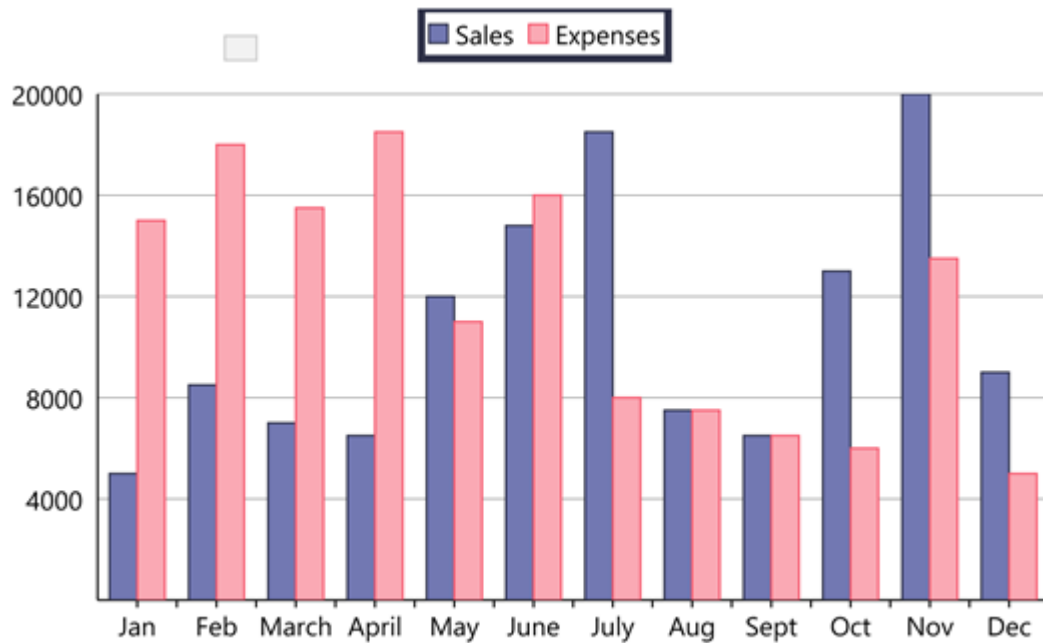
## Legend

FlexChart provides the option to display legend for denoting the type of data plotted on the axes. The position of legend is by default set to "Auto", which means the legend positions itself automatically depending on the real estate available on the device. This allows the chart to efficiently occupy the available space on the device.

You can select where and how to display the legend by setting the [LegendPosition](#) property of the legend. You can also style the legend by setting its orientation through the [LegendOrientation](#) property, and adding a border through the [Stroke](#) property. You can also toggle the visibility of any series on clicking the corresponding legend item by setting the [LegendToggle](#) property to true.

The image below shows how the FlexChart appears after these properties have been set.





- To hide the legend, set the `LegendPosition` property to **None**.
- The legend automatically wraps when `LegendPosition` property is set to **Top** or **Bottom**, orientation is set to **Horizontal** and there is not enough screen real estate.

### In Code

The following code example demonstrates how to set these properties in C#. This example uses the sample created in the [Customize Appearance](#) section.

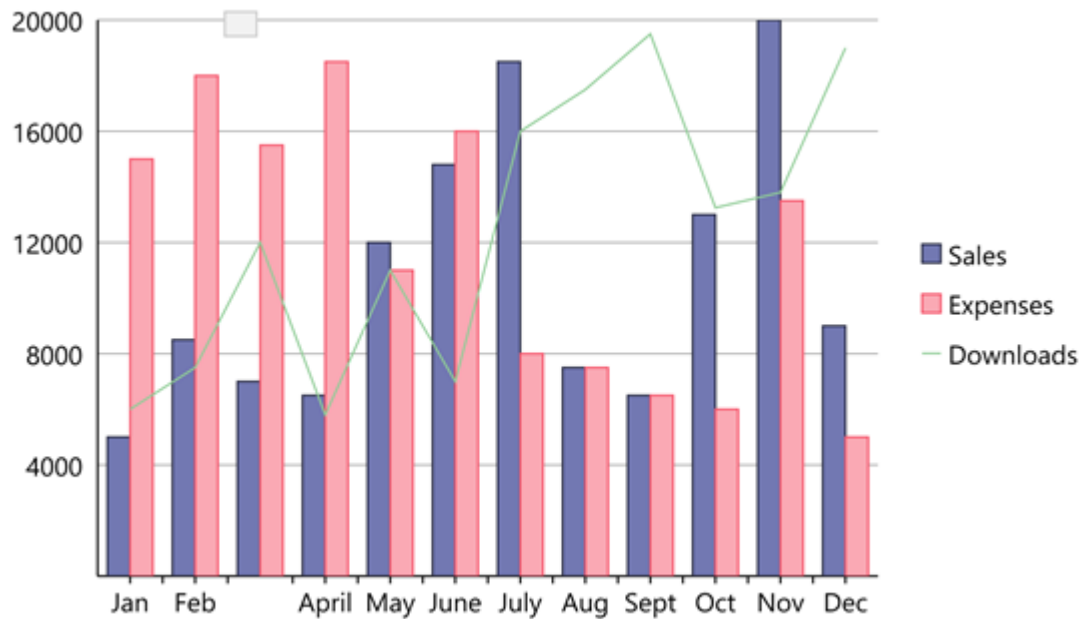
C#

```
chart.LegendToggle = true;  
chart.LegendPosition = ChartPositionType.Top;  
chart.LegendOrientation = Orientation.Horizontal;  
chart.LegendStyle.Stroke = Color.FromHex("#2D3047");  
chart.LegendStyle.StrokeThickness = 4;
```

## Mixed Charts

You can add multiple series to your charts and set a different `ChartType` for each series. Such charts are helpful in analyzing complex chart data on a single canvas. The same data can be used with different visualizations or related data can be displayed together to convey trends.

The following image shows a `FlexChart` with multiple series.



### In Code

The following code example demonstrates how to create multiple instances of type `ChartSeries` with different `ChartTypes` and add them to the `FlexChart` control.

C#

```
ChartSeries series = new ChartSeries();
series.SeriesName = "Sales";
series.Binding = "Sales";
series.ChartType = ChartType.Column;
series.Style.Fill = Color.FromHex("#7278B2");
series.Style.Stroke = Color.FromHex("#2D3047");
chart.Series.Add(series);

ChartSeries series1 = new ChartSeries();
series1.SeriesName = "Expenses";
series1.Binding = "Expenses";
series1.ChartType = ChartType.Column;
series1.Style.Fill = Color.FromHex("#FAA9B4");
series1.Style.Stroke = Color.FromHex("#F6546A");
chart.Series.Add(series1);

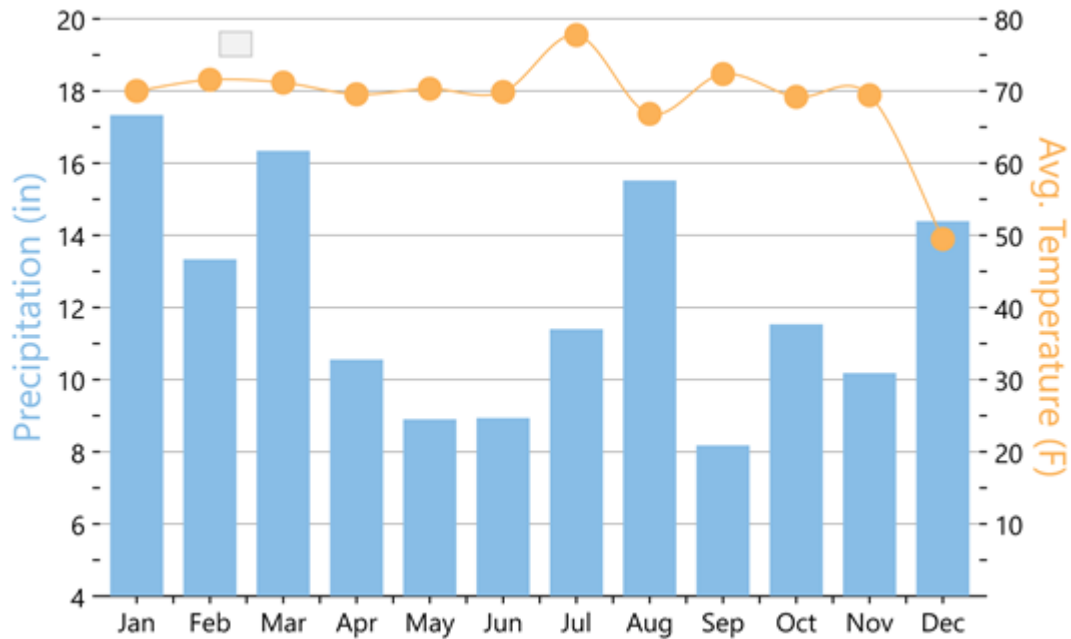
ChartSeries series2 = new ChartSeries();
series2.SeriesName = "Downloads";
series2.Binding = "Downloads";
series2.ChartType = ChartType.Line;
chart.Series.Add(series2);
```

## Multiple Y Axes

You can add multiple axes to the `FlexChart` control and customize its properties. This feature allows you to add two series in a single `FlexChart` instance with same X axis but different Y axis. In order to further customize the appearance,

users can choose to differentiate the two series by selecting different chart types.

The image below shows a FlexChart control with a common X axis that displays months, and left and right Y axes that display two different series for Temperature and Precipitation, respectively.



The following steps demonstrate how to add multiple Y axes in the FlexChart control.

1. In the Solution Explorer, right-click your portable application and select **Add | New Item**. The New Item dialog appears.
2. In the New Item dialog, select Forms XAML Page and provide a name to it, lets say **MultipleYAxes**.
3. Click **OK** to add the XAML page to your project.
4. In the XAML page, replace the existing code with the following to create a FlexChart control that displays two series elements, namely **Precipitation** and **Temperature**.

#### XAML

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
              xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
              xmlns:cl="clr-
namespace:Cl.Xamarin.Forms.Chart;assembly=Cl.Xamarin.Forms.Chart"
              x:Class="QuickstartChart.MultipleYAxes">
    <cl:FlexChart x:Name="flexChart" BindingX="MonthName">
        <cl:FlexChart.Series>
            <cl:ChartSeries Binding="Precipitation" SeriesName="Precip"/>
            <cl:ChartSeries Binding="Temp" ChartType="SplineSymbols" SeriesName="Avg.
Temp">
                <cl:ChartSeries.AxisY>
                    <cl:ChartAxis Position="Right" Min="0" Max="80" MajorUnit="10"
Title="Avg. Temperature (F)" AxisLine="False" MajorGrid="False">
                        <cl:ChartAxis.TitleStyle>
                            <cl:ChartStyle Stroke="#fbb258" FontSize="20"/>
                        </cl:ChartAxis.TitleStyle>
                    </cl:ChartAxis>
                </cl:ChartSeries.AxisY>
            </cl:ChartSeries>
        </cl:FlexChart.Series>
    </cl:FlexChart>
</ContentPage>
```

```

        <cl:FlexChart.AxisY>
            <cl:ChartAxis Min="4" Max="20" MajorUnit="2" MajorGrid="True"
Title="Precipitation (in)" AxisLine="False">
                <cl:ChartAxis.TitleStyle>
                    <cl:ChartStyle Stroke="#88bde6" FontSize="20"/>
                </cl:ChartAxis.TitleStyle>
            </cl:ChartAxis>
        </cl:FlexChart.AxisY>
    </cl:FlexChart>
</ContentPage>

```

5. In the Solution Explorer, open MultipleYAxes.xaml.cs file. Add the following code to it that adds data to be displayed in the FlexChart control at run time.

C#

```

public partial class MultipleYAxes : ContentPage
{
    public MultipleYAxes()
    {
        InitializeComponent();

        this.flexChart.ItemsSource = GetWeatherData();
        this.flexChart.LegendPosition = ChartPositionType.None;
    }

    public IEnumerable<WeatherData> GetWeatherData()
    {
        List<WeatherData> weatherData = new List<WeatherData>();
        string[] monthNames = new string[] { "Jan", "Feb", "Mar", "Apr",
"May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
        //double[] tempData = new double[] { 24, 30, 45, 58, 68, 75, 83, 80,
72, 62, 47, 32 };
        Random random = new Random();

        for (int i = 0; i < monthNames.Length; i++)
        {
            WeatherData wd = new WeatherData();
            wd.MonthName = monthNames[i];
            wd.Precipitation = random.Next(8, 18) + random.NextDouble();
            wd.Temp = Math.Tan(i * i) + 70;
            weatherData.Add(wd);
        }
        return weatherData;
    }
}

public class WeatherData
{
    public string MonthName { get; set; }
    public double Temp { get; set; }
    public double Precipitation { get; set; }
}

```

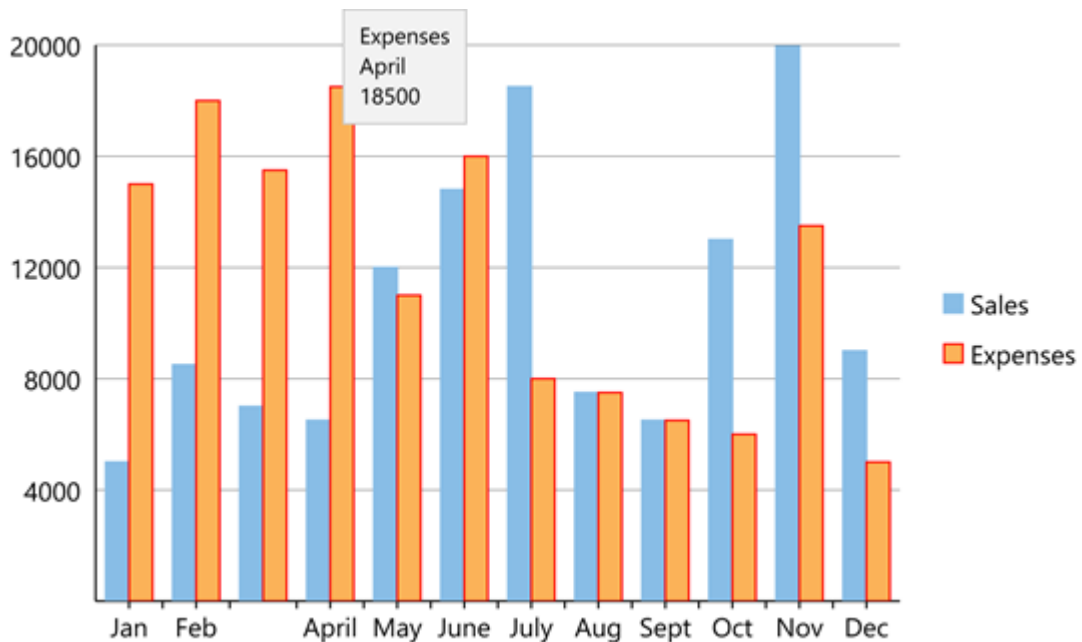
6. Press **F5** to run the project and view two Y axes against a common X axis.

## Selection

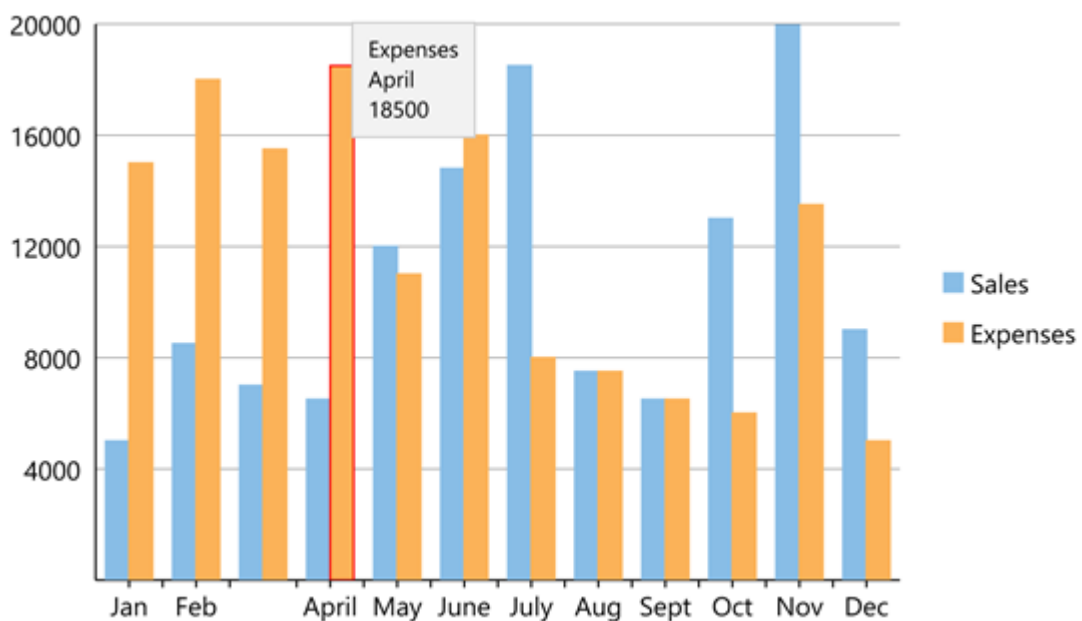
You can choose what element of the FlexChart should be selected when the user taps on any region in a FlexChart by setting the [SelectionMode](#) property. This property provides three options:

- **None** - Does not select any element.
- **Point** - Highlights the point that the user taps.
- **Series** - Highlights the series that the user taps. The user can tap the series on the plot itself, or the series name in the legend.

The images below show how the FlexChart appears after these properties have been set.



When SelectionMode is set to Series



When SelectionMode is set to Point


### In Code

The following code example demonstrates how to set these properties in C#. This example uses the sample created in the [Quick Start](#) section.

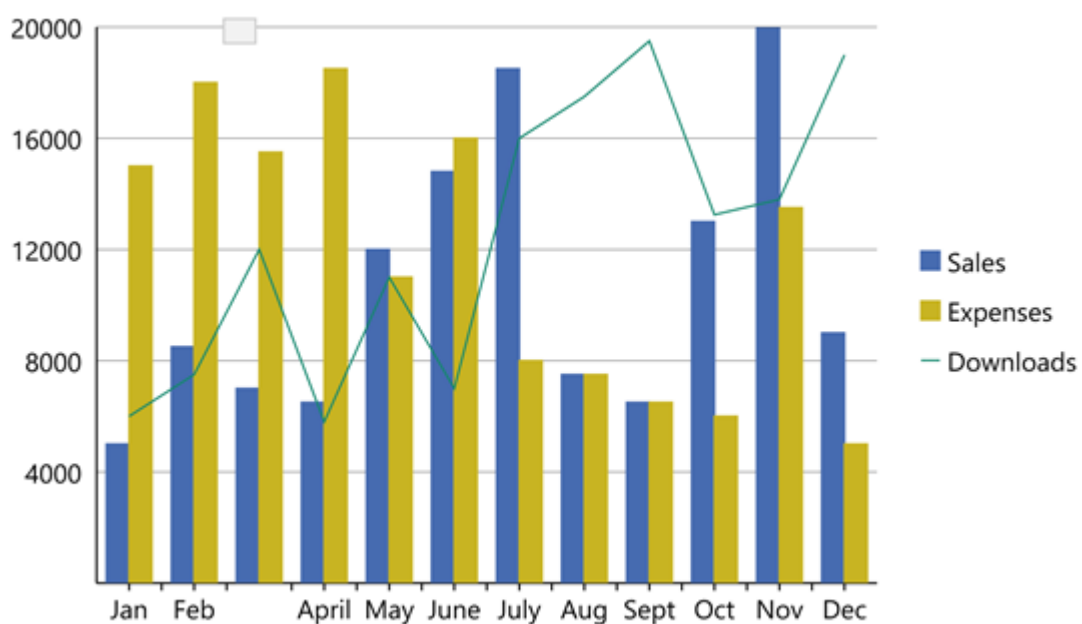
```
C#  
  
//Create series  
ChartSeries series = new ChartSeries();  
series.SeriesName = "Sales";  
series.Binding = "Sales";  
series.ChartType = ChartType.Column;  
chart.Series.Add(series);  
  
ChartSeries series1 = new ChartSeries();  
series1.SeriesName = "Expenses";  
series1.Binding = "Expenses";  
series1.ChartType = ChartType.Column;  
chart.Series.Add(series1);  
  
//Set selection mode  
chart.SelectionMode = ChartSelectionMode.Point;
```

## Themes

An easy way to enhance the appearance of the FlexChart control is to use pre-defined themes instead of customizing each element. The [Palette](#) property is used to specify the theme to be applied on the control.

 **Note:** Remove the Palette property from code to apply default theme.

The image below shows how the FlexChart control appears when the Palette property is set to Cocoa.



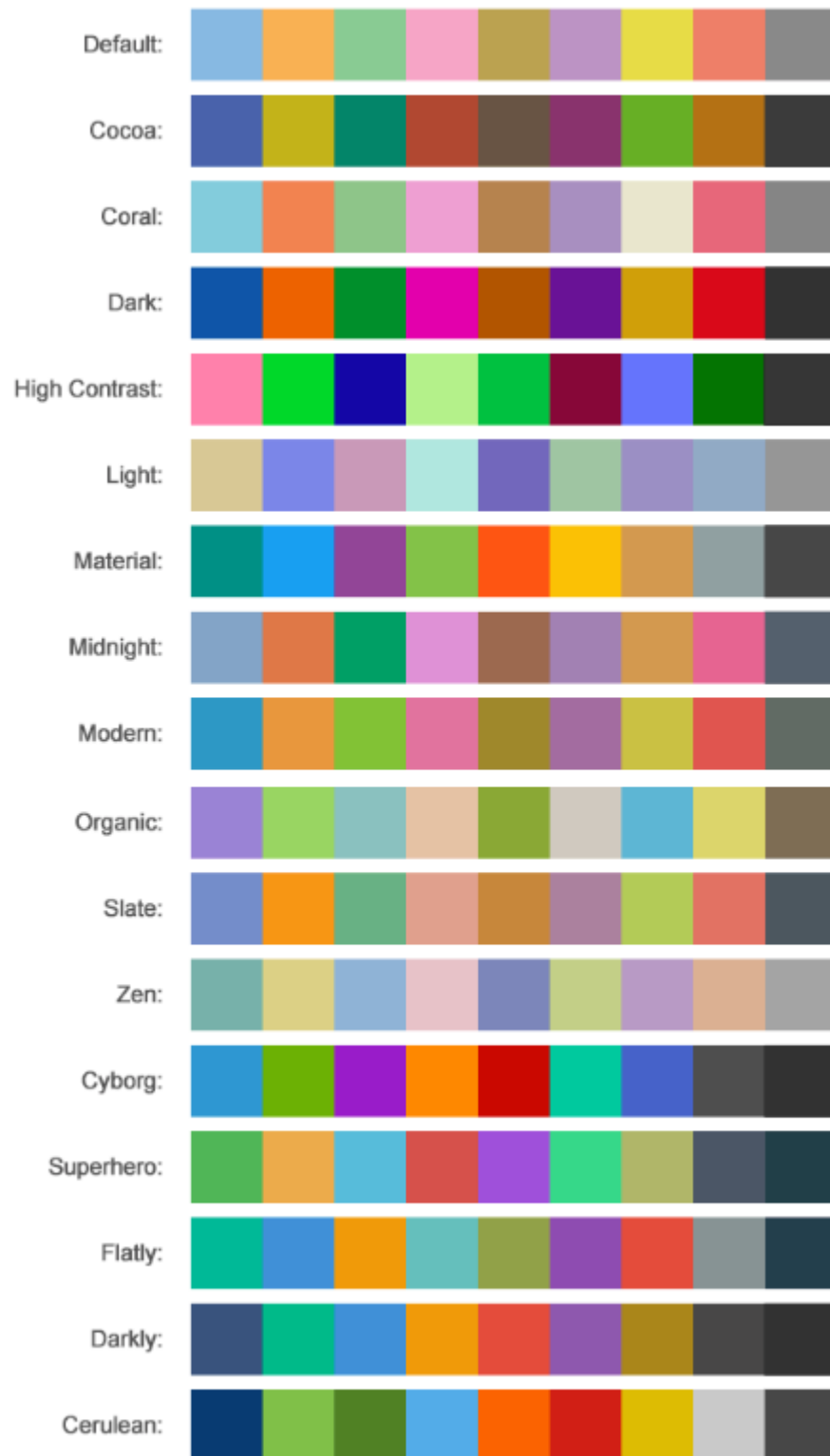
### In Code

The following code example demonstrates how to apply themes in C#.

C#

```
//setting the Palette  
chart.Palette = Palette.Cocoa;
```

FlexChart comes with pre-defined templates that can be applied for quick customization. Here are the 17 pre-defined templates available in the [Palette](#) enumeration.



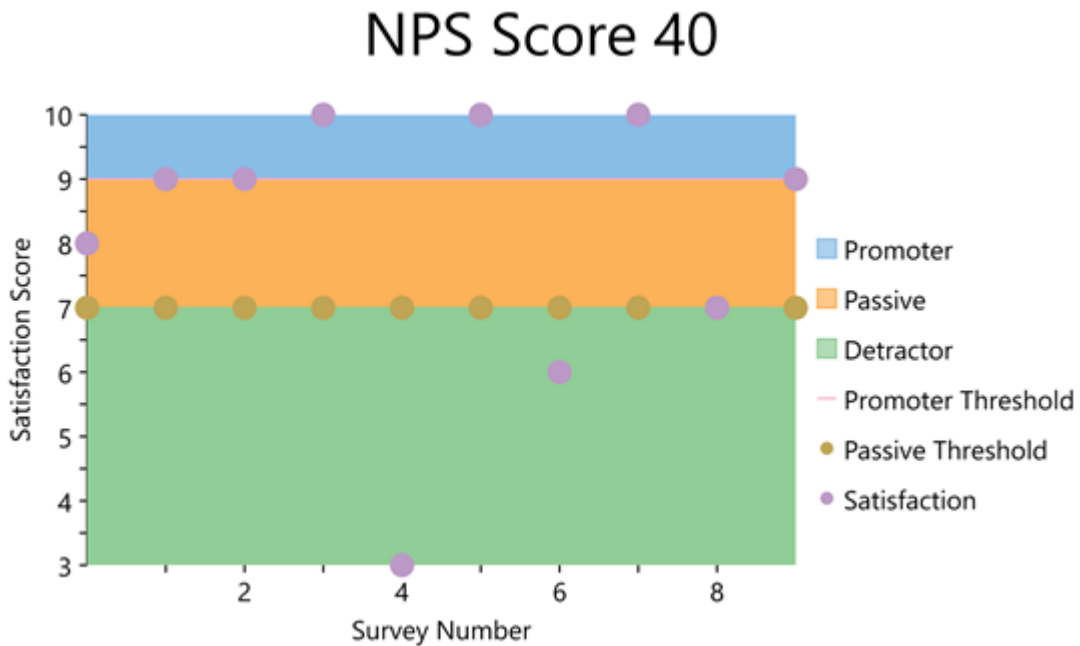
## Zones

FlexChart allows you to create and apply colored regions called zones on the chart. These colored zones categorize the data points plotted on the chart into regions, making it easier for the user to read and understand the data. Users can easily identify the category in which a particular data point lies.

To explain how creating zones can be helpful, consider a customer satisfaction survey that aims at identifying whether



a customer is a promoter, a passive customer, or a detractor for a specific product. The responses recorded in the survey can be used to calculate NPS (Net Promoter Score) that classifies customers into promoters, passives, and detractors. This scenario can be realized in FlexChart by plotting customer responses as data points in chart and categorizing them in colored zones using area chart, separated by line type data series as follows:



### Creating zones in FlexChart

In FlexChart, zones can be created as data series available through the [ChartSeries](#) class. Each zone can be created as area charts by setting the [ChartType](#) property to **Area**, highlighted in distinct colors. To distinguish each zone, line type data series can be created as thresholds in the chart.

Complete the following steps to create zones in FlexChart.

#### Step 1: Create a class to record survey result data

1. In the Solution Explorer, right-click your project name (portable app).
2. Select **Add | New Item**. The **Add New Item** dialog appears.
3. Select **Class** from the dialog and provide a name to it, for example **SurveyResult**.
4. Click **Add** to add the class to your project.
5. Add the following code to the **SurveyResult** class.

