ComponentOne

Xamarin Edition

Table of Contents

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

	FlexChart and FlexPie controls in your Xamarin.iOS application.
C1.iOS.Grid	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.
C1.iOS.Gauge	Installing this NuGet package adds all the references that enable you to use the Gauge control in your Xamarin.iOS application.
C1.iOS.Input	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.

Calendar

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

CollectionView

- C1.CollectionView.dll

Core

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

FlexChart

- C1.Android.Chart.dll
- C1.Xamarin.Forms.Chart.dll
- C1.Xamarin.Forms.Chart.Platform.Android.dll
- C1.Xamarin.Forms.Chart.Platform.iOS.dll
- C1.Xamarin.Forms.Chart.Platform.UWP.dll
- C1.iOS.FlexChart.dll
- C1.UWP.FlexChart.dll

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

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.grapecity.com/en/licensing/grapecity/.

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

- Licensing App using GrapeCity ComponentOneMenu Extension
- Licensing App using Website

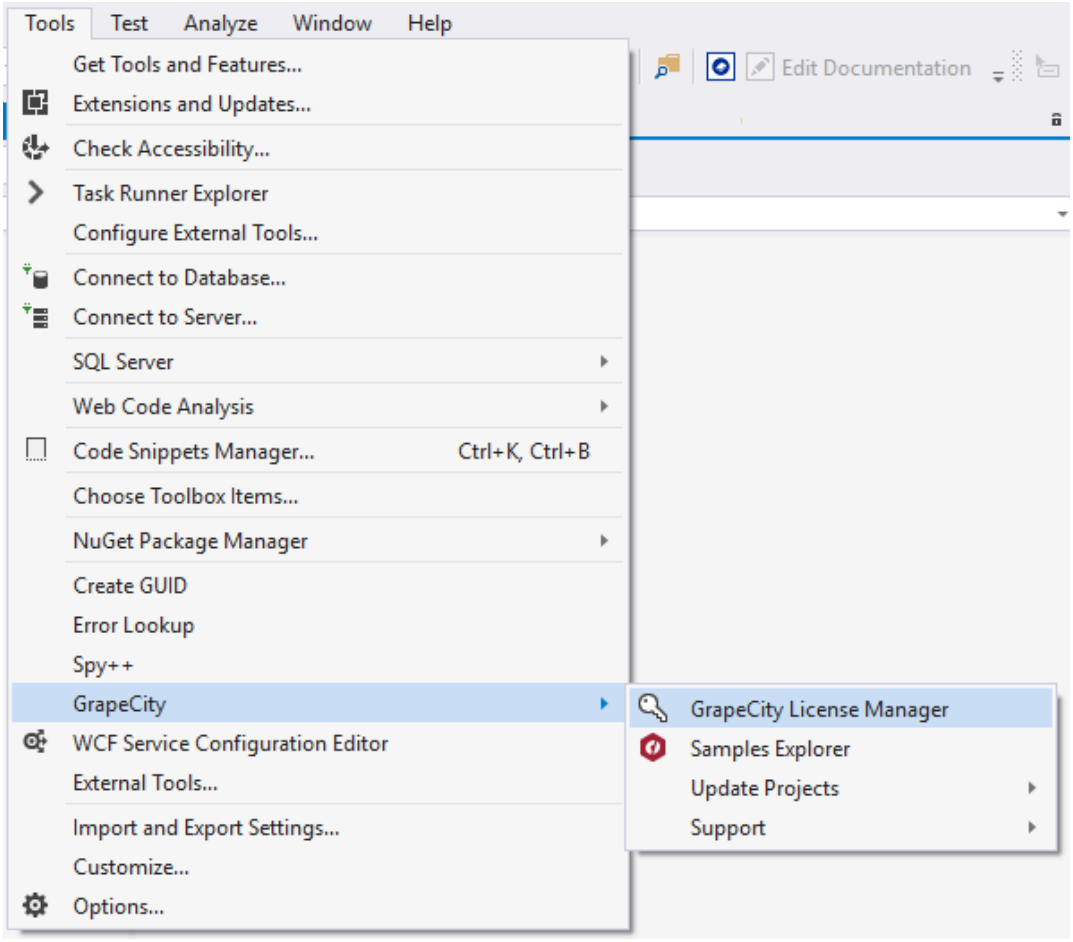
License App using GrapeCity ComponentOneMenu Extension

If you are using ComponentOne controls with Visual Studio 2017 or above, you can use the GrapeCity ComponentOneMenu Extension to **License applications**, open **Sample Explorer**, **update projects** and contact **Technical Support**. After installation, it gets available with various options as shown below.