```
C#
public class SurveyResult
{
    public int surveyNumber { get; set; }
    public int satisfaction { get; set; }

    public SurveyResult(int surveyNumber, int satisfaction)
    {
        this.surveyNumber = surveyNumber;
        this.satisfaction = satisfaction;
    }
}
```

#### Step 2: Initialize a FlexChart control

1. Right-click your project in the **Solution Explorer** and select **Add | New Item**. The **Add New Item** dialog appears.
2. Select **Forms Xaml Page** from the installed templates and provide a name to it, for example **SurveyZones**.
3. Click **Add** to add the XAML page to your project.
4. Initialize the FlexChart control in the SurveyZones.xaml page by adding the following XAML code.

**XAML**

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:cl="clr-
namespace:Cl.Xamarin.Forms.Chart;assembly=Cl.Xamarin.Forms.Chart"
             x:Class="QuickstartChart.SurveyZones">
    <StackLayout>
        <Label x:Name="lbl" HorizontalOptions="Center" Font="Large"/>
        <Grid VerticalOptions="FillAndExpand" >
            <cl:FlexChart x:Name="flex" BindingX="surveyNumber"
VerticalOptions="FillAndExpand"
                        ChartType="Scatter" Grid.Row="0">
                <cl:FlexChart.AxisX>
                    <cl:ChartAxis AxisLine="false" MajorGrid="false" Title="Survey Number"
/>
                </cl:FlexChart.AxisX>
                <cl:FlexChart.AxisY>
                    <cl:ChartAxis Title="Satisfaction Score" Max="10"/>
                </cl:FlexChart.AxisY>
            </cl:FlexChart>
        </Grid>
    </StackLayout>
</ContentPage>
```

**Step 3: Bind data and create zones**

1. Expand the SurveyZones.xaml node and open the C# code.
2. Add the following import statements.

**C#**

```
using Xamarin.Forms;
using Cl.Xamarin.Forms.Chart;
using System.Collections.ObjectModel;
```

3. Add the following code in SurveyZones() class constructor to create zones in chart and display NPS score in a data label.

**C#**

```
public partial class SurveyZones : ContentPage
{
    public SurveyZones()
    {
        InitializeComponent();
    }
}
```

```
//customer survey results
int[] scores = { 8, 9, 9, 10, 3, 10, 6, 10, 7, 9 };

ObservableCollection<SurveyResult> results = new
ObservableCollection<SurveyResult>();
flex.ItemsSource = results;

Point[] zone1 = new Point[10];
Point[] zone2 = new Point[10];
Point[] zone3 = new Point[10]; ;

Point[] threshold1 = new Point[10];
Point[] threshold2 = new Point[10];

//populate points for zones and thresholds
for (int i = 0; i < 10; i++)
{
    results.Add(new SurveyResult(i, scores[i]));
    zone1[i] = new Point(i, 10);
    zone2[i] = new Point(i, 9);
    zone3[i] = new Point(i, 7);
    threshold1[i] = new Point(i, 9);
    threshold2[i] = new Point(i, 7);
}

//Zone 1 for promoters
var seriesZone1 = new ChartSeries();
seriesZone1.ItemsSource = zone1;
seriesZone1.Binding = "Y";
seriesZone1.BindingX = "X";
seriesZone1.ChartType = C1.Xamarin.Forms.Chart.ChartType.Area;
seriesZone1.SeriesName = "Promoter";

//Zone 2 for passives
var seriesZone2 = new ChartSeries();
seriesZone2.ItemsSource = zone2;
seriesZone2.Binding = "Y";
seriesZone2.BindingX = "X";
seriesZone2.ChartType = ChartType.Area;
seriesZone2.SeriesName = "Passive";

//Zone 3 for detractors
var seriesZone3 = new ChartSeries();
seriesZone3.ItemsSource = zone3;
seriesZone3.Binding = "Y";
seriesZone3.BindingX = "X";
seriesZone3.ChartType = ChartType.Area;
seriesZone3.SeriesName = "Detractor";

flex.Series.Add(seriesZone1);
flex.Series.Add(seriesZone2);
```

```
flex.Series.Add(seriesZone3);

//Promotor Threshold line
var seriesThreshold1 = new ChartSeries();
seriesThreshold1.ItemsSource = threshold1;
seriesThreshold1.Binding = "Y";
seriesThreshold1.BindingX = "X";
seriesThreshold1.SeriesName = "Promoter Threshold";
seriesThreshold1.ChartType = ChartType.Line;

//Passive Threshold line
var seriesThreshold2 = new ChartSeries();
seriesThreshold2.ItemsSource = threshold2;
seriesThreshold2.Binding = "Y";
seriesThreshold2.BindingX = "X";
seriesThreshold2.SeriesName = "Passive Threshold";

flex.Series.Add(seriesThreshold1);
flex.Series.Add(seriesThreshold2);

//add customer satisfaction results
var satisfactionSeries = new ChartSeries();
satisfactionSeries.SeriesName = "Satisfaction";
satisfactionSeries.Binding = "satisfaction";

flex.Series.Add(satisfactionSeries);
lbl.Text = "NPS Score " + GetNPS(scores).ToString();
}

public double GetNPS(int[] scores)
{
    double promoter = 0;
    double detractor = 0;
    foreach (int score in scores)
    {
        if (score >= 9)
        {
            promoter++;
        }
        else if (score < 7)
        {
            detractor++;
        }
    }
    double nps = ((promoter - detractor) / scores.Length) * 100;
    return nps;
}
}
```

## FlexGrid

**FlexGrid** provides a powerful and flexible way to display data from a data source in tabular format. FlexGrid is a full-featured grid, providing various features including automatic column generation; sorting, grouping and filtering data using the *CollectionView*; and intuitive touch gestures for cell selection, sorting, scrolling and editing. FlexGrid brings a spreadsheet-like experience to your mobile apps with quick cell editing capabilities.

FlexGrid provides design flexibility with conditional formatting and cell level customization. This allows developers to create complex grid-based applications, as well as provides the ability to edit and update databases at runtime.

## Quick Start: Add Data to FlexGrid

This section describes how to add a FlexGrid control to your portable or shared app and add data to it. For information on how to add Xamarin components in C# or XAML, see [Adding Xamarin Components using C#](#) or [Adding Xamarin Components using XAML](#).

This topic comprises of three steps:

- **Step 1: Create a Data Source for FlexGrid**
- **Step 2: Add a FlexGrid control**
- **Step 3: Run the Application**

The following image shows how the FlexGrid appears, after completing the steps above:

	ID	Name	Country	CountryID
	0	Oprah Orsted	Egypt	15
	1	Xavier Frommer	Congo	18
	2	Herb Paulson	Egypt	15
	3	Charlie Jammers	Congo	18
	4	Steve Krause	Thailand	20
	5	Larry Neiman	Egypt	15
	6	Gil Griswold	Brazil	4

### Step 1: Create a Data Source for FlexGrid

The following class serves as a data source for the FlexGrid control.

C#

```
public class Customer
{
    int _id, _countryID;
    string _first, _last;
    bool _active;
    double _weight;
    DateTime _hired;
    static Random _rnd = new Random();
}
```

```
static string[] _firstNames =
"Gil|Oprah|Xavier|Herb|Charlie|Larry|Steve".Split('|');
static string[] _lastNames =
"Orsted|Frommer|Jammers|Krause|Neiman".Split('|');
static string[] _countries = "Brazil|Congo|Egypt|United
States|Japan|Thailand".Split('|');

public Customer()
: this(_rnd.Next(10000))
{
}
public Customer(int id)
{
    ID = id;
    First = GetString(_firstNames);
    Last = GetString(_lastNames);
    CountryID = _rnd.Next() % _countries.Length;
    Active = _rnd.NextDouble() >= .5;
    Hired = DateTime.Today.AddDays(-_rnd.Next(1, 365));
    Weight = 50 + _rnd.NextDouble() * 50;
}
public int ID
{
    get { return _id; }
    set { _id = value; }
}
public string Name
{
    get { return string.Format("{0} {1}", First, Last); }
}
public string Country
{
    get { return _countries[_countryID]; }
}
public int CountryID
{
    get { return _countryID; }
    set {_countryID = value; }
}
public bool Active
{
    get { return _active; }
    set { _active = value; }
}
public string First
{
    get { return _first; }
    set {_first = value; }
}
public string Last
{

```

```

        get { return _last; }
        set { _last = value; }
    }
    public DateTime Hired
    {
        get { return _hired; }
        set { _hired = value; }
    }
    public double Weight
    {
        get { return _weight; }
        set { _weight = value; }
    }
    static string GetString(string[] arr)
    {
        return arr[_rnd.Next(arr.Length)];
    }
    static string GetName()
    {
        return string.Format("{0} {1}", GetString(_firstNames),
        GetString(_lastNames));
    }
    // Provide static list.
    public static ObservableCollection<Customer> GetCustomerList(int count)
    {
        var list = new ObservableCollection<Customer>();
        for (int i = 0; i < count; i++)
        {
            list.Add(new Customer(i));
        }
        return list;
    }
    //Provide static value members.
    public static string[] GetCountries() { return _countries; }
    public static string[] GetFirstNames() { return _firstNames; }
    public static string[] GetLastNames() { return _lastNames; }
}

```

**Back to Top**

## Step 2: Add a FlexGrid control

Complete the following steps to initialize a FlexGrid control in C# or XAML.

### In Code

1. Add a new class (for example QuickStart.cs) to your project and include the following references:

```

C#
using Xamarin.Forms;
using Cl.Xamarin.Forms.Grid;

```

2. Instantiate a FlexGrid control in a new method GetGridControl().

```

C#

```

```
public static FlexGrid GetGridControl()
{
    //Create an instance of the Control and set its properties
    FlexGrid grid = new FlexGrid();
    grid.ItemsSource = Customer.GetCustomerList(10);
    return grid;
}
```

## In XAML

1. Add a new Forms XAML Page (for example QuickStart.xaml) to your project and modify the <ContentPage>tag to include the following references:

### XAML

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="Appl.QuickStart"
xmlns:cl="clr-namespace:C1.Xamarin.Forms.Grid;assembly=C1.Xamarin.Forms.Grid">
```

2. Initialize a FlexGrid control by adding the markup for the control between the <ContentPage> </ContentPage>tags and inside the <StackLayout> </StackLayout>tags, as shown below.

### XAML

```
<StackLayout>
    <Grid VerticalOptions="FillAndExpand">
        <cl:C1FlexGrid x:Name="grid"/>
    </Grid>
</StackLayout>
```

3. In the **Solution Explorer**, expand the QuickStart.xaml node and open QuickStart.xaml.cs to view the C# code.
4. In the QuickStart() class constructor, set the BindingContext for the FlexGrid.

The following code shows what the QuickStart() class constructor looks like after completing this step.

### C#

```
public QuickStart()
{
    InitializeComponent();
    grid.BindingContext = new Customer();
}
```

## Back to Top

## Step 3: Run the Application

1. In the **Solution Explorer**, double-click **App.cs** to open it.
2. Complete the following steps to display the FlexGrid control.
  - o **To return a C# class:** In the class constructor App(), set a new ContentPage as the MainPage and assign the control to the ContentPage's content by invoking the method GetGridControl() defined in the previous procedure, **Step 2: Add a FlexGrid Control**.

The following code shows the class constructor App() after completing steps above.

### C#



```
public App()
{
    // The root page of your application
    MainPage = new ContentPage
    {
        Content = QuickStart.GetGridControl()
    };
}
```

- **To return a Forms XAML Page:** In the class constructor `App()`, set the Forms XAML Page `QuickStart` as the `MainPage`.

The following code shows the class constructor `App()`, after completing this step.

```
C#
public App()
{
    // The root page of your application
    MainPage = new QuickStart();
}
```

3. Some additional steps are required to run iOS and UWP apps:

- **iOS App:**

1. In the **Solution Explorer**, double click `AppDelegate.cs` inside `YourAppName.iOS` project to open it.
2. Add the following code to the `FinishedLaunching()` method.

```
C#
C1.Xamarin.Forms.Grid.Platform.iOS.FlexGridRenderer.Init();
```

- **UWP App:**

1. In the **Solution Explorer**, expand `MainPage.xaml`.
2. Double click `MainPage.xaml.cs` to open it.
3. Add the following code to the class constructor.

```
C#
C1.Xamarin.Forms.Grid.Platform.UWP.FlexGridRenderer.Init();
```

4. Press **F5** to run the project.

**Back to Top**

## Features

### Custom Cells

FlexGrid gives you complete control over the contents of the cells. You can customize each column's appearance by modifying the `CellTemplate` or `CellEditingTemplate` properties in XAML, or you can create your own `Cell Factory` class to customize cell content entirely in code.

	First	Last	Performance
	Vic	Lehman	
	Andy	Trask	
	Gil	Paulson	
	Noah	Krause	
	Larry	Cole	
	Ed	Cole	
	Mark	Myers	


The following code example demonstrates how to add custom cell content in the FlexGrid control.

#### Custom Cells in XAML

The **CellTemplate** and **CellEditingTemplate** properties are used to modify a column's appearance in XAML. These properties accept a standard **DataTemplate** as their value. The following XAML example demonstrates how to add custom cell content, such as a **RadialGauge**, using the **CellTemplate** property on a specific column.

#### XAML

```
<c1:FlexGrid AutoGenerateColumns="False" IsReadOnly="True">
  <c1:FlexGrid.Columns>
    <c1:GridColumn Header="My Value">
      <c1:GridColumn.CellTemplate>
        <DataTemplate >
          <gauge:C1RadialGauge IsAnimated="False" Value="{Binding
MyValue}" Min="50"
                                Max="100" ShowText="None" ShowRanges="False"
                                WidthRequest="50" HeightRequest="50">
            <gauge:C1RadialGauge.Ranges>
              <gauge:GaugeRange Min="50" Max="80" Color="Green"/>
              <gauge:GaugeRange Min="80" Max="90"
Color="Yellow"/>
              <gauge:GaugeRange Min="90" Max="100" Color="Red"/>
            </gauge:C1RadialGauge.Ranges>
          </gauge:C1RadialGauge>
        </DataTemplate>
      </c1:GridColumn.CellTemplate>
    </c1:GridColumn>
  </c1:FlexGrid.Columns>
</c1:FlexGrid>
```

 For the best performance, your cell template should contain a **single control** (View). In this case, the Xamarin.Forms platform is capable of efficiently rendering single Views as bitmaps on Android and iOS so performance is not dramatically affected. If your cell template contains a layout control (such as a **Grid** or **StackLayout**), it will work, but a new View will be created for every cell on **Android and iOS**. This may drastically affect performance if you have a lot of rows. You can work around this limitation by creating a [custom renderer](#). This limitation does not impact **CellEditingTemplate** because only one cell is in edit mode.

### Custom Cells in Code

FlexGrid provides a simpler interface for customizing cells in code. The GridCellFactory class is used by the grid to create every cell shown in the grid. Custom cell factories can be highly customized, or they can be general, reusable classes for defining cell appearances.

To customize cells code, you first define your own class that extends GridCellFactory. Then you assign an instance of your class to the FlexGrid.CellFactory property. Your cell factory class should override either the BindCellContent method or the CreateCellContent method. To customize the gridlines appearance you should override CreateCell.

- **BindCellContent:** Override this method to customize the cell text. The result is a text-only cell, but you can conditionally format the text as needed.
- **CreateCellContent:** Override this method to completely customize the cell content. The result is a View (control). You would use this method to insert your own control, such as an Image or gauge.
- **PrepareCell:** Override this method to customize the cell background and border (gridlines).

The following code example demonstrates a cell factory that overrides **BindCellContent** and conditionally sets the cell text red or green depending on the value:

C#

```
// set the cell factory property on FlexGrid to custom class
grid.CellFactory = new MyCellFactory();

// Custom Cell Factory class that applies conditional formatting to cell text
public class MyCellFactory : GridCellFactory
{
    ///
    /// Override BindCellContent to customize the cell content
    ///
    public override void BindCellContent(GridCellType cellType, GridCellRange range,
    View cellContent)
    {
        base.BindCellContent(cellType, range, cellContent);
        if(cellType == GridCellType.Cell && cellType !=
GridCellType.ColumnHeader && range.Column == 3)
        {
            var label = cellContent as Label;
            if(label != null)
            {
                var originalText = label.Text;
                double cellValue;
                if(double.TryParse(originalText, out cellValue))
                {
                    label.TextColor = cellValue < 70.0 ? Color.Red : Color.Green;
                    label.Text = String.Format("{0:n2}", cellValue);
                }
            }
        }
    }
}
```

```

        }
    }
}

```

FlexGrid supports just one cell factory, however, a single cell factory can handle any number of custom cells in multiple columns. For instance, you can apply different formatting for any number of columns by just providing additional IF statements and checking the range.Column parameter.

The following code example shows the default CreateCellContent method that returns a Label as the cell content. You can modify this to return your own View.

#### In Code

C#

```

public class MyCellFactory: GridCellFactory
{
    ///
    /// Override CreateCellContent to return your own custom view as a cell
    ///
    public override View CreateCellContent(GridCellType cellType,
GridCellRange range, object cellContentType)
    {
        if (cellType == GridCellType.Cell)
        {
            if (Grid.Columns.Count>range.Column)
            {
                var r = Grid.Rows[range.Row];
                var c = Grid.Columns[range.Column];
                return base.CreateCellContent(cellType, range, cellContentType);
            }
            return null;
        }
        else if (cellType == GridCellType.ColumnHeader)
        {
            return new Label();
        }
        else
        {
            return null;
        }
    }
}

```

The following code example shows how to customize cell background colors using PrepareCell.

#### In Code

C#

```

public class MyCellFactory : GridCellFactory
{
    ///
    /// Override CreateCell to customize the cell frame/gridlines ///

```

```

    public override void PrepareCell(GridCellType cellType, GridCellRange range,
GridCellView cell)
    {
        base.PrepareCell(cellType, range, cell);
        if (cellType == GridCellType.Cell && range.Column == 3)
        {
            var cellValue = Grid[range.Row, range.Column] as int?;
            if (cellValue.HasValue)
            {
                cell.BackgroundColor = cellValue < 50.0 ?
Color.FromRgb((double)0xFF / 255.0, (double)0x70 / 255.0, (double)0x70 / 255.0) :
Color.FromRgb((double)0x8E / 255.0, (double)0xE9 / 255.0, (double)0x8E / 255.0);
            }
        }
    }
}

```

## Custom Cell Editor

To provide a custom cell editor for a column you would add your custom editor to the **CellEditingTemplate**. The following XAML example demonstrates how to add a custom cell editor, such as a DatePicker, using the CellEditingTemplate property on a specific column.

### In XAML

#### XAML

```

<cl:FlexGrid AutoGenerateColumns="False">
  <cl:FlexGrid.Columns>
    <cl:GridColumn Binding="HireDate">
      <cl:GridColumn.CellEditingTemplate>
        <DataTemplate>
          <DatePicker Date="{Binding ., Mode=TwoWay}" />
        </DataTemplate>
      </cl:GridColumn.CellEditingTemplate>
    </cl:GridColumn>
  </cl:FlexGrid.Columns>
</cl:FlexGrid>

```

## Customize Appearance

FlexGrid has various built-in properties to customize grid's appearance. User can customize various attributes, such as background color, alternating row color, text color, header color, font, selection mode, selected cell color, etc.

The image below shows customized appearance in FlexGrid after these properties have been set.

	ID	First	Last	Weight
	0	Jack	Myers	88.5
	1	Zeb	Krause	69.5
	2	Ted	Griswold	71.1
	3	Karl	Jammers	76.4
	4	Andy	Griswold	83.4
	5	Larry	Frommer	72.5
	6	Gil	Ambers	90.2

The following code example demonstrates how to set this property in C# and XAML. The example uses the sample created in the [Quick Start](#) section.

#### In Code

C#

```
grid.BackgroundColor = Color.FromHex("#FCF0E5");
grid.AlternatingRowBackgroundColor = Color.FromHex("#DA6CB5");
grid.SelectionMode = GridSelectionMode.Cell;
grid.SelectionTextColor = Color.FromHex("#FFFFFF");
grid.ColumnHeaderBackgroundColor = Color.FromHex("#7B7E7D");
grid.ColumnHeaderTextColor = Color.FromHex("#FFFFFF");
grid.SelectionBackgroundColor = Color.FromHex("#FCF0E5");
```

#### In XAML

XAML

```
<cl:FlexGrid SelectionMode="Cell"

    BackgroundColor="White" AlternatingRowBackgroundColor="Purple"

    ColumnHeaderBackgroundColor="Silver"
    SelectionBackgroundColor="Navy" SelectionTextColor="White" x:Name="grid"

    AutoGenerateColumns="False">
```

## Clipboard and Keyboard Support

FlexGrid comes with clipboard support to readily provide cut, copy and paste operations. The control also supports hardware keyboards by allowing navigation through arrow keys.

The supported keystrokes and their purpose are listed alphabetically as follows:

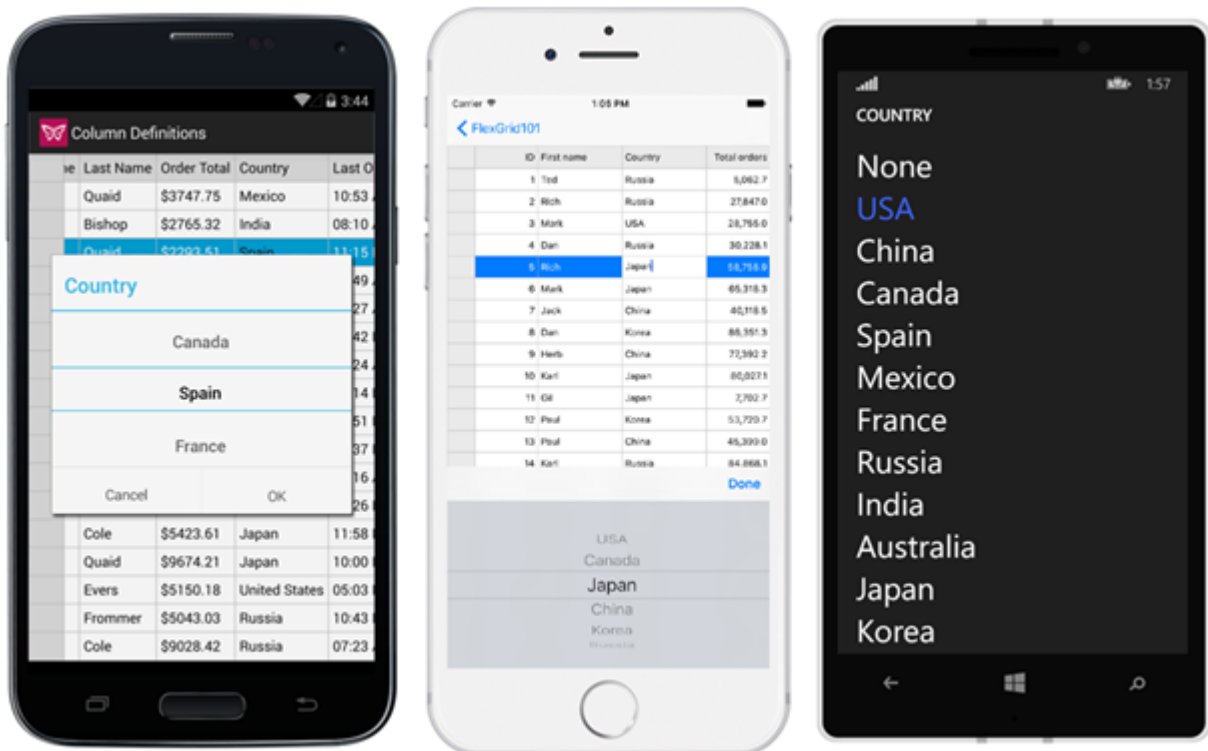
Keystroke	Description/Purpose
-----------	---------------------

Ctrl+C	Copies the selected text.
Ctrl+X	Cuts the selected text.
Ctrl+V	Pastes the copied text.
F2	Enters the edit mode.
Enter/Return	Leaves the edit mode.
Left	Navigates cell selection towards the left.
Right	Navigates cell selection towards the right.
Up	Navigates the cell selection upward.
Down	Navigates the cell selection downward.
Shift+Left	Expands column range towards the left or decreases column range based on the context. If range has already expanded to right, this will gradually decrease range selection down to one column. After this single column threshold is hit, it will then expand the selection to the left if it continues to be pressed after this.
Shift+Right	Expands column range towards the right or decreases column range based on the context. If range has already expanded to left, this will gradually decrease range selection down to one column. After this single column threshold is hit, it will then expand the selection to the right if it continues to be pressed after this.
Shift+Up	Expands row range upwards or decreases row range based on the context. If range has already expanded downwards, this will gradually decrease range selection to one row. After this single row threshold is hit, it will then expand the selection to the upwards if it continues to be pressed after this.
Shift+Down	Expands row range downwards or decreases row range based on the context. If range has already expanded upwards, this will gradually decrease range selection to one row. After this single row threshold is hit, it will then expand the selection to the downwards if it continues to be pressed after this.

## Data Mapping

Data Mapping provides auto look-up capabilities in FlexGrid. For example, you may want to display a customer name instead of his ID, or a color name instead of its RGB value. When data mapping is configured for a column, a picker is displayed when the user edits the cell.

The image given below shows a FlexGrid with Data Mapping.



The `GridDataMap` class has three key properties.

- `ItemsSource` (IEnumerable) - an IEnumerable collection that contains the items to map.
- `SelectedValuePath` (string) - the name of the property that contains the keys (data values).
- `DisplayMemberPath` (string) - the name of the property to use as the visual representation of the items.

The code below binds a grid to a collection of customers, then assigns a `GridDataMap` to the grid's 'CountryId' column so that the grid displays country names instead of raw IDs.

### In Code

C#

```
grid.ItemsSource = customers;
grid.Columns["CountryId"].DataMap = new GridDataMap()
{ ItemsSource = countries, DisplayMemberPath = "Country", SelectedValuePath = "ID" };
```

If you use a `KeyValuePair` collection as the `GridDataMap` `ItemsSource`, you should set the `DisplayMemberPath` to 'Value' and `SelectedValuePath` to 'Key'. In case you follow MVVM, you may want to define the data map binding in XAML. The markup below shows how to configure that data map in XAML on a single `GridColumn`.

### In XAML

XAML

```
<cl:GridColumn Binding="CountryId" Header="Country">
  <cl:GridColumn.DataMap>
    <cl:GridDataMap ItemsSource="{Binding Countries, Source={StaticResource
viewModel}}">
      SelectedValuePath="ID"
      DisplayMemberPath="Name" />
    </cl:GridDataMap>
  </cl:GridColumn.DataMap>
```



```
</cl:GridColumn>
```

The view model exposes a collection of countries with an ID and Name property. One way to instantiate your view model is as a static XAML resource like it's used in the example above. The markup below shows how to add a view model to your page's resources.

#### XAML

```
<ContentPage.Resources>
  <ResourceDictionary>
    <local:MyViewModel x:Key="viewModel" />
  </ResourceDictionary>
</ContentPage.Resources>
```

## Defining Columns

With automatic column generation as one of the default features of FlexGrid, the control lets you specify the columns, allowing you to choose which columns to show, and in what order. This gives you control over each column's width, heading, formatting, alignment, and other properties. To define columns for the FlexGrid, ensure that the [AutoGenerateColumns](#) is set to **false** (by default this property is **true**).

The image below shows how the FlexGrid appears, after defining columns.

	ID	First	Last	Weight
	0	Larry	Jammers	50.2
	1	Karl	Stevens	63.9
	2	Noah	Neiman	92.4
	3	Fred	Stevens	51.7
	4	Karl	Stevens	62.0
	5	Ben	Krause	93.8
	6	Ed	Griswold	78.1

The following code example demonstrates how to define FlexGrid columns in C# and XAML. The example uses the sample created in the [Quick start](#) section.

#### In Code

##### C#

```
var data = Customer.GetCustomerList(100);
grid.ItemsSource = data;
```

#### In XAML

##### XAML

```
<Grid VerticalOptions="FillAndExpand">
  <cl:FlexGrid x:Name="grid" AutoGenerateColumns="False">
    <cl:FlexGrid.Columns>
      <cl:GridColumn Binding="ID" Width="100"/>
      <cl:GridColumn Binding="First" Width="Auto"/>
      <cl:GridColumn Binding="Last"/>
      <cl:GridColumn Binding="Weight" Format="N1"/>
    </cl:FlexGrid.Columns>
  </cl:FlexGrid>
</Grid>
```

## Editing

FlexGrid has built-in support for fast, in-cell editing like you find in Excel. There is no need to add extra columns with Edit buttons that switch between display and edit modes. Users can start editing by typing into any cell. This puts the cell in quick-edit mode. In this mode, pressing a cursor key finishes the editing and moves the selection to a different cell.

Another way to start editing is by clicking a cell twice. This puts the cell in full-edit mode. In this mode, pressing a cursor key moves the caret within the cell text. To finish editing and move to another cell, the user must press the Enter key. Data is automatically coerced to the proper type when editing finishes. If the user enters invalid data, the edit is cancelled and the original data remains in place. You can disable editing at the grid using the [IsReadOnly](#) property of the grid.

FlexGrid provides support for various types of Editing, including inline, form-based and custom cell editing.

## Inline Editing

FlexGrid allows a user to perform in-line editing using your default device keyboard. Double clicking inside a cell puts it into a quick edit mode. Once you select the content inside the cell or row, it gives you options to Cut, Copy, Replace etc. for a smooth editing experience.

Once you have entered the new data, simply press **Enter**, this automatically updates the data in the appropriate format. You can set the [IsReadOnly](#) to **true** for the rows and columns that you want to restrict editing for.

The image below shows how the FlexGrid appears, after these properties have been set.

	ID	Name ▲
	24	Andy Frommer
	90	Andy Kraus
	60	Andy Neiman
	73	Andy Trask
	44	Ben Myers
	65	Ben Paulson
	3	Ben Quaid
	19	Ben Ulam

The following code example demonstrates how to set this property in C# and XAML. The example uses the sample created in the [Quick start](#) section.

#### In Code

C#

```
grid.IsReadOnly = false;
```

#### In XAML

XAML

```
<cl:FlexGrid IsReadOnly="False" x:Name="grid"/>
```

## Filtering

FlexGrid supports filtering through the `ICollectionView` interface. To enable filtering, set the `collectionView.filter` property to a function that determines objects for filtering data in the FlexGrid control. Filtering allows a user to display subsets of data out of large data sets based on the criteria defined. The `CollectionView` interface provided by FlexGrid provides a flexible and efficient way to filter and display the desired dataset.

The FlexGrid control lets a user perform custom filtering by adding a text box to enter a value and display a particular set of data. It also allows a user to use various options, such as `BeginsWith`, `Contains`, `EndsWith`, `Equals`, `LessThan`, `GreaterThan` etc. A user can define the filtering patterns based on their requirements, which can be a specific data or an approximate set of values.

## Search Box Filtering

FlexGrid provides you flexibility to use a search box to filter out data. Users can add the filter search box and set its attributes, including its height, width, color, text, filtering pattern as per their requirements. This example demonstrates a simple grey text box that lets you type the value you want to search in the grid. For example, when you type **Be** in the Filter text box, the collection view interface filters the grid data to display all the values containing **Be**.

Be		
	ID	Name
	2	Fred Ambers
	5	Ben Danson
	7	Ben Myers
	11	Ben Bishop
	13	Ben Krause
	16	Ben Krause
	21	Jack Neiman

The following code example demonstrates how to apply Filtering in FlexGrid in C# and XAML. The example uses the data source, **Customer.cs** created in the [Quick Start](#) section.

1. Add a new Forms XAML Page, Filter.xaml, to your project.
2. To initialize a FlexGrid control and add filtering in XAML, modify the markup between the <ContentPage> </ContentPage> tags and inside the <StackLayout> </StackLayout> tags, as shown below.

### In XAML

#### XAML

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
              xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
              x:Class="App1.Filter">

    xmlns:cl="clrnamespace:C1.Xamarin.Forms.Grid;assembly=C1.Xamarin.Forms.Grid">
    <StackLayout Orientation="Vertical">
        <Entry x:Name="filterEntry" Placeholder="Filter Text" ></Entry>
        <cl:FlexGrid x:Name="grid" AutoGenerateColumns="True"
            VerticalOptions="FillAndExpand">
```

```

        <cl:FlexGrid.Behaviors>
            <cl:FullTextFilterBehavior FilterEntry="{x:Reference Name=filterEntry}"
                <cl:FullTextFilterBehavior FilterEntry="{x:Reference
Name=filterEntry}"
            Mode="WhileTyping" MatchNumbers="True" TreatSpacesAsAndOperator="True" />
        </cl:FlexGrid.Behaviors>
    </cl:FlexGrid>
</StackLayout>
</ContentPage>

```

3. In the **Solution Explorer**, expand the Filter.xaml node and open Filter.xaml.cs to view the C# code.
4. Add the following code in the Filter class constructor to initialize the FlexGrid control and add data to it:

#### In Code

```

C#
InitializeComponent();
this.grid.ItemsSource = Appl.Customer.GetCustomerList(50);

```

## Formatting Columns

You can convert the raw values appearing in any column of the FlexGrid into different formats. FlexGrid supports standard .NET format strings used to display numeric values as local currency, percentages, dates, and more. Formatting only applies to display value of a cell and does not affect the underlying data.

### Formatting Numbers and Dates

You can specify a column's format by setting its [Format](#) property to a valid string. The string should contain one or two characters from a known list of formats. The first character specifies the format (format specifier) while the second optional character specifies precision (precision specifier). For example, the following format string converts a raw value to display local currency up to 2 decimal places.

```

C#
flexGrid.Columns["UnitPrice"].Format = "c2";

```

The following table shows all the numeric format strings that FlexGrid currently supports. Note that these format strings are not case sensitive.

Format Specifier	Name	Precision Specifier	Example
"C" or "c"	Currency	Number of decimal digits	123.4567 (C2) → \$123.46
"D" or "d"	Decimal	Minimum number of digits	1234 (D6) → 001234
"E" or "e"	Exponential	Number of decimal digits	1,234 (E2) → 1.23E3
"F" or "f"	Fixed Point	Number of decimal digits	-1234.56 (F4) → -1234.5600

"G" or "g"	General	Number of significant digits	123.45 (G4) → 123.5
"N" or "n"	Number	Desired number of decimal places	1234 (N1) → 1234.0
"P" or "p"	Percent	Desired number of decimal places	1 (P2) → 100.00%
"X" or "x"	Hexadecimal	Desired number of digits in the result	123 (X2) → 7B

The following table shows the date/time format strings that FlexGrid currently supports.

Format Specifier	Description	Example
"d"	Short date pattern	3/31/2016 31/3/2016 (FR) 2016/3/31 (JP)
"D"	Long date pattern	Thursday, March 31, 2016
"f"	Full date, short time pattern	Thursday, March 31, 2016 12:00 AM
"F"	Full date, full time pattern	Thursday, March 31, 2016 12:00:00 AM
"g"	General date, short time pattern	3/31/2016 12:00 AM
"G"	General date, long time pattern	3/31/2016 12:00:00 AM
"M" or "m"	Month, day pattern	March 31
"t"	Short time pattern	12:00 AM
"T"	Long time pattern	12:00:00 AM
"u"	Universal sortable pattern	2016-03-31 12:00:00Z
"U"	Universal full pattern	Thursday, March 31, 2016 12:00:00 AM
"Y" or "y"	Year, month pattern	March, 2016

### Custom Date/Time Format Strings

FlexGrid also supports custom date and time format strings that allow you to display date and time values in numerous ways. For example, the format string below converts March 31st 2016 to be displayed as "2016-03".

C#
<code>column.Format = "yyyy-MM";</code>

The following table lists some common format specifier for creating custom date and time strings.

Format Specifier	Description	Example
------------------	-------------	---------

"d"	Day of the month	1
"dd"	Day of the month	01
"ddd"	Abbreviated day of the week	Mon
"dddd"	Full day of the week	Monday
"h"	The hour using 12-hour clock	1
"hh"	The hour using 12-hour clock	01
"H"	The hour using 24-hour clock	13
"HH"	The hour using 24-hour clock	13
"m"	Minute of time	1
"mm"	Minute of time	01
"M"	Month number	1
"MM"	Month number	01
"MMM"	Abbreviated month name	Mar
"MMMM"	Full month name	March
"s"	Second of time	1
"ss"	Second of time	01
"tt"	The AM/PM Designator	AM
"yy"	Abbreviated year	16
"yyyy"	Full year	2016
"\"	Escape character	H\H → 13H
Any other character	The character is copied to the result string	yyyy-MM → 2016-03

### Formatting Other Data Types

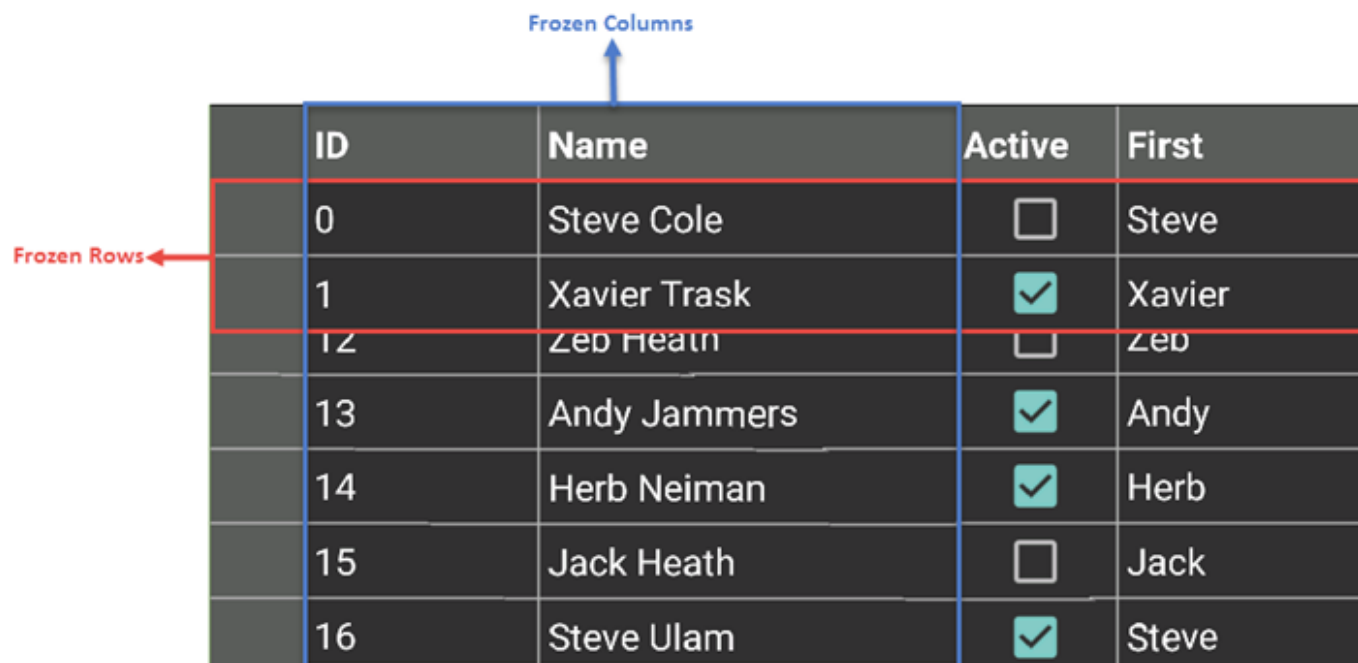
If you need to format a column that is not numerical or date/time, refer to [Custom Cells](#) and [Data Mapping](#) topics.

## Frozen Rows and Columns

FlexGrid allows you to freeze rows and columns so that they remain visible as the user scrolls the grid. Frozen cells can be edited and selected as regular cells, exactly as in Excel. A user can freeze rows and columns separately and also simultaneously. While working with large data sets in grid, it is convenient to keep some of the rows or/and columns locked in the view by setting the values for [FrozenRows](#) and [FrozenColumns](#) properties of FlexGrid.

Freeing a cell in the grid doesn't affect the SelectionMode of the FlexGrid. If a user has set a value of SelectionMode in FlexGrid, then the frozen cells along with the cells that are not frozen can be selected simultaneously by the user.

In this example, first two rows and first two columns of the FlexGrid are frozen. The image below shows how the FlexGrid appears, after applying cell freezing to rows and columns:



	ID	Name	Active	First
	0	Steve Cole	<input type="checkbox"/>	Steve
	1	Xavier Trask	<input checked="" type="checkbox"/>	Xavier
	2	Zeb Heath	<input type="checkbox"/>	Zeb
	3	Andy Jammers	<input checked="" type="checkbox"/>	Andy
	4	Herb Neiman	<input checked="" type="checkbox"/>	Herb
	5	Jack Heath	<input type="checkbox"/>	Jack
	6	Steve Ulam	<input checked="" type="checkbox"/>	Steve

The following code example demonstrates how to freeze rows and columns in C# and XAML. The example uses the sample created in the [Quick start](#) section.

#### In Code

C#

```
grid.FrozenColumns = 2;  
grid.FrozenRows = 2;
```

#### In XAML

XAML

```
<Grid>  
  <cl:FlexGrid x:Name="grid" FrozenColumns="2" FrozenRows="2"/>  
</Grid>
```

## Grouping

FlexGrid supports grouping through the **ICollectionView**. To enable grouping, add one or more **GroupDescription** objects to the **C1GroupedCollectionView** property. **GroupDescription** objects are flexible, allowing you to group data based on value or on grouping functions. You can tap anywhere in the group row to expand or collapse grouped rows.

The image below shows how the FlexGrid appears, after these properties have been set.



	ID	Name	Country	CountryID
	▲ Key: Brazil (1 items)			
	0	Noah Jammers	Brazil	4
	▶ Key: Indonesia (1 items)			
	▶ Key: Vietnam (1 items)			
	▲ Key: Egypt (1 items)			
	3	Charlie Heath	Egypt	15
	▲ Key: Russia (2 items)			
	4	Oprah Stevens	Russia	8
	9	Noah Neiman	Russia	8

The following code example demonstrates how to apply Grouping in FlexGrid in C# and XAML. The example uses the data source, **Customer.cs** created in the [Quick start](#) section.

1. Add a new Forms XAML Page, Grouping.xaml, to your portable project.
2. To initialize a FlexGrid control and enabling grouping in XAML, modify the markup between the `<ContentPage>` `</ContentPage>` tags and inside the `<StackLayout>` `</StackLayout>` tags as illustrated in the code below.

#### Example Title

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="App3.Page1"
xmlns:cl="clr-namespace:C1.Xamarin.Forms.Grid;assembly=C1.Xamarin.Forms.Grid" >
  <ContentPage.ToolbarItems>
    <ToolbarItem x:Name="tb_Collapse" Text="Collapse"></ToolbarItem>
  </ContentPage.ToolbarItems>
  <StackLayout>
    <Grid VerticalOptions="FillAndExpand">
      <cl:FlexGrid x:Name="grid" AutoGenerateColumns="True"/>
    </Grid>
  </StackLayout>
</ContentPage>
```

3. In the Solution Explorer, expand the Grouping.xaml node and open the Merging.xaml.cs to view the C# code.
4. Add the following code in the Grouping class constructor to apply grouping to the column Country in the FlexGrid control.

#### C#

```
C1CollectionView<Customer> colView;
Func<Customer, string> selector;
public Grouping()
{
    InitializeComponent();
```

```
        this.tb_Collapse.Command = new Command((s) => {
            {
                this.grid.CollapseGroups();
            }
        });
        UpdateColView();
    }
    private async void UpdateColView()
    {
        var data = Customer.GetCustomerList(100);
        colView = new ClCollectionView<Customer>(data);
        await colView.GroupAsync(c => c.Country);
        this.grid.ItemsSource = colView;
    }
```

## Merging Cells

**FlexGrid** allows you to merge cells, making them span multiple rows or columns. This capability enhances the appearance and clarity of the data displayed on the grid. The effect of these settings is similar to the HTML `<ROWSPAN>` and `<COLSPAN>` tags.

To enable cell merging, you must do two things:

1. Set the grid's [AllowMerging](#) property to some value other than **None** in XAML.
2. If you wish to merge columns, set the [AllowMerging](#) property to **true** in C# for each column that you would like to merge. If you wish to merge rows, set the [AllowMerging](#) property to **true** in C# for each row that you would like to merge.

Merging occurs if the adjacent cells contain the same non-empty string. There is no method to force a pair of cells to merge. Merging occurs automatically based on the cell contents. This makes it easy to provide merged views of sorted data, where values in adjacent rows present repeated data.

Cell merging has several possible uses. For instance, you can use this feature to create merged table headers, merged data views, or grids where the text spills into adjacent columns.

The image given below shows FlexGrid control with merged cells in Country column.

Merged Cells



	ID	Name	Country	Country II	Active	First
	0	Mark Paulson	Italy	22	<input type="checkbox"/>	Mark
	1	Ed Jammers	Thailand	20	<input checked="" type="checkbox"/>	Ed
	2	Karl Ulam		20	<input type="checkbox"/>	Karl
	3	Herb Jammers	Indonesia	3	<input type="checkbox"/>	Herb
	4	Dan Ambers	Brazil	4	<input type="checkbox"/>	Dan
	5	Steve Evers	Indonesia	3	<input type="checkbox"/>	Steve
	6	Zeb Neiman	United Kingdor	21	<input type="checkbox"/>	Zeb
	7	Larry Frommer	China	0	<input checked="" type="checkbox"/>	Larry
	8	Fred Heath	Myanmar	23	<input checked="" type="checkbox"/>	Fred

The following code example demonstrates how to apply merging in FlexGrid control in C# and XAML. The example uses the data source class, **Customer.cs** created in the [Quick start](#).

1. Add a new Forms XAML Page, Merging.xaml to your portable project.
2. To initialize a FlexGrid control and enabling merging in XAML, modify the markup between the <ContentPage> </ContentPage> tags and inside the <StackLayout> </StackLayout> tags as illustrated in the code below.

#### In XAML

##### XAML

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:cl="clr-namespace:Cl.Xamarin.Forms.Grid;assembly=Cl.Xamarin.Forms.Grid"
    x:Class="CellMerging.Merging" x:Name="page">
    <Grid RowSpacing="0">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition />
        </Grid.RowDefinitions>
        <cl:FlexGrid x:Name="grid" AllowMerging="Cells"/>
    </Grid>
</ContentPage>
```

3. In the **Solution Explorer**, expand the Merging.xaml node and open the Merging.xaml.cs to view the C# code.
4. Add the following code in the Merging class constructor to apply merging to the column Country in the FlexGrid control.

#### In Code

C#

```
public partial class Merging : ContentPage
{
    public Merging()
    {
        InitializeComponent();
        var data = Customer.GetCustomerList(100);
        grid.ItemsSource = data;
        grid.Columns["Country"].AllowMerging = true;
    }
}
```

## Resizing Columns

FlexGrid's [AllowResizing](#) property lets a user resize the column by simply touching and dragging the handle between two columns. By default, its value is true. To allow resizing for specific columns, you can set the **AllowResizing** property for a column instead of the entire FlexGrid.

You can set the **AllowResizing** property of FlexGrid to **false** in case you want to restrict resizing at runtime.

C#

```
grid.AllowResizing = false;
```

The resizing functionality for columns is useful when a user wants to add data in FlexGrid. The user can simply resize the column as per the requirement directly on the device without requiring to change or set the width in code.

The following code example demonstrates how to set the **AllowResizing** property in FlexGrid. The example uses sample created in the [Quick start](#) section.

C#

```
grid.AllowResizing = true;
```

## Row Details

The FlexGrid control allows you to create a hierarchical grid by adding a row details section to each row. Adding a row details sections allows you to group some data in a collapsible template and present only a summary of the data for each row. The row details section is displayed only when the user taps a row.

The image given below shows a FlexGrid with row details section added to each row.

Collapsible Template  
to show Row Details

	Id	First Name	Last Name
-	0	Paul	Griswold
Country: Pakistan City: Islamabad Address: 399 Park BLVD PostalCode: 10174			
+	1	Herb	Danson
-	2	Ted	Stevens
Country: India City: Kolkata Address: 435 Fake ST			

The following code example demonstrates how to add row details section to FlexGrid control in C# and XAML. This example uses a new data source class, **Customer.cs**.

1. Create a new data source class, **Customer.cs** and add it to your portable project.
2. Add the following code to the Customer.cs class.

C#

```
public class Customer :
    INotifyPropertyChanged,
    IEditableObject
{
    #region ** fields

    int _id, _countryId, _orderCount;
    string _first, _last;
    string _address, _city, _postalCode, _email;
    bool _active;
    DateTime _lastOrderDate;
    double _orderTotal;

    static Random _rnd = new Random();
    static string[] _firstNames =
        "Andy|Ben|Paul|Herb|Ed|Ted|Zeb".Split('|');
    static string[] _lastNames =
        "Ambers|Danson|Evers|Frommer|Griswold|Orsted|Stevens".Split('|');
    static KeyValuePair<string, string[]>[] _countries = "China-
        Beijing,Chongqing,Chaohu|India-New Delhi,Mumbai,Delhi,Chennai,Kolkata|United
        States-Washington,New York,Los Angeles|Indonesia-Jakarta,South Tangerang|Brazil-
        Brasilia,Sao Paulo,Rio de Janeiro,|Pakistan-
        Islamabad,Karachi,Lahore,Quetta|Russia-Moscow,Saint
        Petersburg,Novosibirsk,Rostov-na-Donu|Japan-Tokyo,Yokohama,Osaka,Saitama|Mexico-
        Mexico City,Guadalajara,Monterrey".Split('|').Select(str => new
```

```

KeyValuePair<string, string[]>(str.Split('-').First(), str.Split('-')
.Skip(1).First().Split(','))).ToArray();
    static string[] _emailServers = "gmail|yahoo|outlook|aol".Split('|');
    static string[] _streetNames =
"Main|Broad|Grand|Panoramic|Green|Golden|Park|Fake".Split('|');
    static string[] _streetTypes = "ST|AVE|BLVD".Split('|');
    static string[] _streetOrientation = "S|N|W|E|SE|SW|NE|NW".Split('|');

#endregion

#region ** initialization

public Customer()
    : this(_rnd.Next(10000))
{
}

public Customer(int id)
{
    Id = id;
    FirstName = GetRandomString(_firstNames);
    LastName = GetRandomString(_lastNames);
    Address = GetRandomAddress();
    CountryId = _rnd.Next() % _countries.Length;
    var cities = _countries[CountryId].Value;
    City = GetRandomString(cities);
    PostalCode = _rnd.Next(10000, 99999).ToString();
    Email = string.Format({0}@{1}.com,

(FirstName + LastName.Substring(0, 1)).ToLower(),

GetRandomString(_emailServers));
    LastOrderDate = DateTime.Today.AddDays(-_rnd.Next(1,
365)).AddHours(_rnd.Next(0, 24)).AddMinutes(_rnd.Next(0, 60));
    OrderCount = _rnd.Next(0, 100);
    OrderTotal = Math.Round(_rnd.NextDouble() * 10000.00, 2);
    Active = _rnd.NextDouble() >= .5;
}

#endregion

#region ** object model

public int Id
{
    get { return _id; }
    set
    {
        if (value != _id)
        {
            _id = value;

```

```
        OnPropertyChanged("Id");
    }
}

public string FirstName
{
    get { return _first; }
    set
    {
        if (value != _first)
        {
            _first = value;
            OnPropertyChanged("FirstName");
            OnPropertyChanged("Name");
        }
    }
}

public string LastName
{
    get { return _last; }
    set
    {
        if (value != _last)
        {
            _last = value;
            OnPropertyChanged("LastName");
            OnPropertyChanged("Name");
        }
    }
}

public string Address
{
    get { return _address; }
    set
    {
        if (value != _address)
        {
            _address = value;
            OnPropertyChanged("Address");
        }
    }
}

public string City
{
    get { return _city; }
    set
    {
```

```
        if (value != _city)
        {
            _city = value;
            OnPropertyChanged("City");
        }
    }

    public int CountryId
    {
        get { return _countryId; }
        set
        {
            if (value != _countryId && value > -1 && value <
                _countries.Length)
            {
                _countryId = value;
                //_city = _countries[_countryId].Value.First();
                OnPropertyChanged("CountryId");
                OnPropertyChanged("Country");
                OnPropertyChanged("City");
            }
        }
    }

    public string PostalCode
    {
        get { return _postalCode; }
        set
        {
            if (value != _postalCode)
            {
                _postalCode = value;
                OnPropertyChanged("PostalCode");
            }
        }
    }

    public string Email
    {
        get { return _email; }
        set
        {
            if (value != _email)
            {
                _email = value;
                OnPropertyChanged("Email");
            }
        }
    }
}
```



```
public DateTime LastOrderDate
{
    get { return _lastOrderDate; }
    set
    {
        if (value != _lastOrderDate)
        {
            _lastOrderDate = value;
            OnPropertyChanged("LastOrderDate");
        }
    }
}

public int OrderCount
{
    get { return _orderCount; }
    set
    {
        if (value != _orderCount)
        {
            _orderCount = value;
            OnPropertyChanged("OrderCount");
        }
    }
}

public double OrderTotal
{
    get { return _orderTotal; }
    set
    {
        if (value != _orderTotal)
        {
            _orderTotal = value;
            OnPropertyChanged("OrderTotal");
        }
    }
}

public bool Active
{
    get { return _active; }
    set
    {
        if (value != _active)
        {
            _active = value;
            OnPropertyChanged("Active");
        }
    }
}
```

```
public string Name
{
    get { return string.Format("{0} {1}", FirstName, LastName); }
}

public string Country
{
    get { return _countries[_countryId].Key; }
}

public double OrderAverage
{
    get { return OrderTotal / (double)OrderCount; }
}

#endregion

#region ** implementation

// ** utilities
static string GetRandomString(string[] arr)
{
    return arr[_rnd.Next(arr.Length)];
}
static string GetName()
{
    return string.Format("{0} {1}", GetRandomString(_firstNames),
GetRandomString(_lastNames));
}

// ** static list provider
public static ObservableCollection<Customer> GetCustomerList(int count)
{
    var list = new ObservableCollection<Customer>();
    for (int i = 0; i < count; i++)
    {
        list.Add(new Customer(i));
    }
    return list;
}

private static string GetRandomAddress()
{
    if (_rnd.NextDouble() > 0.9)
        return string.Format("{0} {1} {2} {3}", _rnd.Next(1, 999),
GetRandomString(_streetNames), GetRandomString(_streetTypes),
GetRandomString(_streetOrientation));
    else
        return string.Format("{0} {1} {2}", _rnd.Next(1, 999),
GetRandomString(_streetNames), GetRandomString(_streetTypes));
}
```

```
    }

    // ** static value providers
    public static KeyValuePair<int, string>[] GetCountries() { return
_countries.Select((p, index) => new KeyValuePair<int, string>(index,
p.Key)).ToArray(); }
    public static string[] GetFirstNames() { return _firstNames; }
    public static string[] GetLastNames() { return _lastNames; }

    #endregion

    #region ** INotifyPropertyChanged Members

    // interface allows bounds controls to react to changes in data objects.
    public event PropertyChangedEventHandler PropertyChanged;

    private void OnPropertyChanged(string propertyName)
    {
        OnPropertyChanged(new PropertyChangedEventArgs(propertyName));
    }

    protected void OnPropertyChanged(PropertyChangedEventArgs e)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, e);
    }

    #endregion

    #region IEditableObject Members

    // interface allows transacted edits

    Customer _clone;
    public void BeginEdit()
    {
        _clone = (Customer) this.MemberwiseClone();
    }

    public void EndEdit()
    {
        _clone = null;
    }

    public void CancelEdit()
    {
        if (_clone != null)
        {
            foreach (var p in this.GetType().GetRuntimeProperties())
            {
                if (p.CanRead && p.CanWrite)
```

```

        {
            p.SetValue(this, p.GetValue(_clone, null), null);
        }
    }
}

#endregion
}

```

3. Add a new Forms Xaml Page, RowDetails to your portable project.
4. To initialize a FlexGrid control and adding row details section, modify the markup between the <ContentPage> </ContentPage> tags as illustrated in the code below.

### In XAML

#### XAML

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:cl="clr-
namespace:C1.Xamarin.Forms.Grid;assembly=C1.Xamarin.Forms.Grid"
             x:Class="FlexGridRowDetails.RowDetails" x:Name="page">
    <Grid RowSpacing="0">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition />
        </Grid.RowDefinitions>
        <cl:FlexGrid x:Name="grid" Grid.Row="3" AutoGenerateColumns="False">
            <cl:FlexGrid.Columns>
                <cl:GridColumn Binding="Id" Width="Auto"/>
                <cl:GridColumn Binding="FirstName" Width="*/>
                <cl:GridColumn Binding="LastName" Width="*/>
            </cl:FlexGrid.Columns>
            <cl:FlexGrid.Behaviors>
                <cl:FlexGridDetailProvider x:Name="details" Height="170">
                    <DataTemplate>
                        <Grid>
                            <Grid.RowDefinitions>
                                <RowDefinition />
                                <RowDefinition />
                                <RowDefinition />
                                <RowDefinition />
                            </Grid.RowDefinitions>
                            <Grid.ColumnDefinitions>
                                <ColumnDefinition Width="Auto" />
                                <ColumnDefinition />
                            </Grid.ColumnDefinitions>
                            <Label Text="Country:"/>
                            <Label Text="{Binding Country}" Grid.Column="1"/>
                            <Label Text="City:" Grid.Row="1"/>

```

```

        <Label Text="{Binding City}" Grid.Row="1" Grid.Column="1"/>
        <Label Text="Address:" Grid.Row="2"/>
        <Label Text="{Binding Address}" Grid.Row="2" Grid.Column="1"/>
        <Label Text="PostalCode:" Grid.Row="3"/>
        <Label Text="{Binding PostalCode}" Grid.Row="3" Grid.Column="1"/>
    </Grid>
</DataTemplate>
</cl:FlexGridDetailProvider>
</cl:FlexGrid.Behaviors>
</cl:FlexGrid>
</Grid>
</ContentPage>

```

5. In the **Solution Explorer**, expand the RowDetails.xaml node and open the Merging.xaml.cs to open the C# code behind.
6. Add the following code in the RowDetails class constructor to add row details section in the FlexGrid control.

### In Code

```

C#
public partial class RowDetails : ContentPage
{
    public RowDetails()
    {
        InitializeComponent();
        var data = Customer.GetCustomerList(1000);
        grid.ItemsSource = data;
    }
}

```

### Customizing expand and collapse buttons

You can also customize the expand and collapse buttons by handling the OnRowHeaderLoading event, and setting the ExpandButton.CheckedImageSource and UncheckedImageSource properties as illustrated in the following code example.

```

C#
private void OnRowHeaderLoading(object sender, GridRowHeaderLoadingEventArgs e)
{
    e.ExpandButton.CheckedImageSource = ImageSource.FromResource("collapse.png");
    e.ExpandButton.UncheckedImageSource = ImageSource.FromResource("expand.png");
}

```

## Sorting

FlexGrid allows you to sort grid's data using the following methods:

- **Sorting at Runtime:** The [AllowSorting](#) property of FlexGrid allows users to sort the grid. On setting the **AllowSorting** property to **true**, a user can simply tap a column's header to sort the grid by that column at runtime.
- **Sorting through Code:** Using CollectionView interface, you can enable sorting through code in C# or XAML.

To enable **sorting**, add one or more objects to the [CollectionView.SortDescriptions](#) property.

### Sorting at runtime

The image below shows how the FlexGrid appears after sorting is applied to the grid by tapping header of the column, **Country**.

#### In Code

	ID	Name	Country ▲
	18	Larry Richards	Bangladesh
	51	Andy Stevens	Bangladesh
	14	Ben Quaid	Brazil
	36	Steve Neiman	Brazil
	39	Rich Orsted	Brazil
	86	Ed Heath	China
	91	Ulrich Krause	China

C#

```
grid.AllowSorting = true;
```

#### In XAML

XAML

```
<cl:FlexGrid AllowSorting="True">
```

### Sorting through Code

The image below shows how the FlexGrid appears after sorting is applied to the column **Name**.

	ID	Name	Country	CountryID
	0	Ben Cole	United States	2
	3	Ben Orsted	Russia	8
	8	Herb Heath	Mexico	10
	1	Herb Stevens	Brazil	4
	6	Jack Heath	Thailand	20
	2	Mark Jammers	Vietnam	12
	4	Noah Heath	Egypt	15

The following code example demonstrates how to sort a FlexGrid control in **C# and XAML**. This example uses the data source, **Customer.cs** created in the [Quick start](#) section.



Import the following references in the class:

```
using Xamarin.Forms;
using Cl.CollectionView;
using Cl.Xamarin.Forms.Grid;
```

## In Code

C#

```
public static FlexGrid GetGrid()
{
    var dataCollection = Customer.GetCustomerList(10);
    ClCollectionView<Customer> cv = new ClCollectionView<Customer>
(dataCollection);
    var sort = cv.SortDescriptions.FirstOrDefault(sd => sd.SortPath == "Name");
    var direction = sort != null ? sort.Direction : SortDirection.Descending;
    cv.SortAsync(x => x.Name, direction == SortDirection.Ascending ?
SortDirection.Descending : SortDirection.Ascending);
    FlexGrid _grid = new FlexGrid();
    _grid.ItemsSource = cv;
    _grid.VerticalOptions = LayoutOptions.FillAndExpand;
    return _grid;
    cv.SortChanged += cv_SortChanged;
}
static void cv_SortChanged(object sender, EventArgs e)
{
    throw new NotImplementedException();
}
```

## Sorting a column by a different field

By default, sorting is applied on the bound field. However, you can sort a column by a different field. All you need to do is set the **SortMemberPath** property to the column by which you want the grid to be sorted. For example, the following column is bound to "FullName" but sorts by "LastName".

```
C#  
  
column.Binding = "FullName";  
column.SortMemberPath = "LastName";
```

## Selecting Cells

The [SelectionMode](#) property of the FlexGrid allows you to define the selection mode of the cells by setting its value to **Row**, **Cell**, **CellRange**, or **RowRange**. You can also disable the selection by setting the selection mode to **GridSelectionModeNone**, which is its default value.

The image below shows how the FlexGrid appears after the `SelectionMode` property is set to `CellRange`.

	ID	Name	Country	CountryID
	0	Ulrich Bishop	Japan	9
	1	Oprah Ulam	Japan	9
	2	Karl Trask	India	1
	3	Vic Paulson	United States	2
	4	Zeb Orsted	Pakistan	5

The following code example demonstrates how to choose selection modes in FlexGrid. The example uses the sample created in the [Quick start](#) section.

### In Code

```
C#  
  
grid.SelectionMode = GridSelectionMode.CellRange;
```

### In XAML

```
XAML  
  
<Grid VerticalOptions="FillAndExpand">  
  <cl:FlexGrid SelectionMode="CellRange" x:Name="grid" HeadersVisibility="All"/>  
</Grid>
```

## Star and Auto Sizing

FlexGrid's Star and Auto Sizing feature allows you to set the width of columns in the FlexGrid by using both the star and auto sizing system instead of explicitly setting width of each column. Star sizing determines the width of each column automatically with respect to available screen space and in relation with other columns to avoid horizontal scrolling and display the entire data on the screen. Auto sizing determines the width of a column based on the size of



the content. You can use both sizing options to implement flexible layouts with the FlexGrid. The sample below uses Star and Auto sizing to specify the width of columns.

In this example, the grid has four columns. The width of **first** and **third** columns are set to \*, and the width of the **second** column is set to 2\*. The **fourth** column is set to **Auto**.

The image below shows how the FlexGrid appears, after star sizing is applied to columns' width:

	ID	First	Last	Weight
	0	Andy	Krause	59.0
	1	Oprah	Lehman	53.8
	2	Ed	Neiman	66.1
	3	Xavier	Jammei	83.1
	4	Herb	Cole	83.1
	5	Gil	Myers	85.6
	6	Xavier	Myers	93.2
	7	Larry	Stevens	95.1

The following code example demonstrates star sizing in FlexGrid in XAML. The example uses the sample created in the [Quick Start](#) section.

#### In Code

##### XAML

```
<Grid>
  <cl:FlexGrid x:Name="grid" AutoGenerateColumns="False" AllowResizing="None"
Grid.Row="1">
  <cl:FlexGrid.Columns>
    <cl:GridColumn Binding="ID" Width="*" />
    <cl:GridColumn Binding="First" Width="2*" />
    <cl:GridColumn Binding="Last" Width="*" />
    <cl:GridColumn Binding="Weight" Width="Auto" Format="N1" />
  </cl:FlexGrid.Columns>
</cl:FlexGrid>
</Grid>
```

## Wordwrap

FlexGrid's `WordWrap` property allows a user to display multi-line text in a single cell of the grid. To enable word wrapping in a FlexGrid column, set the value of `WordWrap` property to **true**. By default, its value is set to **False**. The **WordWrap** property determines whether the grid should automatically break long strings containing multiple words and special characters such as spaces in multiple lines. Multiple line text can be displayed in both fixed and scrollable cells.

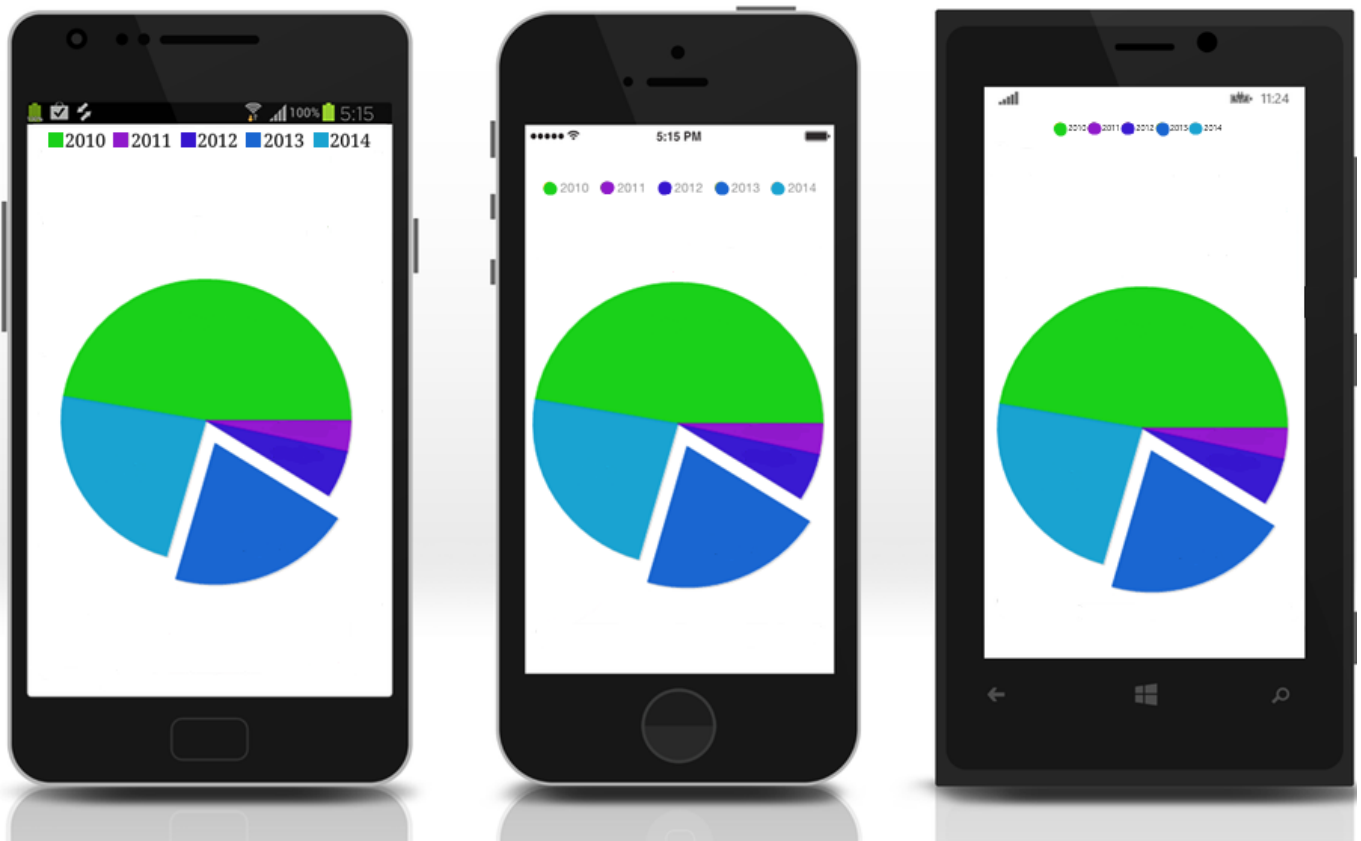
```
C#  
  
//to enable word wrap  
grid.Columns[1].WordWrap = true;
```

To resize the row heights automatically, you can then call the `AutoSizeRow` and `AutoSizeRows` methods as illustrated in the code below.

```
C#  
  
// resize a single row  
grid.AutoSizeRow(1);  
  
// resize all rows  
grid.AutoSizeRows(0, grid.Rows.Count - 1);
```

## FlexPie

**FlexPie** allows you to create customized pie charts that represent a series as slices of a pie, wherein the arc length of each slice depicts the value represented by that slice. These charts are commonly used to display proportional data such as percentage cover. The multi-colored slices make pie charts easy to understand and usually the value represented by each slice is displayed with the help of labels.



## Key Features

- **Touch Based Labels:** Display values using touch based labels.
- **Exploding and Donut Pie Charts:** Use simple FlexPie properties to convert it into an exploding pie chart or a donut pie chart.

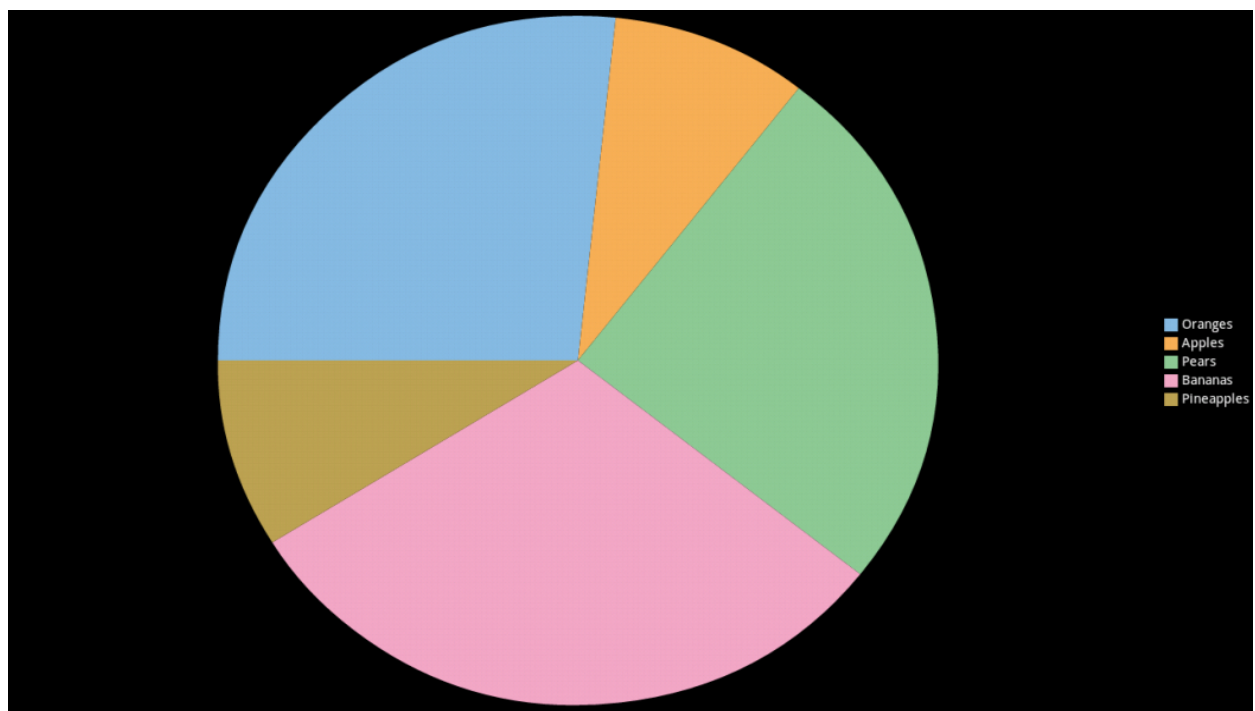
## Quick Start: Add data to FlexPie

This section describes how to add a [FlexPie](#) control to your portable or shared app and add data to it. For information on how to add Xamarin components in C# or XAML, see [Adding Xamarin Components using C#](#) or [Adding XamarinComponents using XAML](#).

This topic comprises of three steps:

- **Step 1: Create a data source for FlexPie**
- **Step 2: Add a FlexPie control**
- **Step 3: Run the Project**

The following image shows how the FlexPie appears after completing the steps above:



### Step 1: Create a data source for FlexPie

The following classes serve as a data source for the FlexPie control:

```
C#  
  
class FlexPieDataSource  
{  
    private List<FruitEntity> entityList;  
  
    public List<FruitEntity> Data  
    {  
        get { return entityList; }  
    }  
}
```

```
    }

    public FlexPieDataSource()
    {
        entityList = new List<FruitEntity>();
        string[] fruits = new string[] { "Oranges", "Apples", "Pears", "Bananas",
"Pineapples" };
        Random random = new Random();
        for (int i = 0; i < fruits.Length; i++)
        {
            decimal value = (decimal)random.NextDouble() * 100;
            entityList.Add(new FruitEntity(fruits[i], value));
        }
    }
}

class FruitEntity
{
    public string Name { get; set; }
    public decimal Value { get; set; }

    public FruitEntity(string name, decimal value)
    {
        this.Name = name;
        this.Value = value;
    }
}
```

**Back to Top**

## Step 2: Add FlexPie control

Complete the following steps to initialize a FlexPie control in C# or XAML.

### In Code

1. Add a new class (for example QuickStart.cs) to your Portable or Shared project and include references as shown below:

C#

```
using Xamarin.Forms;
using Cl.Xamarin.Forms.Chart;
```

2. Instantiate a FlexPie control in a new method, GetFlexPie().

C#

```
public static FlexPie GetFlexPie()
{
    FlexPie chart = new FlexPie();
    FlexPieDataSource ds = new FlexPieDataSource();
    chart.BindingName = "Name";
    chart.Binding = "Value";
    chart.ItemsSource = ds.Data;
    return chart;
}
```

```
}
```

## In XAML

1. Add a new Forms Xaml Page (for example FlexPieQuickStart.xaml) to your Portable or Shared project and modify the <ContentPage> tag to include the following reference:

### XAML

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:cl="clr-
namespace:Cl.Xamarin.Forms.Chart;assembly=Cl.Xamarin.Forms.Chart"
             x:Class="QuickstartChart.FlexPieQuickStart">
```

2. Initialize a FlexPie control by adding the markup for the control between the <ContentPage> </ContentPage> tags inside the <StackLayout> </StackLayout> tags, as shown below:

### XAML

```
<StackLayout>
    <cl:FlexPie x:Name="chart" ItemsSource="{Binding Data}" BindingName="Name"
                Binding="Value" Grid.Row="1" Grid.ColumnSpan="2"
VerticalOptions="FillAndExpand">
</cl:FlexPie>
</StackLayout>
```

3. In the **Solution Explorer**, expand the FlexPieQuickStart.xaml node and open FlexPieQuickStart.xaml.cs to open the C# code behind.
4. In the FlexPieQuickStart() class constructor, set a new instance of FlexPieDataSource as a BindingContext for the FlexPie.

The following code shows what the FlexPieQuickStart() class constructor looks like after completing this step.

### C#

```
public FlexPieQuickStart()
{
    InitializeComponent();
    chart.BindingContext = new FlexPieDataSource();
}
```

## Back to Top

## Step 3: Run the Project

1. In the **Solution Explorer**, double click App.xaml.cs to open it.
2. Complete the following steps to display the FlexPie control.
  - **To return a C# class:** In the class constructor App(), set a new ContentPage as the MainPage and assign the control to the ContentPage's Content by invoking the GetFlexPie() method defined in the previous procedure, **Step 2: Add a FlexPie Control**.

The following code shows the class constructor App() after completing steps above.

### C#

```
public App()
{
```

```
// The root page of your application
MainPage = new ContentPage
{
    Content = QuickStart.GetFlexPie()
};
}
```

- **To return a Forms Xaml Page:** In the class constructor App(), set the Forms Xaml Page FlexPieQuickStart as the MainPage.

The following code shows the class constructor App() after completing this step.

```
C#
public App()
{
    // The root page of your application
    MainPage = new FlexPieQuickStart();
}
```

3. Some additional steps are required to run the iOS and UWP apps:

- **iOS App:**

1. In the **Solution Explorer**, double click AppDelegate.cs inside YourAppName.iOS project, to open it.
2. Add the following code to the FinishedLaunching() method.

```
C#
Cl.Xamarin.Forms.Chart.Platform.iOS.FlexPieRenderer.Init();
```

- **UWP App:**

1. In the **Solution Explorer**, expand MainPage.xaml.
2. Double click MainPage.xaml.cs to open it.
3. Add the following code to the class constructor.

```
C#
Cl.Xamarin.Forms.Chart.Platform.UWP.FlexPieRenderer.Init();
```

4. Press **F5** to run the project.

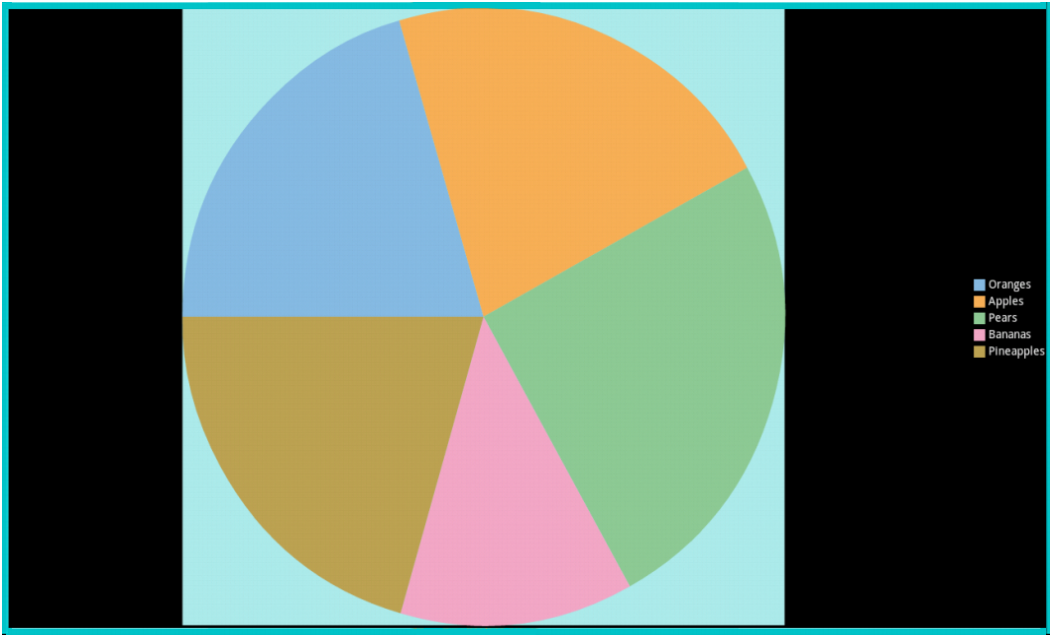
**Back to Top**

## Features

### Customize Appearance

Although Xamarin controls match the native controls on all three platforms by default and are designed to work with both: light and dark themes available on all platforms. But, there are several properties to customize the appearance of the FlexPie control. You can change the background color of the FlexPie and the plot area.

The image below shows the how the FlexPie appears after these properties have been set.



In Code

The following code example demonstrates how to set these properties in C#. This example uses the sample created in the [Quick Start](#) section.

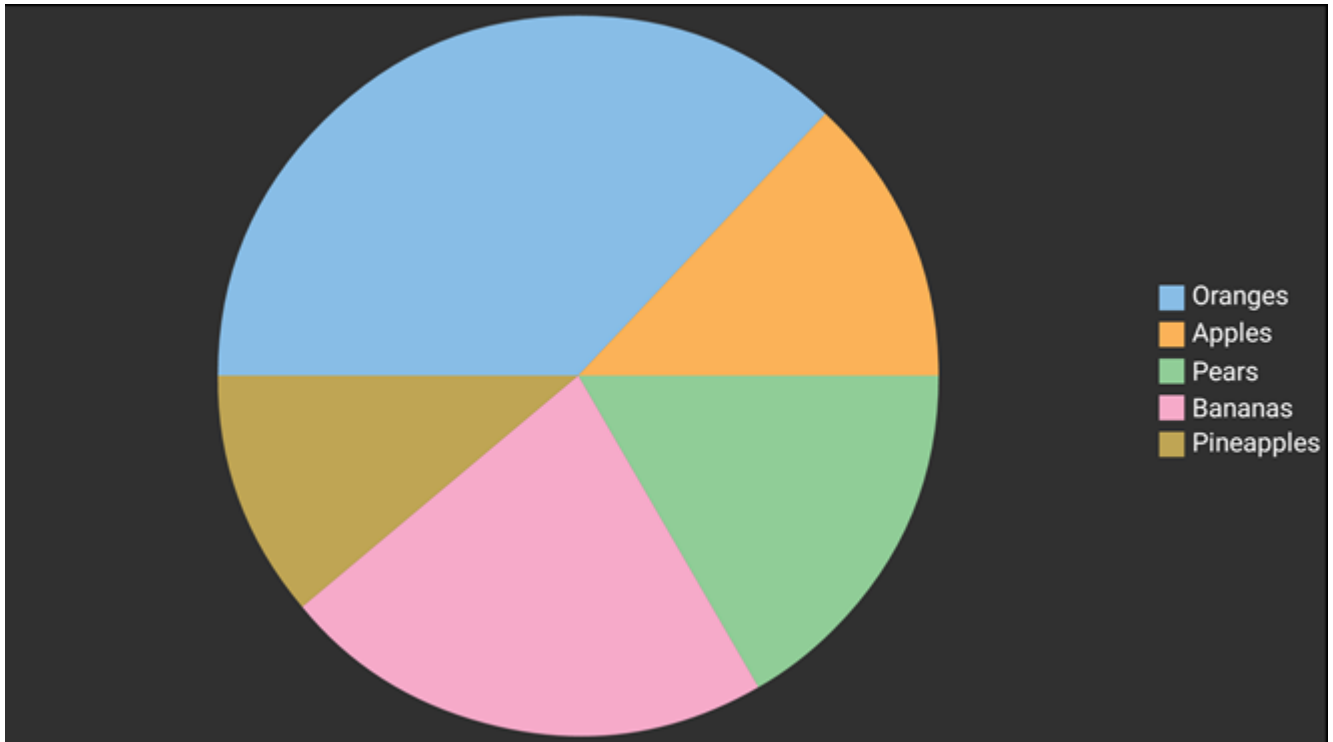
```
C#  
  
//Customize chart plot area  
ChartStyle s = new ChartStyle();  
s.Fill = Color.Black;  
s.StrokeThickness = 0;  
chart.PlotStyle = s;  
  
//Set background color  
chart.BackgroundColor = Color.Aqua;
```

Data Binding

You can bind the [FlexPie](#) control to data by using the following set of properties for data binding:

Property	Description
<a href="#">Binding</a>	Property for binding values
<a href="#">BindingName</a>	Property for binding with items appearing in the legend
<a href="#">ItemsSource</a>	Property for binding with collection of items

The image given below shows a FlexPie control with data bound to values and items appearing in the legend.



### In Code

The following code examples illustrate how to set Data Binding in FlexPie control. The example uses the sample created in the [Quick Start](#) section.

1. Set the BindingName property to Name within the GetFlexPie () method in QuickStart class file as illustrated in the following code.

```
C#  
  
chart.BindingName = "Name";  
chart.Binding = "Value";  
chart.ItemsSource = new Object[]  
{  
    new {Value=100, Name="Oranges"},  
    new {Value=35, Name="Apples"},  
    new {Value=45, Name="Pears"},  
    new {Value=60, Name="Bananas"},  
    new {Value=30, Name="Pineapples"}  
};
```

### In XAML

You can also set the Binding through XAML as illustrated in the following code snippet.

```
XAML  
  
<cl:FlexPie x:Name="chart" BindingName="Name" Binding="Value">
```

## Data Labels

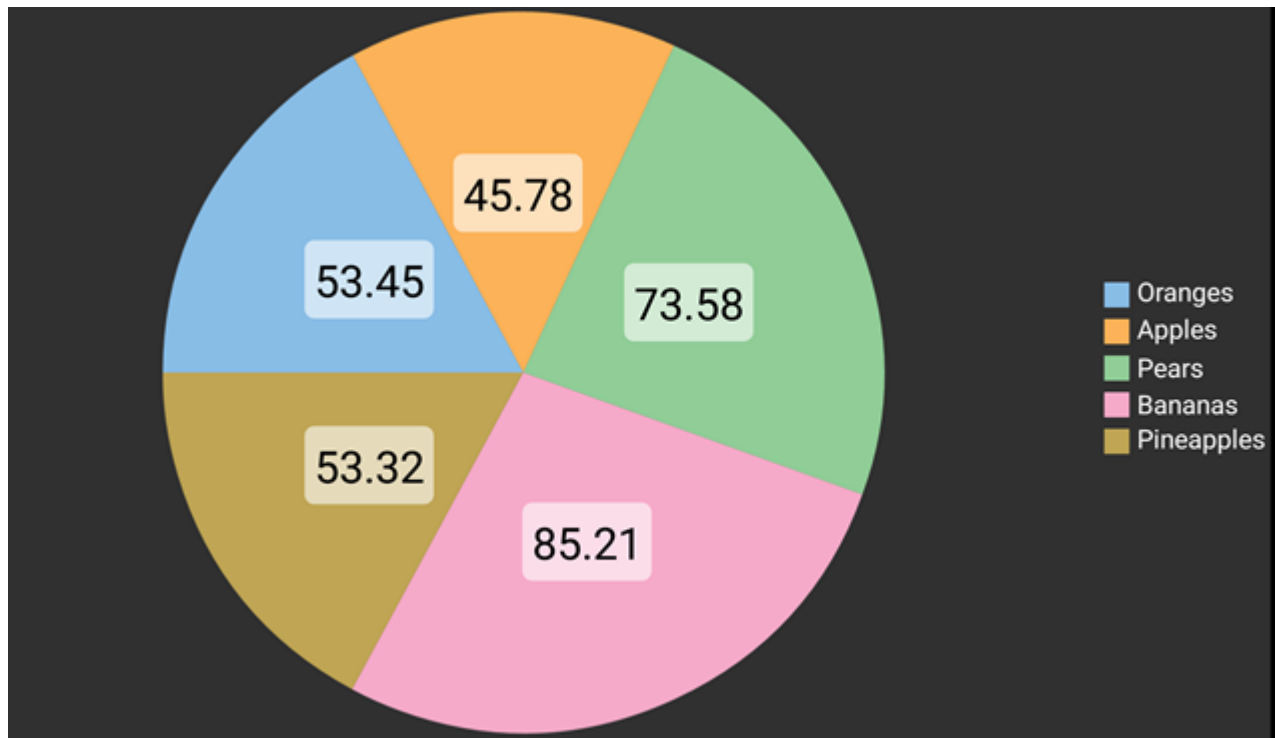
You can add static data labels in the FlexPie to show the exact values corresponding to each slice of the pie. You can



choose to set data labels at the following positions:

- **Inside** - data labels get displayed within the pie near the edge.
- **Outside** - data labels get displayed outside the pie along the edge.
- **Center** - data labels get displayed right in the center of pie slice.
- **None** - data labels are not visible (default value).

The following image shows a FlexPie with data labels displayed in the **Center**.



A data label can contain any amount of text and other UI elements. To display data labels in a FlexPie control, you must set the [PieDataLabel.Position](#) property and define the content template to display as the label.

### In XAML

To set the position and define the content template in XAML, add the following markup between the FlexPie control tags.

#### XAML

```
<cl:FlexPie.DataLabel>
    <cl:PieDataLabel Position="Center" Border="True" Content="{{value:F2}}"
        <cl:PieDataLabel.Style >
            <cl:ChartStyle Fill="#99FFFFFF" FontSize="18" ></cl:ChartStyle>
        </cl:PieDataLabel.Style>
    </cl:PieDataLabel>
</cl:FlexPie.DataLabel>
```

### In Code

To change the label position in code, add the following code.

#### C#

```
//Setting data labels position
```

```
chart.DataLabel.Position = PieLabelPosition.Outside;
```

The data context for the label is an instance of **PieDataPoint**, which exposes the following properties:

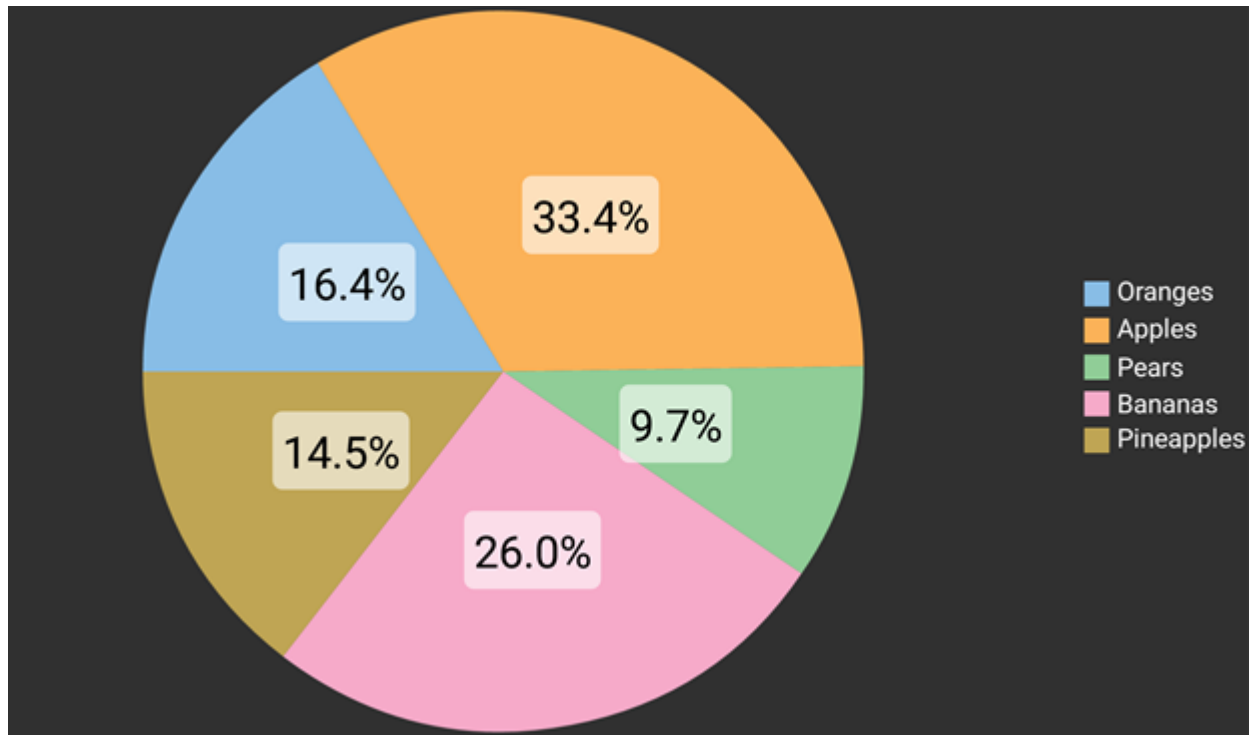
Parameter	Description
x	Show the X value of the data point.
y	Show the Y value of the data point.
value	Show the Y value of the data point.
name	Show the X value of the data point.
seriesName	Shows the name of the series.
pointIndex	Shows the index of the data point.
Percentage	Shows the percentage of a pie slice to the whole pie chart.

To display percentage values as the data labels, you would bind a Xamarin.Forms Label to 'Percentage' as shown below.

In XAML

```
XAML
<c1:FlexPie.DataLabel>
    <c1:PieDataLabel Position="Center" Border="True" Content="{
{Percentage}}">
        <c1:PieDataLabel.Style >
            <c1:ChartStyle Fill="#99FFFFFF" FontSize="18" >
</c1:ChartStyle>
        </c1:PieDataLabel.Style>
    </c1:PieDataLabel>
</c1:FlexPie.DataLabel>
```

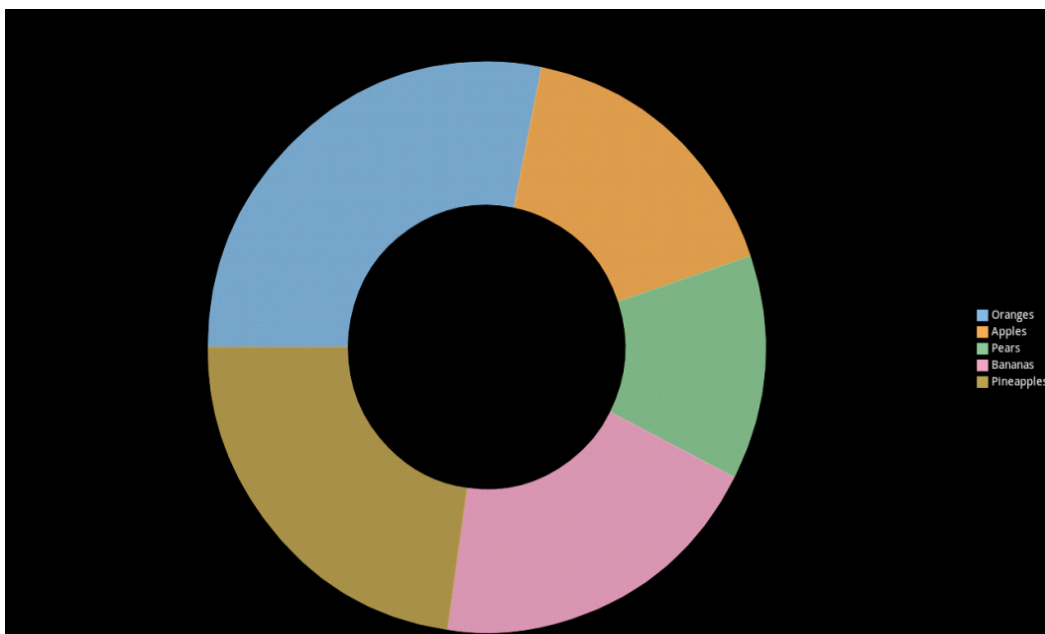
The following image shows a FlexPie displaying percentage values.



## Donut Pie Chart

The [InnerRadius](#) property can be used to leave a blank inner space in the FlexPie, creating a Donut Pie Chart. The blank space can be used to display additional data.

The following image shows a donut FlexPie.



The following code example demonstrates how to set this property in C#. This example uses the sample created in the [Quick Start](#) section.

### In Code

```
C#
```

```
//set donut chart  
chart.InnerRadius = 0.5;
```

## Exploded Pie Chart

The [Offset](#) property can be used to push the pie slices away from the center of the FlexPie, producing an exploded pie chart. This property accepts a decimal value to determine how far the pie slices should be pushed from the center.

The image below shows an exploded FlexPie.



The following code example demonstrates how to set this property in C#. This example uses the sample created in the [Quick Start](#) section.

### In Code

```
C#  
chart.Offset = 0.2;
```

[copyCode](#)

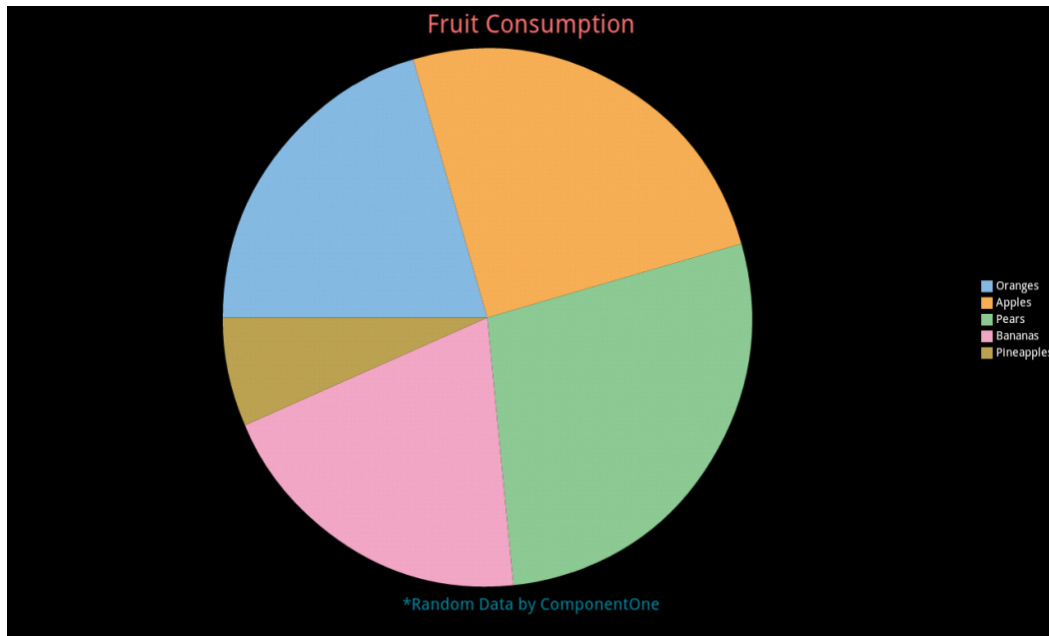
## Header and Footer

You can add a title to the FlexPie control by setting its **Header** property. Besides a title, you may also set a footer by setting the **Footer** property.

There are also some additional properties to customize header and footer text in a FlexPie.

- **HeaderAlignment** - Lets you set the alignment for header text.
- **FooterAlignment** - Lets you set the alignment for footer text.

The image below shows how the FlexPie appears after these properties have been set.



### In Code

The following code example demonstrates how to set these properties in C#. This example uses the sample created in the [Quick Start](#) section.

C#

```
//Set header and footer
chart.Header = "Fruit Consumption";
chart.HeaderAlignment = LayoutAlignment.Fill;

chart.Footer = "Random data by ComponentOne";
chart.FooterAlignment = LayoutAlignment.Fill;
```

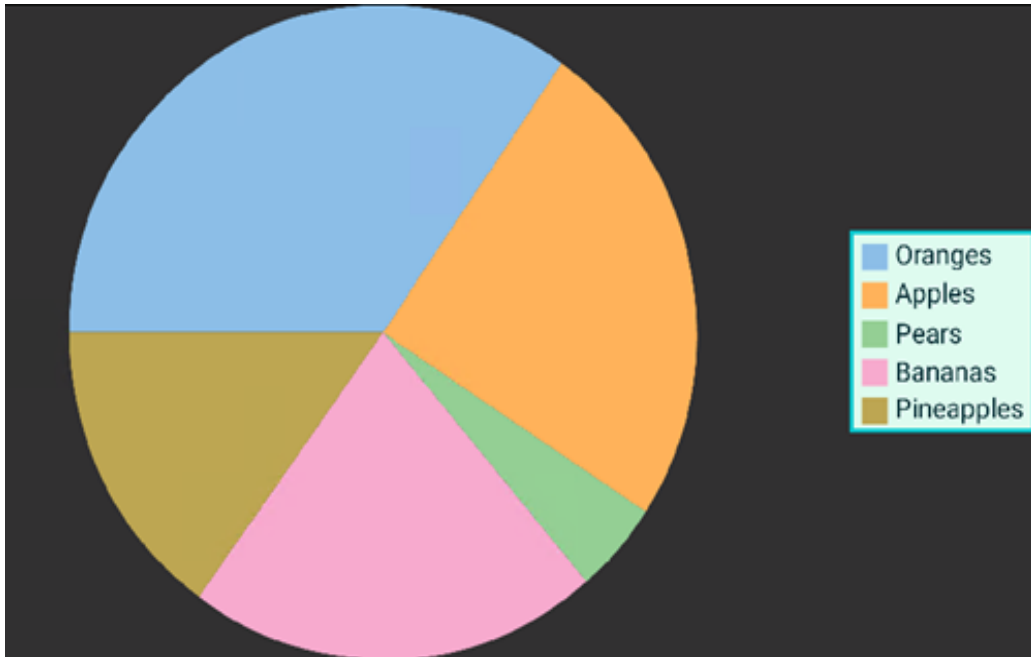
## Legend

FlexPie provides the option to display legend for denoting the type of the data presented in the pie slices. The position of legend is by default set to "Auto", which means the legend positions itself automatically depending on the real estate available on the device. This allows the pie to efficiently occupy the available space on the device. Users have the option to customize the appearance of the legend and enhance the visual appeal of the FlexPie control. You can also style the legend by setting its orientation through the [LegendOrientation](#) property, and adding a border through the [Stroke](#) property.



The legend automatically wraps when the Position property is set to Top, Bottom, Left or Right, Orientation is set to Horizontal and there is not enough screen real estate.

The image below shows how the FlexPie appears after these properties have been set.



### In Code

The following code example demonstrates how to set these properties in C#. This examples uses the sample created in the [Quick Start](#) section.

C#

```
chart.LegendPosition = ChartPositionType.Top;  
chart.LegendOrientation = Orientation.Horizontal;  
chart.LegendStyle.Stroke = Color.FromHex("#2D3047");  
chart.LegendStyle.StrokeThickness = 4;
```

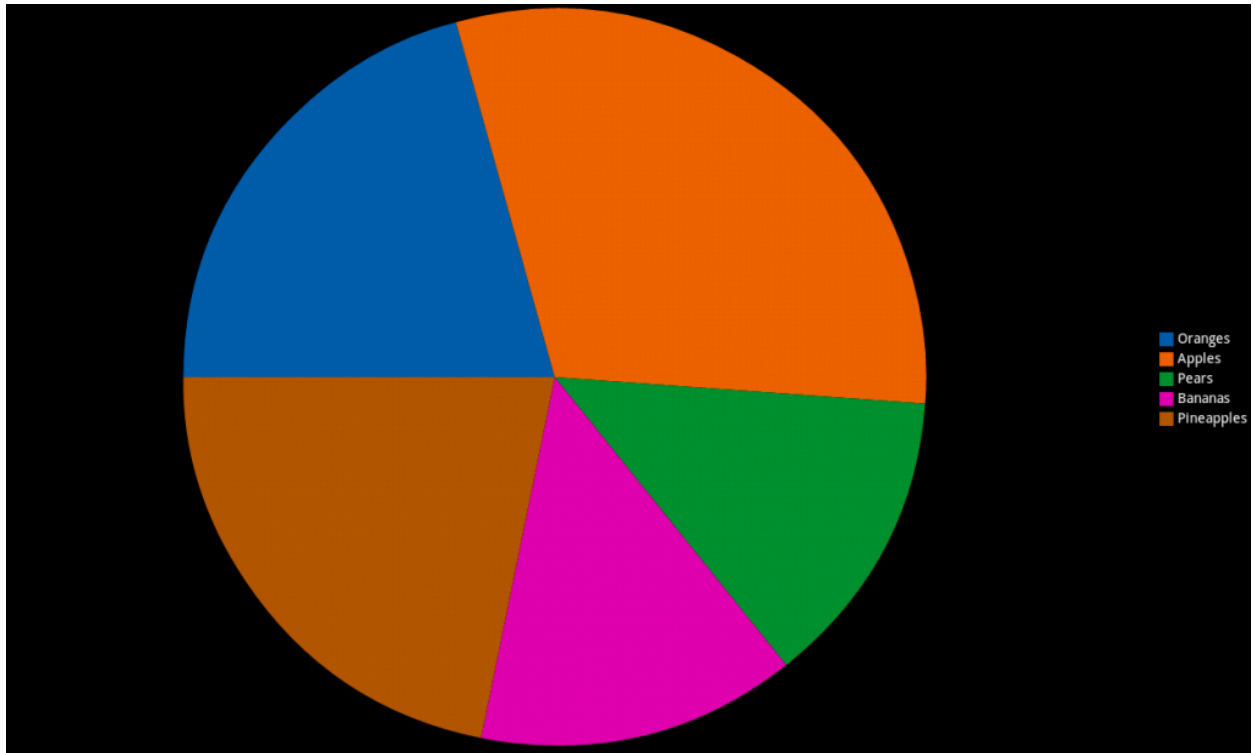
## Themes

Enhance the appearance of the control by using pre-defined themes. The [Palette](#) property can be used to specify the theme to be applied on the control.



**Note:** Remove the **Palette** property from the code to apply the default theme.

The image below shows how the FlexPie appears when the Palette property is set to Dark.



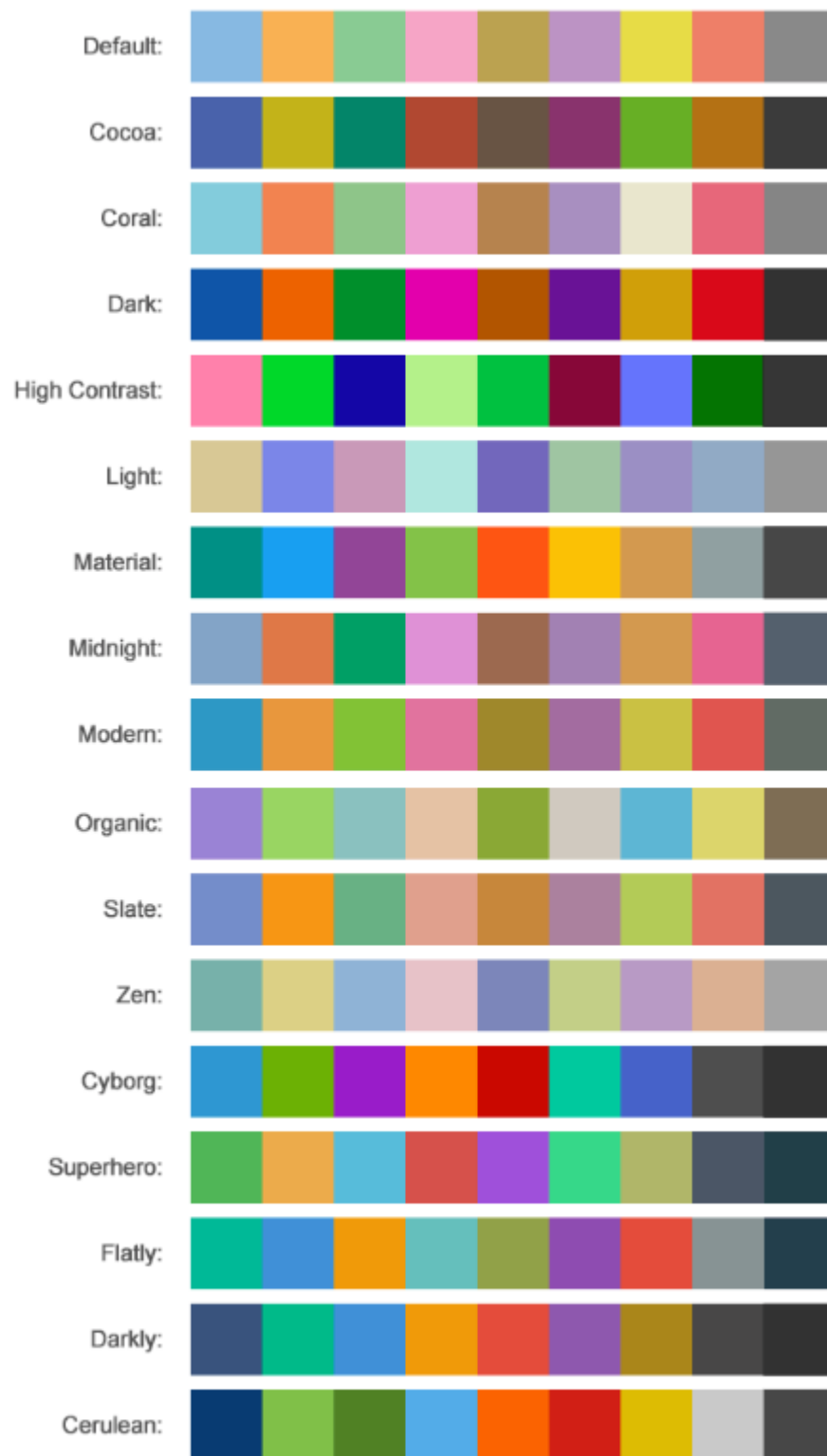
The following code example demonstrates how to apply theme in C#. This example uses the sample created in the [Quick Start](#) section.

#### In Code

C#

```
//setting themes  
chart.Palette = Palette.Cyborg;
```

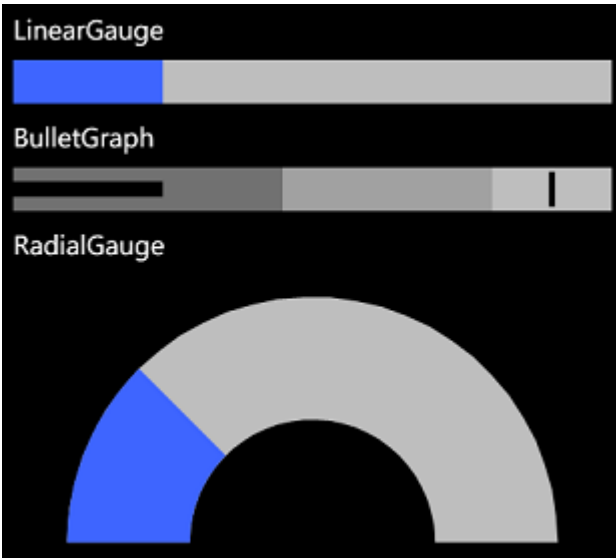
FlexPie comes with pre-defined templates that can be applied for quick customization. Here are the pre-defined templates available in the [Palette](#) enumeration.



## Gauge

**Gauge** allows you to display information in a dynamic and unique way by delivering the exact graphical representation you require. Gauges are better than simple labels because they also display a range, allowing users to determine instantly whether the current value is low, high, or intermediate.





### Key Features

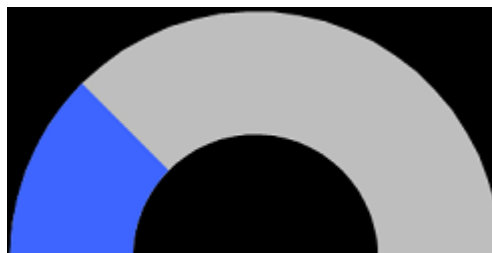
- **Easy Customization:** Restyle the Gauge by changing a property to create gauges with custom colors, fills and more.
- **Ranges:** Add colored ranges to the Gauge to draw attention to a certain range of values. Use simple properties to customize their start and end points, as well as appearance.
- **Direction:** Place the LinearGauge and BulletGraph horizontally or vertically.
- **Pointer Customization:** Customize the pointer color, border, origin and more to make the Gauge more appealing.
- **Animation:** Use out-of-the-box animations to add effects to the Gauge control.

### Gauge Types

Xamarin Edition comprises of three kinds of gauges: [LinearGauge](#), [RadialGauge](#) and [BulletGraph](#).

Type	Image	Usage
<b>Linear Gauge:</b> A linear gauge displays the value along a linear scale, using a linear pointer. The linear scale can be either horizontal or vertical, which can be set using the LinearGaugeDirection property.		A linear gauge is commonly used to denote data as a scale value such as length, temperature, etc.

**Radial Gauge:** A radial gauge displays the value along a circular scale, using a curved pointer. The scale can be rotated as defined by the [StartAngle](#) and [SweepAngle](#) properties.



A radial gauge is commonly used to denote data such as volume, velocity, etc.

**Bullet Graph:** A bullet graph displays a single value on a linear scale, along with a target value and ranges that instantly indicate whether the value is [Good](#), [Bad](#) or in some other state.



A bullet graph is a variant of a linear gauge, designed specifically for use in dashboards that display a number of single value data, such as yearly sales revenue.

## Quick Start: Add and Configure

### BulletGraph Quick Start

This section describes how to add a `BulletGraph` to your portable or shared app and set its value. For information on how to add Xamarin components in C# or XAML, see [Adding Xamarin Components using C#](#) or [Adding Xamarin Components using XAML](#).

This topic comprises of three steps:


- **Step 1: Add a `BulletGraph` control**
- **Step 2: Run the Project**

The following image shows a `BulletGraph`.



#### Step 1: Add a `BulletGraph` control

The `Value` property denotes the value of the Gauge. Multiple ranges can be added to a single Gauge and the position of the range is defined by the `Min` and `Max` properties of the range. If you set the `IsReadOnly` property to **false**, then the user can edit the value by tapping on the gauge.

 **Note:** The `C1BulletGraph.Origin` property can be used to change the origin of the BulletGraph pointer. By default, the **Origin** is set to 0.

Complete the following steps to initialize a BulletGraph control in C# or in XAML.

### In Code

1. Add a new class (for example QuickStart.cs) to your portable or shared project and include the following references:

C#

```
using Xamarin.Forms;
using Cl.Xamarin.Forms.Gauge;
```

2. Instantiate a BulletGraph control in a new method, GetBulletGraph().

C#

```
public static C1BulletGraph GetBulletGraph()
{
    //Instantiate BulletGraph and set its properties
    C1BulletGraph gauge = new C1BulletGraph();
    gauge.Value = 80;
    gauge.Min = 0;
    gauge.Max = 100;
    gauge.Thickness = 0.1;
    gauge.Direction = LinearGaugeDirection.Right;
    gauge.PointerColor = Xamarin.Forms.Color.Black;

    //Set its Good, Bad, and Target
    gauge.Good = 100;
    gauge.GoodRangeColor = Color.FromHex("#CCCCCC");
    gauge.Bad = 50;
    gauge.BadRangeColor = Color.FromHex("#666666");
    gauge.Target = 75;
    gauge.TargetColor = Xamarin.Forms.Color.Black;

    return gauge;
}
```

### In XAML

1. Add a new Forms XAML Page (for example QuickStart.xaml) to your portable or shared project and modify the `<ContentPage>` tag to include the following references.

XAML

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="Appl.QuickStart"
xmlns:cl="clr-namespace:Cl.Xamarin.Forms.Gauge;assembly=Cl.Xamarin.Forms.Gauge">
```

2. Initialize a BulletGraph control by adding the markup for the control between the `<ContentPage>` `</ContentPage>` tags inside the `<StackLayout>` `</StackLayout>` tags.

C#

```
<StackLayout>
```

```
<c1:C1BulletGraph Value="80" Min="0" Max="100" HeightRequest="50"
WidthRequest="50"
    Thickness="0.1" Good="100" GoodRangeColor="#CCCCCC" Bad="50"
BadRangeColor="#666666" Target="75"
    TargetColor="Black" PointerColor="Black" Direction="Right">
</c1:C1BulletGraph>
</StackLayout>
```

## Back to Top

### Step 2: Run the Project

1. In the **Solution Explorer**, double click **App.cs** to open it.
2. Complete the following steps to display the BulletGraph control.
  - **To return a C# class:** In the class constructor App(), set a new ContentPage as the MainPage and assign the control to the ContentPage's Content by invoking the method GetBulletGraph() defined in the previous procedure, **Step 1: Add a BulletGraph Control**.

The following code shows the class constructor App() after completing steps above.

```
C#
public App()
{
    // The root page of your application
    MainPage = new ContentPage
    {
        Content = QuickStart.GetBulletGraph()
    };
}
```

- **To return a Forms XAML Page:** In the class constructor App(), set the Forms XAML Page QuickStart as the MainPage.

The following code shows the class constructor App(), after completing this step.

```
C#
public App()
{
    // The root page of your application
    MainPage = new QuickStart();
}
```

3. Some additional steps are required to run the iOS and UWP apps:
  - **iOS App:**
    1. In the **Solution Explorer**, double click AppDelegate.cs inside YourAppName.iOS project, to open it.
    2. Add the following code to the FinishedLaunching() method.

```
C#
C1.Xamarin.Forms.Gauge.Platform.iOS.C1GaugeRenderer.Init();
```

- **UWP App:**
  1. In the **Solution Explorer**, expand MainPage.xaml.
  2. Double click MainPage.xaml.cs to open it.
  3. Add the following code to the class constructor.

```
C#
```

```
C1.Xamarin.Forms.Gauge.Platform.UWP.C1GaugeRenderer.Init();
```

4. Press **F5** to run the project.

**Back to Top**

## LinearGauge Quick Start

This section describes how to add a [LinearGauge](#) control to your portable or shared app and set its value. For information on how to add Xamarin components in C# or XAML, see [Adding Xamarin Components using C#](#) or [Adding Xamarin Components using XAML](#).

This topic comprises of two steps:

- **Step 1: Add a LinearGauge control**
- **Step 2: Run the Project**

The following image shows a LinearGauge with custom ranges.



### Step 1: Add a LinearGauge control

The [Value](#) property denotes the value of the gauge. Multiple ranges can be added to a single Gauge and the position of the range is defined by the [Min](#) and [Max](#) properties of the range. If you set the [IsReadOnly](#) property to **false**, then the user can edit the value by tapping on the gauge.



**Note:** The [C1LinearGauge.Origin](#) property can be used to change the origin of the LinearGauge pointer. By default, the Origin is set to 0.

Complete the following steps to initialize a LinearGauge control in C# or XAML.

### In Code

1. Add a new class (for example QuickStart.cs) to your portable or shared project and include the following references.

```
C#
```

```
using Xamarin.Forms;
using C1.Xamarin.Forms.Gauge;
```

2. Instantiate a LinearGauge control in a new method GetLinearGauge().

```
C#
```

```
public static C1LinearGauge GetLinearGauge()
{
    // Instantiate LinearGauge and set its properties
    C1LinearGauge gauge = new C1LinearGauge();
    gauge.Value = 35;
    gauge.Thickness = 0.1;
    gauge.Min = 0;
    gauge.Max = 100;
    gauge.Direction = LinearGaugeDirection.Right;
    gauge.PointerColor = Xamarin.Forms.Color.Blue;

    //Create Ranges
```

```
GaugeRange low = new GaugeRange();
GaugeRange med = new GaugeRange();
GaugeRange high = new GaugeRange();

//Customize Ranges
low.Color = Xamarin.Forms.Color.Red;
low.Min = 0;
low.Max = 40;
med.Color = Xamarin.Forms.Color.Yellow;
med.Min = 40;
med.Max = 80;
high.Color = Xamarin.Forms.Color.Green;
high.Min = 80;
high.Max = 100;

//Add Ranges to Gauge
gauge.Ranges.Add(low);
gauge.Ranges.Add(med);
gauge.Ranges.Add(high);

return gauge;
}
```

## In XAML

1. Add a new Forms XAML Page (for example QuickStart.xaml) to your Portable or Shared project and modify the <ContentPage>tag to include references as shown below:

### XAML

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="Appl.QuickStart"
xmlns:cl="clr-
namespace:C1.Xamarin.Forms.Gauge;assembly=C1.Xamarin.Forms.Gauge">
```

2. Initialize a LinearGauge control by adding the markup for the control between the <ContentPage> </ContentPage>tags and inside the <StackLayout> </StackLayout> tags, as shown below:

### XAML

```
<StackLayout>
<c1:C1LinearGauge Value="35" Min="0" Max="100" Thickness="0.1"
    PointerColor="Blue" Direction="Right">
    <c1:C1LinearGauge.Ranges>
        <c1:GaugeRange Min="0" Max="40" Color="Red"/>
        <c1:GaugeRange Min="40" Max="80" Color="Yellow"/>
        <c1:GaugeRange Min="80" Max="100" Color="Green"/>
    </c1:C1LinearGauge.Ranges>
    </c1:C1LinearGauge>
</StackLayout>
```

[Back to Top](#)

## Step 2: Run the Project

1. In the **Solution Explorer**, double click App.cs to open it.
2. Complete the following steps to display the LinearGauge control.
  - **To return a C# class:** In the class constructor App(), set a new ContentPage as the MainPage and assign the control to the ContentPage's Content by invoking the method GetLinearGauge() defined in the previous procedure, **Step 1: Add a LinearGauge Control**.

The following code shows the class constructor App() after completing steps above.

```
C#  
  
public App()  
{  
    // The root page of your application  
    MainPage = new ContentPage  
    {  
        Content = QuickStart.GetLinearGauge()  
    };  
}
```

- **To return a Forms XAML Page:** In the class constructor App(), set the Forms XAML Page QuickStart as the MainPage.

The following code shows the class constructor App(), after completing this step.

```
C#  
  
public App()  
{  
    // The root page of your application  
    MainPage = new QuickStart();  
}
```

3. Some additional steps are required to run the iOS and UWP apps:
  - **iOS App:**
    1. In the **Solution Explorer**, double click AppDelegate.cs inside YourAppName.iOS project, to open it.
    2. Add the following code to the FinishedLaunching() method.

```
C#  
  
C1.Xamarin.Forms.Gauge.Platform.iOS.C1GaugeRenderer.Init();
```

- **UWP App:**
  1. In the **Solution Explorer**, expand MainPage.xaml.
  2. Double click MainPage.xaml.cs to open it.
  3. Add the following code to the class constructor.

```
C#  
  
C1.Xamarin.Forms.Gauge.Platform.UWP.C1GaugeRenderer.Init();
```

4. Press **F5** to run the project.

[Back to Top](#)

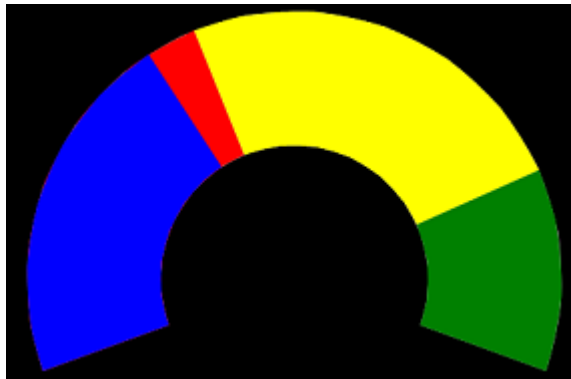
## RadialGauge Quick Start

This section describes how to add a [RadialGauge](#) control to your portable or shared app and set its value. For information on how to add Xamarin components in C# or XAML, see [Adding Xamarin Components using C#](#) or [Adding Xamarin Components using XAML](#).

This topic comprises of two steps:

- **Step 1: Add a RadialGauge control**
- **Step 2: Run the Project**

The following image shows how the RadialGauge appears, after completing the steps above:




### Step 1: Add a RadialGauge control

The [Value](#) property denotes the value of the gauge. Multiple ranges can be added to a single Gauge and the position of the range is defined by the [Min](#) and [Max](#) properties of the range. If you set the [IsReadOnly](#) property to **false**, then the user can edit the value by tapping on the gauge.

The [StartAngle](#) property specifies the RadialGauge's starting angle and the [SweepAngle](#) property specifies an angle representing the length of the RadialGauge's arc. These properties can be used to specify the start and end points of the radial gauge arc. The angle for both properties are measured clockwise, starting at the 9 o'clock position. When the SweepAngle is negative, the gauge is formed in counter clockwise direction.

The control also provides the [AutoScale](#) property. When this property is set to **true**, the RadialGauge is automatically scaled to fill its containing element.

 **Note:** The [C1RadialGauge.Origin](#) property can be used to change the origin of the RadialGauge pointer. By default, the Origin is set to 0.

Complete the following steps to initialize a RadialGauge control in C# or XAML.

### In Code

1. Add a new class (for example QuickStart.cs) to your Portable or Shared project and include the following references.

```
C#  
  
using Xamarin.Forms;  
using C1.Xamarin.Forms.Gauge;
```

2. Instantiate a RadialGauge control in a new method GetRadialGauge( ).

```
C#  
  
public static C1RadialGauge GetRadialGauge()  
{
```



```

//Instantiate RadialGauge and set its properties
ClRadialGauge gauge = new ClRadialGauge();
gauge.Value = 35;
gauge.Min = 0;
gauge.Max = 100;
gauge.StartAngle = -20;
gauge.SweepAngle = 220;
gauge.AutoScale = true;
gauge.ShowText = GaugeShowText.None;
gauge.PointerColor = Xamarin.Forms.Color.Blue;

//Create Ranges
GaugeRange low = new GaugeRange();
GaugeRange med = new GaugeRange();
GaugeRange high = new GaugeRange();

//Customize Ranges
low.Color = Xamarin.Forms.Color.Red;
low.Min = 0;
low.Max = 40;
med.Color = Xamarin.Forms.Color.Yellow;
med.Min = 40;
med.Max = 80;
high.Color = Xamarin.Forms.Color.Green;
high.Min = 80;
high.Max = 100;

//Add Ranges to RadialGauge
gauge.Ranges.Add(low);
gauge.Ranges.Add(med);
gauge.Ranges.Add(high);

return gauge;
}

```

## In XAML

1. Add a new Forms XAML Page (for example QuickStart.xaml) to your Portable or Shared project and modify the <ContentPage> tag to include the following references:

### XAML

```

<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="Appl.QuickStart"
xmlns:cl="clr-namespace:C1.Xamarin.Forms.Gauge;assembly=C1.Xamarin.Forms.Gauge">

```

2. Initialize a RadialGauge control by adding the markup for the control between the <ContentPage> </ContentPage> tags and inside the <StackLayout> </StackLayout> tags as shown below:

### C#

```

<StackLayout>
<cl:C1RadialGauge Value="35" Min="0" Max="100" ShowText="None" StartAngle = "-
20"

```

```

        SweepAngle = "220" AutoScale = "true" PointerColor="Blue">
<c1:C1RadialGauge.Ranges>
    <c1:GaugeRange Min="0" Max="40" Color="Red"/>
    <c1:GaugeRange Min="40" Max="80" Color="Yellow"/>
    <c1:GaugeRange Min="80" Max="100" Color="Green"/>
</c1:C1RadialGauge.Ranges>
</c1:C1RadialGauge>
</StackLayout>

```

## Back to Top

### Step 2: Run the Project

1. In the **Solution Explorer**, double click **App.cs** to open it.
2. Complete the following steps to display the RadialGauge control.
  - **To return a C# class:** In the class constructor App(), set a new ContentPage as the MainPage and assign the control to the ContentPage's Content by invoking the method GetRadialGauge() defined in the previous procedure, **Step 1: Add a RadialGauge Control**.

The following code shows the class constructor App() after completing steps above.

```

C#
public App()
{
    // The root page of your application
    MainPage = new ContentPage
    {
        Content = QuickStart.GetRadialGauge()
    };
}

```

- **To return a Forms XAML Page:** In the class constructor App(), set the Forms XAML Page QuickStart as the MainPage.

The following code shows the class constructor App(), after completing this step.

```

C#
public App()
{
    // The root page of your application
    MainPage = new QuickStart();
}

```

3. Some additional steps are required to run the iOS and UWP apps:
  - **iOS App:**
    1. In the **Solution Explorer**, double click AppDelegate.cs inside YourAppName.iOS project, to open it.
    2. Add the following code to the FinishedLaunching() method.

```

C#
C1.Xamarin.Forms.Gauge.Platform.iOS.C1GaugeRenderer.Init();

```

- **UWP App:**
  1. In the **Solution Explorer**, expand MainPage.xaml.

2. Double click MainPage.xaml.cs to open it.
3. Add the following code to the class constructor.

C#

```
C1.Xamarin.Forms.Gauge.Platform.UWP.C1GaugeRenderer.Init();
```

4. Press **F5** to run the project.

**Back to Top**

## Features

### Animation

Gauges come with in-built animations to improve their appearance. The [IsAnimated](#) property allows you to enable or disable the animation effects. The [LoadAnimation](#) and [UpdateAnimation](#) properties enable you to set the easing, duration and other animation-related attributes.

The following code example demonstrates how to set this property in C#. These example uses the sample created in the [Quick start](#).

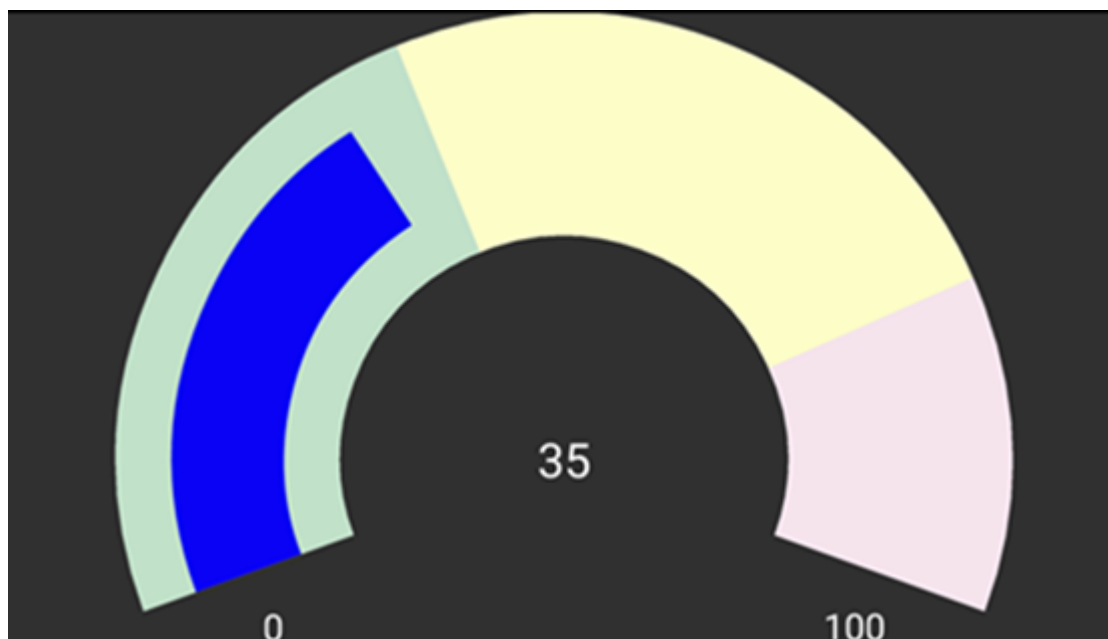
C#

```
gauge.IsAnimated = true;  
gauge.LoadAnimation.Easing = Easing.CubicInOut;  
gauge.UpdateAnimation.Easing = Easing.BounceOut;
```

### Customize Appearance

Xamarin Edition controls match the native controls on all three platforms by default and are designed to work with both: light and dark themes available on all platforms. But, there are several properties available that let you customize the gauge elements to make it visually attractive. The following code example demonstrates how to set different styling properties to customize a RadialGauge.

The following image shows how the RadialGauge appears after its appearance is customized.



The following code example demonstrates how to customize the appearance of a radial gauge in C#. This example uses the sample created in the [Quick start](#).

C#

```
//Customize Gauge
gauge.ShowText = GaugeShowText.All;
gauge.Thickness = 0.5;
gauge.FaceBorderColor = Color.FromHex("#cfd1d4");
gauge.FaceBorderWidth = 5;

gauge.MinTextColor = Color.White;
gauge.MaxTextColor = Color.White;
gauge.MinFontSize = 10;
gauge.MaxFontSize = 10;

//Customize Range Appearance
low.Color = Color.FromHex("#C1E1C9");
med.Color = Color.FromHex("#fdfdc8");
high.Color = Color.FromHex("#f6e4ed");

//Customize Pointer Appearance
gauge.Pointer.Thickness = 0.5;
gauge.PointerColor = Color.FromHex("#0a02f4");
```

## Data Binding

While working in XAML, you can set the [Min](#), [Max](#) and [Value](#) properties from an external data source, instead of directly setting the properties.

The example in this topic follows the MVVM pattern. For information on MVVM pattern, see [MVVM Pattern](#).

The following class can serve as a data source for the gauge.

C#

```
class GaugeData
{
    double _value;
    double _min;
    double _max;

    public double Value
    {
        get { return _value; }
        set
        {
            _value = value;
        }
    }

    public double Min
```

```

        {
            get { return _min; }
            set
            {
                _min = value;
            }
        }
        public double Max
        {
            get { return _max; }
            set
            {
                _max = value;
            }
        }
    }
}

```

### In XAML

The following XAML code shows data binding in gauge.

C#

```

<cl:CLinearGauge Value="{Binding Value}" Min="{Binding Min}" Max="{Binding Max}"
    Thickness="0.1" PointerColor="Blue">
    <cl:CLinearGauge.Ranges>
        <cl:GaugeRange Min="0" Max="40" Color="Red"/>
        <cl:GaugeRange Min="40" Max="80" Color="Yellow"/>
        <cl:GaugeRange Min="80" Max="100" Color="Green"/>
    </cl:CLinearGauge.Ranges>
</cl:CLinearGauge>

```

### In Code

To bind gauge to data, switch to the code view and set the `BindingContext` for the gauge inside the class constructor as shown below.

C#

```

public DataBinding()
{
    InitializeComponent();
    BindingContext = new GaugeData() { Value = 25, Max=100, Min=0 };
}

```

## Direction

The `Direction` property allows you to change the orientation of the gauge as well as the direction in which the pointer moves. For example, setting the `Direction` property to **Down** changes the gauge orientation from horizontal to vertical and the pointer starts from the top of the gauge and move towards the bottom.

The following image shows how the `LinearGauge` appears after this property has been set.



The following code examples demonstrate how to set this property in C# and XAML. These examples use the sample created in the [LinearGauge Quick Start](#) section.

#### In Code

C#

copyCode

```
gauge.Direction = LinearGaugeDirection.Down;
```

#### In XAML

C#

```
<cl:CLinearGauge Value="35" Min="0" Max="100" Thickness="0.1" Direction="Down"
PointerColor="Blue">
```

## Range

You can add multiple ranges to a single Gauge. Each range denotes a region or a state which can help the user identify the state of the Gauge's value. Every range has its [Min](#) and [Max](#) properties that specify the range's position on the Gauge, as well as other properties to define the range's appearance.

The following code examples demonstrate how to add ranges to a Gauge and set their properties in C# and XAML.

#### In Code

Create new instances of type [GaugeRange](#), set their properties and add the newly created ranges to the LinearGauge (or RadialGauge/BulletGraph).

C#

```
//Create Ranges
GaugeRange low = new GaugeRange();
GaugeRange med = new GaugeRange();
GaugeRange high = new GaugeRange();

//Customize Ranges
low.Color = Color.Red;
low.Min = 0;
low.Max = 40;
med.Color = Color.Yellow;
med.Min = 40;
med.Max = 80;
high.Color = Color.Green;
high.Min = 80;
high.Max = 100;

//Add Ranges to Gauge
gauge.Ranges.Add(low);
gauge.Ranges.Add(med);
gauge.Ranges.Add(high);
```

### In XAML

Add the markup for ranges between the opening and closing tags of the control to create new ranges and add them to the LinearGauge (or RadialGauge/BulletGraph).

C#

```
<cl:CLinearGauge Value="35" Min="0" Max="100" Thickness="0.1">
  <cl:CLinearGauge.Ranges>
    <cl:GaugeRange Min="0" Max="40" Color="Red"/>
    <cl:GaugeRange Min="40" Max="80" Color="Yellow"/>
    <cl:GaugeRange Min="80" Max="100" Color="Green"/>
  </cl:CLinearGauge.Ranges>
</cl:CLinearGauge>
```

## User Scenario

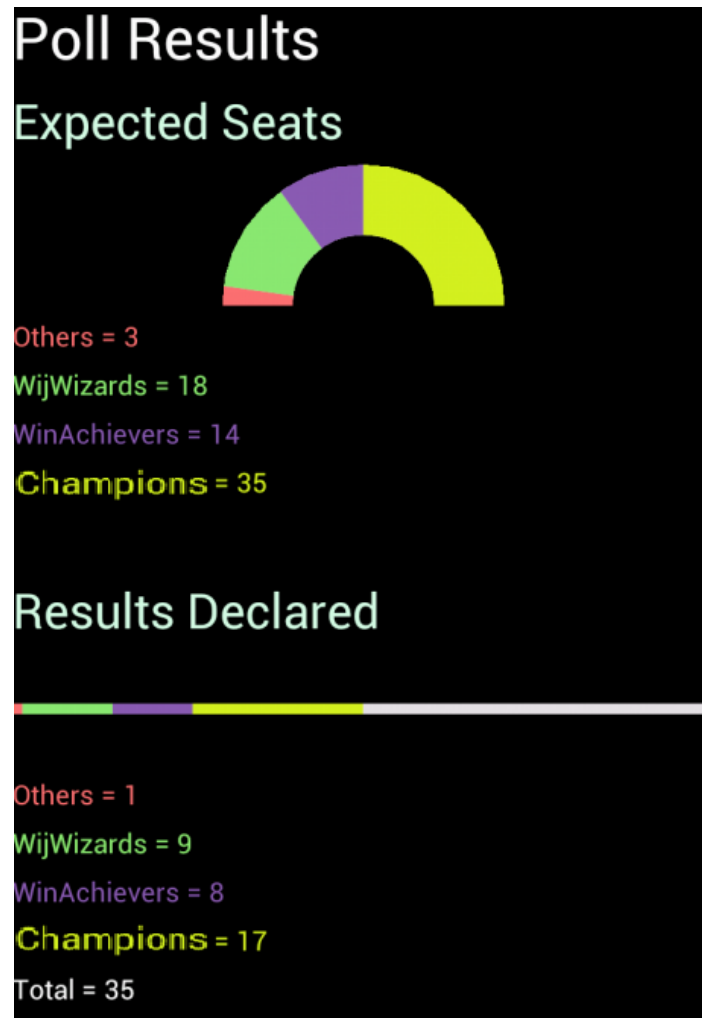
### Poll Results

This topic uses [data binding](#) and [multiple ranges](#) to demonstrate how to use gauges to display poll results. The colored ranges make the results easy to understand. Some corresponding colored labels are also added to display the names of the candidates/political parties.

This topic comprises of four steps:

- **Step 1: Create a data source**
- **Step 2: Add Gauges and Labels**
- **Step 3: Add data to Gauges and Labels**
- **Step 4: Run the Project**

The following image shows how the controls appear after completing the steps above.



#### Step 1: Create a data source

The following class can be used to bind gauges to data.

C#

copyCode

```
class ElectionData
{
    double others, wijWizards, winAchievers, Champions;

    public double Others
    {
        get { return others; }
        set { others = value; }
    }
    public double WijWizards
    {
        get { return wijWizards; }
        set { wijWizards = value; }
    }
    public double WinAchievers
    {
        get { return winAchievers; }
        set { winAchievers = value; }
    }
    public double Champions
    {
        get { return Champions; }
        set { Champions = value; }
    }
}
```



[Back to Top](#)

## Step 2: Add Gauges and Labels

Complete the following steps to add a radial gauge to display the expected seats and a linear gauge to display the results declared. Add labels below the gauges to display the names of the parties and their seat count.

1. Add a new Forms XAML Page (for example ElectionResult.xaml) to your portable or shared project and modify the <ContentPage> tag to include the following references:

### XAML

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="Appl.ElectionResult"
xmlns:cl="clr-namespace:C1.Xamarin.Forms.Gauge;assembly=C1.Xamarin.Forms.Gauge">
```

2. Initialize the controls by adding the markups between the <ContentPage> </ContentPage> tags and inside the <StackLayout> </StackLayout> tags, as shown below:

### Example Title

```
<StackLayout>
<Label Text="Poll Results" TextColor="White" Font="30"/>
<Label Text="Expected Seats" TextColor="#D3FDE5" Font="25"/>
<!--Radial Gauge-->
<cl:C1RadialGauge x:Name="gauge" Min="0" Max="70"
ShowText="None" AutoScale = "true">
<!--Colored ranges to represent parties-->
<cl:C1RadialGauge.Ranges>
<cl:GaugeRange x:Name="Range1" Color="#FF7373"/>
<cl:GaugeRange x:Name="Range2" Color="#8DEB71"/>
<cl:GaugeRange x:Name="Range3" Color="#8F5DB7"/>
<cl:GaugeRange x:Name="Range4" Color="#D4F320"/>
</cl:C1RadialGauge.Ranges>
</cl:C1RadialGauge>
<!--Labels to display party names and seat count-->
<Label x:Name="lb1" TextColor="#FF7373"/>
<Label x:Name="lb11" TextColor="#8DEB71"/>
<Label x:Name="lb12" TextColor="#8F5DB7"/>
<Label x:Name="lb13" TextColor="#D4F320"/>
<Label Text="Results Declared" TextColor="#D3FDE5" Font="25"/>
<!--Linear Gauge-->
<cl:C1LinearGauge x:Name="lGauge" Min="0" Max="70" Thickness="0.1"
Direction="Right" ShowRanges="True">
<!--Colored ranges to represent parties-->
<cl:C1LinearGauge.Ranges>
<cl:GaugeRange x:Name="lRange1" Color="#FF7373"/>
<cl:GaugeRange x:Name="lRange2" Color="#8DEB71"/>
<cl:GaugeRange x:Name="lRange3" Color="#8F5DB7"/>
<cl:GaugeRange x:Name="lRange4" Color="#D4F320"/>
</cl:C1LinearGauge.Ranges>
</cl:C1LinearGauge>
<!--Labels to display party names and seat count-->
<Label x:Name="lb1" TextColor="#FF7373"/>
<Label x:Name="lb11" TextColor="#8DEB71"/>
<Label x:Name="lb12" TextColor="#8F5DB7"/>
<Label x:Name="lb13" TextColor="#D4F320"/>
<Label x:Name="lb14" TextColor="White"/>
</StackLayout>
```

[Back to Top](#)

## Step 3: Add data to Gauges and Labels

Complete the following steps to add data to gauges and labels.

1. In the **Solution Explorer**, expand the ElectionResult.xaml node and open ElectionResult.xaml.cs to open the C# code behind.
2. In the ElectionResult() class constructor, create new instances of the class ElectionData, defined in **Step 1: Create a data source** and add data to the

controls.

The following code shows what the ElectionResult class constructor looks like after completing this step.

C#	copyCode
<pre>public ElectionResult() {     InitializeComponent();      // Results Declared     ElectionData bds = new ElectionData() { Others = 1, WijWizards = 9, WinAchievers = 8, Champions = 17 };      lRange1.Min = 0;     lRange1.Max = lRange1.Min + bds.Others;      lRange2.Min = lRange1.Max;     lRange2.Max = lRange2.Min + bds.WijWizards;      lRange3.Min = lRange2.Max;     lRange3.Max = lRange3.Min + bds.WinAchievers;      lRange4.Min = lRange3.Max;     lRange4.Max = lRange4.Min + bds.Champions;      // Add data to labels     lbl1.Text = "Others = " + bds.Others;     lbl11.Text = "WijWizards = " + bds.WijWizards;     lbl12.Text = "WinAchievers = " + bds.WinAchievers;     lbl13.Text = "Champions = " + bds.Champions;     lbl14.Text = "Total = " + (bds.Others + bds.WijWizards + bds.WinAchievers + bds.Champions).ToString();      // Expected Seats     ElectionData ds = new ElectionData() { Others = 3, WijWizards = 18, WinAchievers = 14, Champions = 35 };      Range1.Min = 0;     Range1.Max = Range1.Min + ds.Others;      Range2.Min = Range1.Max;     Range2.Max = Range2.Min + ds.WijWizards;      Range3.Min = Range2.Max;     Range3.Max = Range3.Min + ds.WinAchievers;      Range4.Min = Range3.Max;     Range4.Max = Range4.Min + ds.Champions;      // Add data to labels     lbl1.Text = "Others = " + ds.Others;     lbl11.Text = "WijWizards = " + ds.WijWizards;     lbl12.Text = "WinAchievers = " + ds.WinAchievers;     lbl13.Text = "Champions = " + ds.Champions + "\n\n"; }</pre>	

**Back to Top**

#### Step 4: Run the Project

Complete the following steps

1. In the **Solution Explorer**, double click App.cs to open it.
2. In the class constructor App(), set the Forms XAML Page ElectionResult as the MainPage.

The following code shows the class constructor App(), after completing this step.

C#
<pre>public App() {     // The root page of your application     MainPage = new ElectionResult(); }</pre>

```
}
```

3. Some additional steps are required to run the iOS and UWP apps:

- o **iOS App:**

1. In the **Solution Explorer**, double click AppDelegate.cs inside YourAppName.iOS project to open it.
2. Add the following code to the FinishedLaunching() method.

```
C#
Cl.Xamarin.Forms.Gauge.Platform.iOS.Forms.Init();
```

- o **UWP App:**

1. In the **Solution Explorer**, expand MainPage.xaml.
2. Double click MainPage.xaml.cs to open it.
3. Add the following code to the class constructor.

```
C#
Cl.Xamarin.Forms.Gauge.Platform.UWP.Forms.Init();
```

4. Press **F5** to run the project.

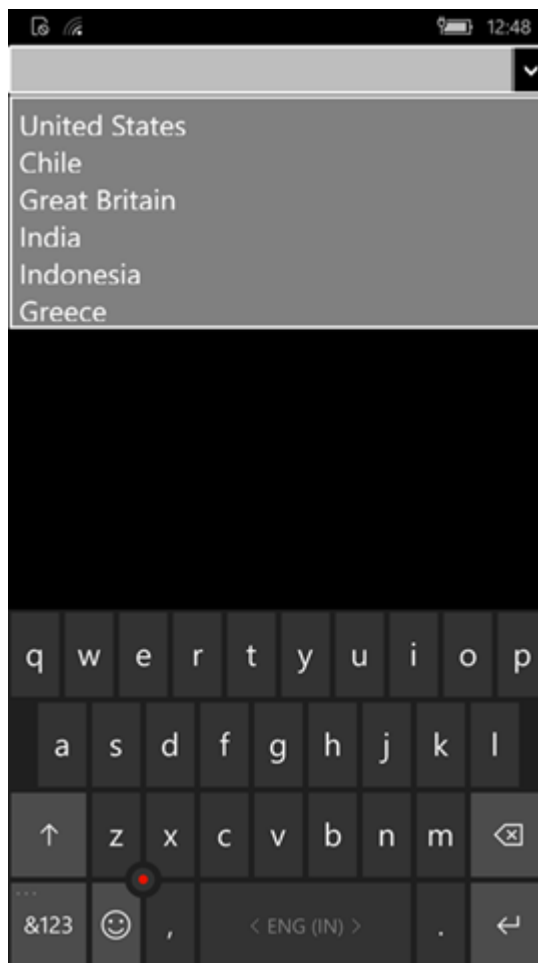
**Back to Top**

## Input

### AutoComplete

[AutoComplete](#) is an editable input control designed to show possible text suggestions automatically as the user types text. The control filters a list of pre-defined items dynamically as a user types to provide suggestions that best or completely matches the input. The suggestions that match the user input appear instantly in a drop-down list.

The image given below shows the AutoComplete control showing pre-populated input suggestions in the drop-down list.



### Key Features

- **Customize Appearance** - Use basic appearance properties to customize the appearance of the drop-down list.
- **Delay** - Use delay feature to provide some time gap (in milliseconds) between text input and suggestion.
- **Dynamic Filtering** - Apply dynamic filtering to enhance searching and filtering as a user types text.
- **Highlight Matches** - Highlight the input text with matching string in the suggestions.

## Quick Start: Populating AutoComplete with Data

This section describes adding the AutoComplete control to your portable or shared application and displaying a list of items in the drop-down as suggestions for users.

Complete the following steps to display an AutoComplete control.

- **Step 1: Add a list of pre-defined suggestions**
- **Step 2: Add AutoComplete control and populate it with suggestions**
- **Step 3: Run the Project**

The following image shows how the AutoComplete control provides a list of suggestions in a drop-down list when the user enters text.



### Step 1: Add a list of pre-defined suggestions

Complete the following steps to add a list of items to be displayed in the drop-down list.

1. Create a new portable or shared Xamarin.Forms application (Refer [Creating a New Xamarin.Forms App](#) for detailed instructions).
2. Add a new class (say Country.cs) to your application.
3. Add the following code to initialize a collection of items (here, a list of major countries).

```
C#  
  
public class Country  
{  
    public string Name { get; set; }  
    public static List<Country> GetCountries()  
    {  
        List<Country> listCountries = new List<Country>();  
        string[] countries = new string[] { "United States", "Chile", "Great  
Britain", "India", "Indonesia", "Greece" };  
        foreach (var item in countries)  
        {  
            listCountries.Add(new Country() { Name = item });  
        }  
        return listCountries;  
    }  
}
```

### Step 2: Add AutoComplete control and populate it with suggestions

1. Add a new Forms XAML Page (say Page1.xaml) to your application.
2. Edit the <ContentPage> tag to include the following reference.

XAML

```
xmlns:cl="clr-namespace:C1.Xamarin.Forms.Input;assembly=C1.Xamarin.Forms.Input"
```

3. Initialize an editable AutoComplete control and set some of its basic properties such as Name, DisplayMemberPath, etc. by adding the given markup between <StackLayout> </StackLayout> tags inside the <ContentPage> </ContentPage> tags.

XAML

```
<StackLayout Orientation="Vertical">
```

```

        <cl:C1AutoComplete
        x:Name="autoComplete"
        IsEditable="True"
        HorizontalOptions="FillAndExpand"
        DropDownBackgroundColor="Gray"
        DisplayMemberPath="Name"
        VerticalOptions="Start">
            <cl:C1AutoComplete.ListView>
                <ListView>
                    <ListView.ItemTemplate>
                        <DataTemplate>
                            <StackLayout Orientation="Horizontal" >
                                <Label Text="{Binding Name}" />
                            </StackLayout>
                        </DataTemplate>
                    </ListView.ItemTemplate>
                </ListView>
            </cl:C1AutoComplete.ListView>
        </cl:C1AutoComplete>
    </StackLayout>

```

4. Expand the Page1.xaml node in the Solution Explorer to open Page1.xaml.cs and add the given code in the constructor to set ItemsSource property of the AutoComplete control.

C#

```
this.autoComplete.ItemsSource = Country.GetCountries();
```

### Step 3: Run the Project

1. In the Solution Explorer, double-click **App.cs** file to open it.
2. To return a Forms XAML Page, set the MainPage to Page1 in the constructor App() as illustrated in the given code

C#

```

public App()
{
    // The root page of your application
    MainPage = new Page1();
}

```

3. Some additional steps are required to run iOS and UWP apps:
  - **iOS App:**
    1. In the Solution Explorer, double click AppDelegate.cs inside YourAppName.iOS project to open it.
    2. Add the following code to the FinishedLaunching() method.

C#

```
C1.Xamarin.Forms.Input.Platform.iOS.C1InputRenderer.Init();
```

- **UWP App:**

1. In the Solution Explorer, expand the MainPage.xaml inside YouAppName.UWP project.
2. Double click the MainPage.xaml.cs to open it and add the following code to the class constructor.

C#

```
C1.Xamarin.Forms.Input.Platform.UWP.C1InputRenderer.Init();
```

4. Press **F5** to run the project.

## Features

### Data Binding

The AutoComplete control provides `DisplayMemberPath` and `ItemsSource` properties to bind the control to data. The `ItemsSource` property lets you bind the control to an enumerable collection of items, and the `DisplayMemberPath` property sets the path to a value on the source object for displaying data.

The following code illustrates how to set these properties in code to achieve data binding.

```
C#  
  
this.autoComplete.ItemsSource = Country.GetCountries();  
this.autoComplete.DisplayMemberPath = "Name";
```

### Delay

The AutoComplete control provides instant text suggestions by searching the best possible match for the user input. However, you can change this default behavior and add some time gap between user input and the search. For this, you can use the `Delay` property and set time delay in milliseconds.

The following code example shows setting time delay in the AutoComplete control.

```
C#  
  
//Setting the time delay  
this.auto.Delay = 1500;
```

### Highlight Matches

The AutoComplete control enables quick identification of user input in the search result by highlighting the matching text. For this, the AutoComplete class provides the `HighlightedColor` property that sets the highlight color for the matching characters. You can explicitly set this property to a specific color so that the user input string gets highlighted in the search results as shown in the following image.



The following code example shows setting the text highlight feature in the AutoComplete control.

C#

```
this.autoComplete.HighlightedColor = Xamarin.Forms.Color.Blue;
```

## CheckBox

**CheckBox** provides a cross-platform implementation of the class checkbox control. CheckBox is a visualization control and provides input for Boolean values. Users can select and clear the control by simply tapping it.

For Android and Windows platforms, CheckBox uses the standard control. It provides an animated and appropriately styled representation for iOS Universal apps as shown in the image below.



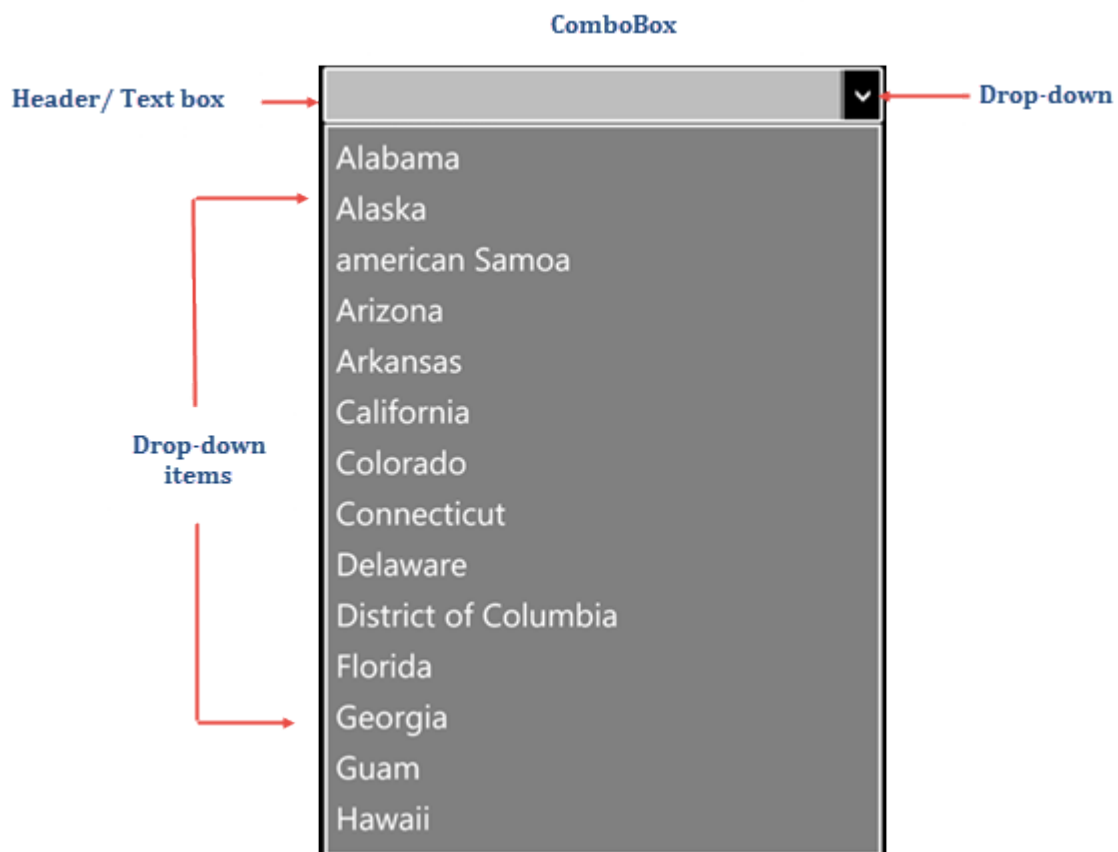
You can set the value of the checkbox by setting the **IsChecked** property. You can customize the style by setting the **Color** property. The following image shows a CheckBox displayed in a FlexGrid template.

	Active	Name	Order Total
	▶	India (15 items)	\$70,573.71
	▶	Japan (11 items)	\$50,893.43
	▶	Mexico (11 items)	\$52,420.97
	▲	China (4 items)	\$22,647.73
	<input type="checkbox"/>	Mark Heath	\$6,333.38
	<input checked="" type="checkbox"/>	Herb Krause	\$448.51
	<input type="checkbox"/>	Ulrich Lehman	\$7,100.58
	<input checked="" type="checkbox"/>	Jack Ulam	\$8,765.26
	▲	Indonesia (6 items)	\$22,132.60
	<input checked="" type="checkbox"/>	Ulrich Quaid	\$265.49
	<input type="checkbox"/>	Ted Richards	\$1,581.26
	<input type="checkbox"/>	Xavier Neiman	\$4,353.91
	<input type="checkbox"/>	Charlie Evers	\$7,787.81
	<input checked="" type="checkbox"/>	Vic Lehman	\$1,488.28
	<input type="checkbox"/>	Larry Bishop	\$6,655.85

## ComboBox

**ComboBox** is an input control that combines the features of a standard text box and a list view. The control is used to display and select data from the list that appears in a drop-down. Users can also type the text into the editable text box that appears in the header to provide input. The control also supports automatic completion to display input suggestions as the user types in the text box.





### Key Features

- **Automatic Completion** - ComboBox supports automatic completion feature that provides relevant suggestions to user while typing the text in the text area.
- **Edit Mode** - By default, the ComboBox control is non-editable. However, you can make it editable so that users can modify their input as well.

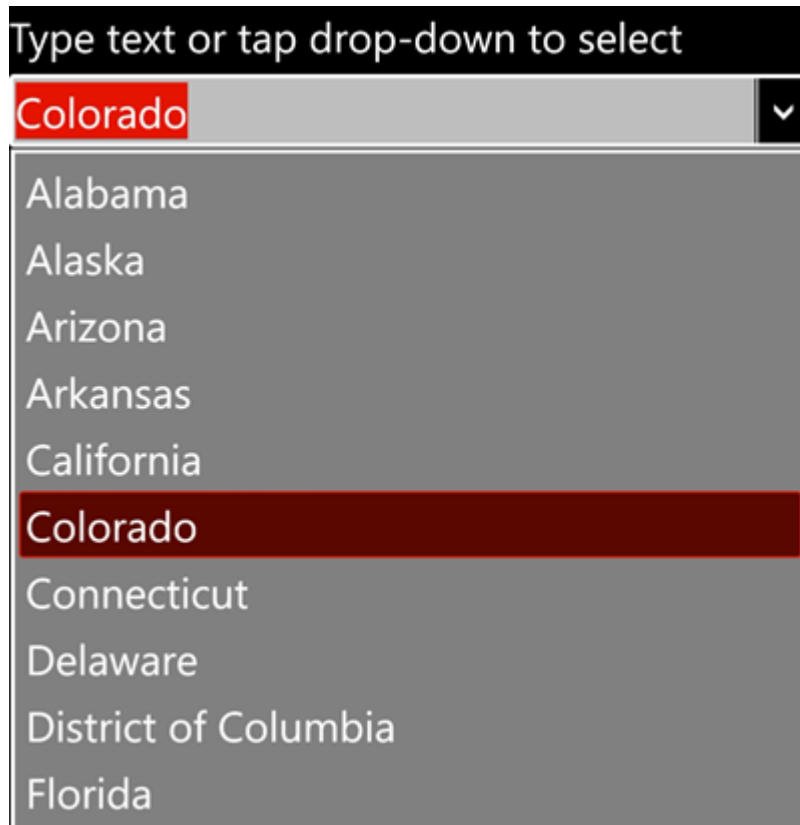
## Quick Start: Display a ComboBox Control

This section describes adding a ComboBox control to your portable or shared application and displaying a list of items in the drop-down as suggestions for users.

Complete the following steps to display a ComboBox control.

- **Step 1: Add an item list to be displayed in the drop-down**
- **Step 2: Add ComboBox control to XAML Page**
- **Step 3: Run the Project**

The following image shows a ComboBox displaying input suggestions as the user types.



### Step 1: Add an item list to be displayed in the drop-down

Complete the following steps to add a list of items to be displayed in the drop-down list.

1. Create a new portable or shared Xamarin.Forms application (Refer [Creating a New Xamarin.Forms App](#) for detailed instructions).
2. Add a new class (say States.cs) to your application.
3. Add the following code to initialize a collection of items (here, a list of major states in the United States).

C#

```
public class States
{
    public string Name { get; set; }
    public States(string name)
    {
        this.Name = name;
    }

    public static List<States> GetStates()
    {
        List<States> data = new List<States>();
        data.Add(new States("Alabama"));
        data.Add(new States("Alaska"));
        data.Add(new States("Arizona"));
        data.Add(new States("Arkansas"));
        data.Add(new States("California"));
        data.Add(new States("Colorado"));
        data.Add(new States("Connecticut"));
        data.Add(new States("Delaware"));
```

```

        data.Add(new States("District of Columbia"));
        data.Add(new States("Florida"));
        data.Add(new States("Georgia"));
        data.Add(new States("Guam"));
        data.Add(new States("Hawaii"));
        data.Add(new States("Idaho"));
        data.Add(new States("Illinois"));
        data.Add(new States("Indiana"));
        data.Add(new States("Iowa"));
        data.Add(new States("Kansas"));
        return data;
    }
}

```

## Step 2: Add ComboBox control to XAML Page

1. Add a new Forms XAML Page (say Page1.xaml) to your application.
2. Edit the <ContentPage> tag to include the following reference.

XAML

```
xmlns:c1="clr-namespace:C1.Xamarin.Forms.Input;assembly=C1.Xamarin.Forms.Input"
```

3. Initialize an editable ComboBox control and set some of its basic properties such as Name, DisplayMemberPath, etc. by adding the given markup between <StackLayout> </StackLayout> tags inside the <ContentPage> </ContentPage> tags.

XAML

```

<StackLayout>
    <Label Text="Type text or tap drop-down to select" FontSize="25"/>
    <c1:C1ComboBox x:Name="cbxEdit" IsEditable="True"
HorizontalOptions="FillAndExpand"
                DropDownBackgroundColor="Gray" DisplayMemberPath="Name"
VerticalOptions="Start" />
</StackLayout>

```

4. Expand the Page1.xaml node in the Solution Explorer to open Page1.xaml.cs and add the given code in the constructor to set the [ItemsSource](#) property of the [C1ComboBox](#) class.

C#

```

var array = States.GetStates();
this.cbxEdit.ItemsSource = array;

```

## Step 3: Run the Project

1. In the Solution Explorer, double-click App.cs file to open it.
2. To return a Forms XAML Page, set the MainPage to Page1 in the constructor App() as illustrated in the given code

C#

```

public App()
{
    // The root page of your application
    MainPage = new Page1();
}

```

3. Some additional steps are required to run iOS and UWP apps:

- **iOS App:**

1. In the Solution Explorer, double click AppDelegate.cs inside YourAppName.iOS project to open it.
2. Add the following code to the FinishedLaunching() method.

```
C#
C1.Xamarin.Forms.Input.Platform.iOS.C1InputRenderer.Init();
```

- **UWP App:**

1. In the Solution Explorer, expand the MainPage.xaml inside YourAppName.UWP project.
2. Double click the MainPage.xaml.cs to open it and add the following code to the class constructor.

```
C#
C1.Xamarin.Forms.Input.Platform.UWP.C1InputRenderer.Init();
```

4. Press **F5** to run the project.

## Features

### Custom Appearance

The ComboBox control comes with various properties to customize its appearance. The [C1ComboBox](#) class provides a set of properties listed below to achieve customization in the control's overall look and feel.

- [DropDownBackgroundColor](#) - sets the background color in drop-down list
- [DropDownBorderColor](#) - sets the color of drop-down border
- [DropDownDirection](#) - Sets the direction in which the drop-down opens, accepting values from the DropDownDirection enumeration

The image given below shows a customized combo box control.



The given code illustrates how to set the above properties and render a customized combo box control. This code example uses the sample created in the [Quick Start](#).

C#

```
this.cbxEit.ButtonColor = Xamarin.Forms.Color.Fuschia;  
this.cbxEit.DropDownBackgroundColor = Xamarin.Forms.Color.Green;  
this.cbxEit.DropDownBorderColor = Xamarin.Forms.Color.Navy;  
this.cbxEit.DropDownDirection = DropDownDirection.ForceBelow;
```

## Data Binding

The ComboBox class provides [DisplayMemberPath](#) and [ItemsSource](#) properties to bind the control to data. The **DisplayMemberPath** property sets the path to a value on the source object for displaying data, and the **ItemsSource** property lets you bind the control to a collection of items.

The following code snippet illustrates how to set these properties in code to achieve data binding.

C#

```
this.cbxEit.ItemsSource = array;  
this.cbxEit.DisplayMemberPath = "Name";
```

## Editing

The ComboBox control allows users to input data by either selecting an item from the drop-down or by typing text into the text box. However, by default, the ComboBox control is non-editable and does not allow users to provide input by typing.

To change this default behavior, the [C1ComboBox](#) class provides the [IsEditable](#) property that can be set to **true** to enable editing in the text box.

C#

```
this.cbxEit.IsEditable = true;
```

## DropDown

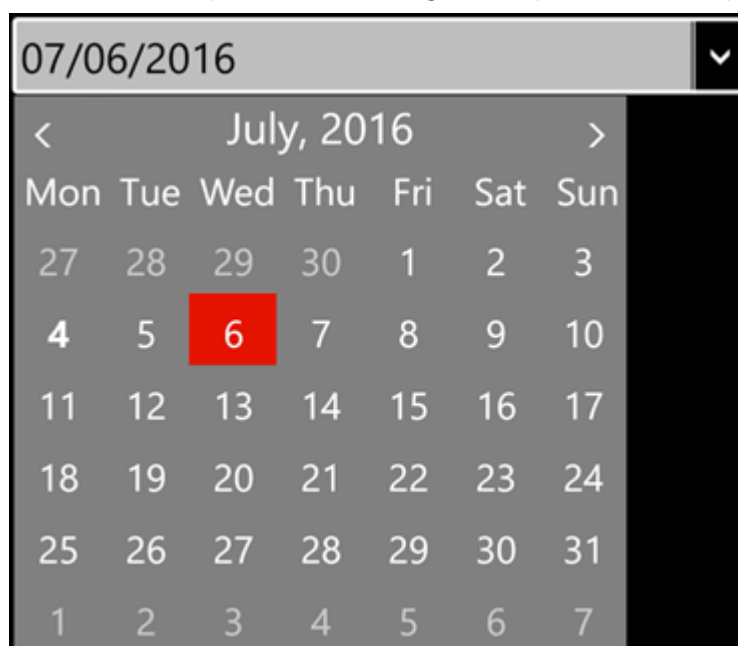
[DropDown](#) is a basic drop-down control that can be used as a base to create custom drop-down controls such as date picker, auto complete menus, etc. The control comprises three major elements, including a Header view, a button, and a DropDown view. The header includes the entire width of the control, while the button is placed on the top of the header, indicating that the control can be expanded. The drop-down includes the entire length of the control and gets expanded or collapsed.

Refer to [Creating a Custom Date Picker using DropDown](#) to know how the control can be used to create full-fledged drop-down menus.

## Creating a Custom Date Picker using DropDown

This topic provides a walkthrough of how the DropDown control can be used to create a custom date picker. For this, you begin by creating a new portable or shared application, initializing a DropDown control in XAML view, and then supplying content to its Header view using the `setHeader` property, and to the DropDown view using the `setDropDown` property. In this example, you create a date picker by using a `MaskedTextField` control in the header and a `Calendar` control in the drop-down.

The image below shows how a custom date picker created using the DropDown control appears.



Complete the following steps to create a custom date picker using the DropDown control.

1. Create a new portable or shared Xamarin.Forms application (Refer [Creating a New Xamarin.Forms App](#) for detailed instructions).

2. Add a new Forms XAML Page (say Page1.xaml) to your application.
3. Edit the <ContentPage> tag to include the required references.

## XAML

```
xmlns:c1="clr-namespace:C1.Xamarin.Forms.Input;assembly=C1.Xamarin.Forms.Input"

xmlns:c1="clr-
namespace:C1.Xamarin.Forms.Calendar;assembly=C1.Xamarin.Forms.Calendar"
```

4. Initialize a DropDown control, a MaskedTextField control, and a Calendar control by adding the given markup between the <ContentPage> </ContentPage> tags.

## XAML

```
<StackLayout>
    <Label BindingContext="{x:Reference page}" Text="{Binding Title}"
    IsVisible="{StaticResource TitleVisible}" HorizontalOptions="Center"
    Font="Large"/>
    <Grid VerticalOptions="FillAndExpand" >
        <c1:C1DropDown x:Name="dropdown"
            HorizontalOptions="FillAndExpand"
            VerticalOptions="Start" >
            <c1:C1DropDown.Header>
                <c1:C1MaskedEntry x:Name="mask"
                    BindingContext="{x:Reference calendar}"
                    Mask="00/00/0000" />
            </c1:C1DropDown.Header>
            <c1:C1DropDown.DropDown>
                <calendar:C1Calendar HorizontalOptions="FillAndExpand"
                x:Name="calendar" BackgroundColor="Transparent">
                </calendar:C1Calendar>
            </c1:C1DropDown.DropDown>
        </c1:C1DropDown>
    </Grid>
</StackLayout>
</ContentPage>
```

5. Expand the Page1.xaml node in the Solution Explorer to open Page1.xaml.cs and add the given code in the constructor.

## XAML

```
public partial class Page1 : ContentPage
{
    public Page1()
    {
        InitializeComponent();
        this.calendar.SelectionChanged += CalendarSelectionChanged;
    }

    private void CalendarSelectionChanged(object sender,
    CalendarSelectionChangedEventArgs e)
    {
        this.mask.Value = calendar.SelectedDate.ToString();
        this.dropdown.IsDropDownOpen = false;
    }
}
```

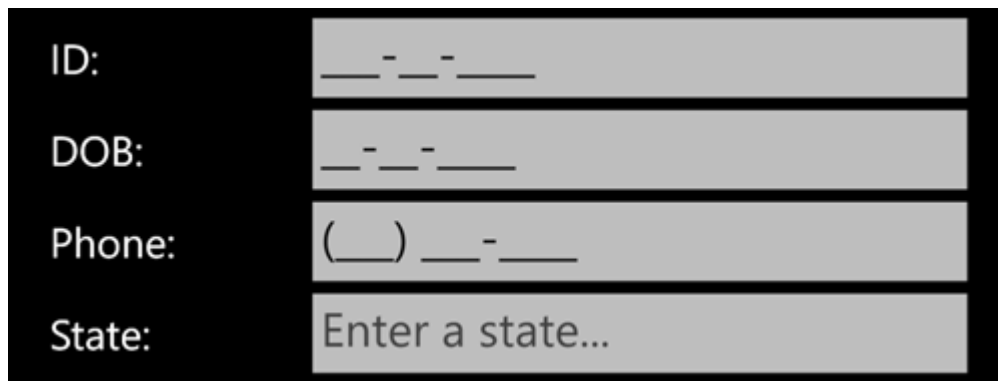
```
}
```

6. Press **F5** to run the project.

## MaskedTextField

**MaskedTextField** is an input control designed to capture properly formatted user input. The control prevents users from entering invalid values in an input field, and characters like slash or hyphen. The control also provides data validation by skipping over invalid entries as the user types. To specify the format in which the data should be entered in an input field, the control uses special characters called mask symbols or mask inputs.

For example, you can use the control to create an input field that accepts phone numbers with area code only. Similarly, you can define a Date field that allows users to enter date in dd/mm/yyyy format. The image given below shows four **MaskedTextField** controls displaying fields namely **ID** to enter a valid identification number, **DOB** to enter date of month in dd-mm-yyyy format, **Phone** to enter a contact number with area code, and **State** showing a water mark (Placeholder) to prompt user to enter the name of state.



The image shows four input fields with their respective labels on the left:

- ID:** The input field contains the mask `__-__-__`.
- DOB:** The input field contains the mask `__-__-__`.
- Phone:** The input field contains the mask `(__)__-__`.
- State:** The input field contains the placeholder text "Enter a state..." in a lighter gray font.

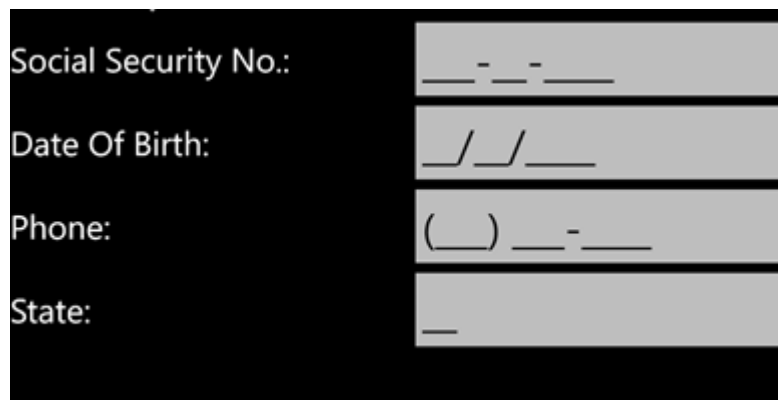
## Quick Start: Display a MaskedTextField Control

This section describes adding a **MaskedTextField** control to your portable or shared application and specifying four input fields, namely Social Security No., Date of Birth, Phone and State. The Social Security No. input field accepts a nine-digit number separated by hyphens, the Date of Birth field accepts a date in mm/dd/yyyy format, the Phone field accepts a 10-digit number with area code, and the State field accepts the abbreviated postal codes of a state.

Complete the following steps to initialize four input fields using **MaskedTextField** control.

- **Step 1: Add MaskedTextField controls to display four input fields**
- **Step 2: Run the Project**

The following image shows the input fields configured after completing the above steps.



The image shows four input fields with their respective labels on the left:

- Social Security No.:** The input field contains the mask `__-__-__`.
- Date Of Birth:** The input field contains the mask `__/__/__`.
- Phone:** The input field contains the mask `(__)__-__`.
- State:** The input field contains a single underscore `_` as a placeholder.



### Step 1: Add MaskedTextField controls to display four input fields

Complete the following steps to initialize a MaskedTextField control in XAML.

1. Create a new portable or shared Xamarin.Forms application (Refer [Creating a New Xamarin.Forms App](#) for detailed instructions).
2. Add a new Forms XAML Page (say Page1.xaml) to your application.
3. Edit the <ContentPage> tag to include the following reference.

XAML

```
xmlns:c1="clr-namespace:C1.Xamarin.Forms.Input;assembly=C1.Xamarin.Forms.Input"
```

4. Initialize four MaskedTextField controls along with corresponding labels within the <StackLayout> tags to display four input fields, and set the **Mask** property for all the masked entry controls.

XAML

```
<StackLayout>
    <Grid>
        <Label Text="Social Security No.:" VerticalOptions="Center" />
        <c1:C1MaskedTextField x:Name="c1MaskedTextBox1" Mask="000-00-0000"
Grid.Column="1" />
        <Label Text="Date Of Birth:" VerticalOptions="Center" Grid.Row="1" />
        <c1:C1MaskedTextField x:Name="c1MaskedTextBox2" Keyboard="Numeric"
Mask="90/90/0000" Grid.Row="1" Grid.Column="1" />
        <Label Text="Phone:" VerticalOptions="Center" Grid.Row="2" />
        <c1:C1MaskedTextField x:Name="c1MaskedTextBox3" Mask="(999) 000-0000"
Keyboard="Telephone" Grid.Row="2" Grid.Column="1" />
        <Label Text="State:" VerticalOptions="Center" Grid.Row="3" />
        <c1:C1MaskedTextField x:Name="c1MaskedTextBox4" Mask="LL" Grid.Row="3"
Grid.Column="1" />
    </Grid>
</StackLayout>
```

### Step 2: Run the Project

1. In the Solution Explorer, double-click **App.cs** file to open it.
2. To return a Forms XAML Page, set the MainPage to Page1 in the constructor App() as illustrated in the given code

C#

```
public App ()
{
    // The root page of your application
    MainPage = new Page1();
}
```

3. Some additional steps are required to run iOS and UWP apps:
  - o **iOS App:**
    1. In the Solution Explorer, double click AppDelegate.cs inside YourAppName.iOS project to open it.
    2. Add the following code to the FinishedLaunching() method.

C#

```
C1.Xamarin.Forms.Input.Platform.iOS.C1InputRenderer.Init();
```

- o **UWP App:**
  1. In the Solution Explorer, expand the MainPage.xaml inside YouAppName.UWP project.

2. Double click the MainPage.xaml.cs to open it and add the following code to the class constructor.

```
C#  
C1.Xamarin.Forms.Input.Platform.UWP.C1InputRenderer.Init();
```

4. Press **F5** to run the project.

## Mask Symbols

The MaskedTextField control provides an editable Mask that supports a set of special mask characters/symbols, which is a subset of .NET MaskedTextBox Mask. These characters are used to specify the format in which the data should be entered in an input field. For this, you need to use the [Mask](#) property for specifying the data entry format.

For example, setting the Mask property for a MaskedTextField control to "90/90/0000" lets users enter date in international date format. Here, the "/" character works as a logical date separator.

The following table enlists mask symbols supported by the MaskedTextField control.

Mask Symbol	Description
0	Digit
9	Digit or space
#	Digit, sign, or space
L	Letter
?	Letter, optional
C	Character, optional
&	Character, required
l	Letter or space
A	Alphanumeric
a	Alphanumeric or space
.	Localized decimal point
,	Localized thousand separator
:	Localized time separator
/	Localized date separator
\$	Localized currency symbol
<	Converts characters that follow to lowercase
>	Converts characters that follow to uppercase
	Disables case conversion
\	Escapes any character, turning it into a literal
All others	Literals.