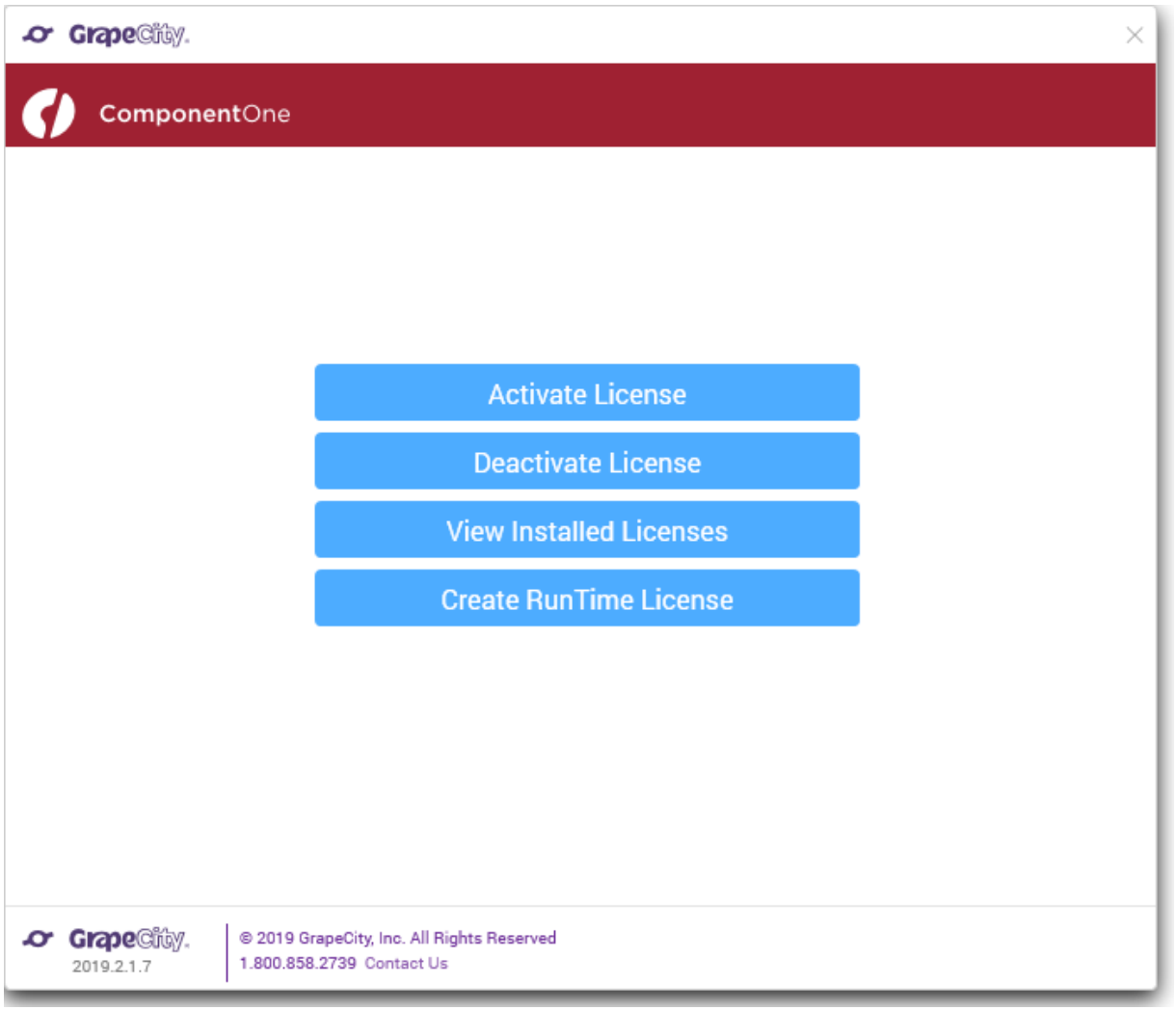
GrapeCity License Manager Extension for Visual Studio

To use GrapeCity License Manager Extension, follow these steps:

1. From the **Tools** menu, select **GrapeCity**. You will see four options as shown in the image below.

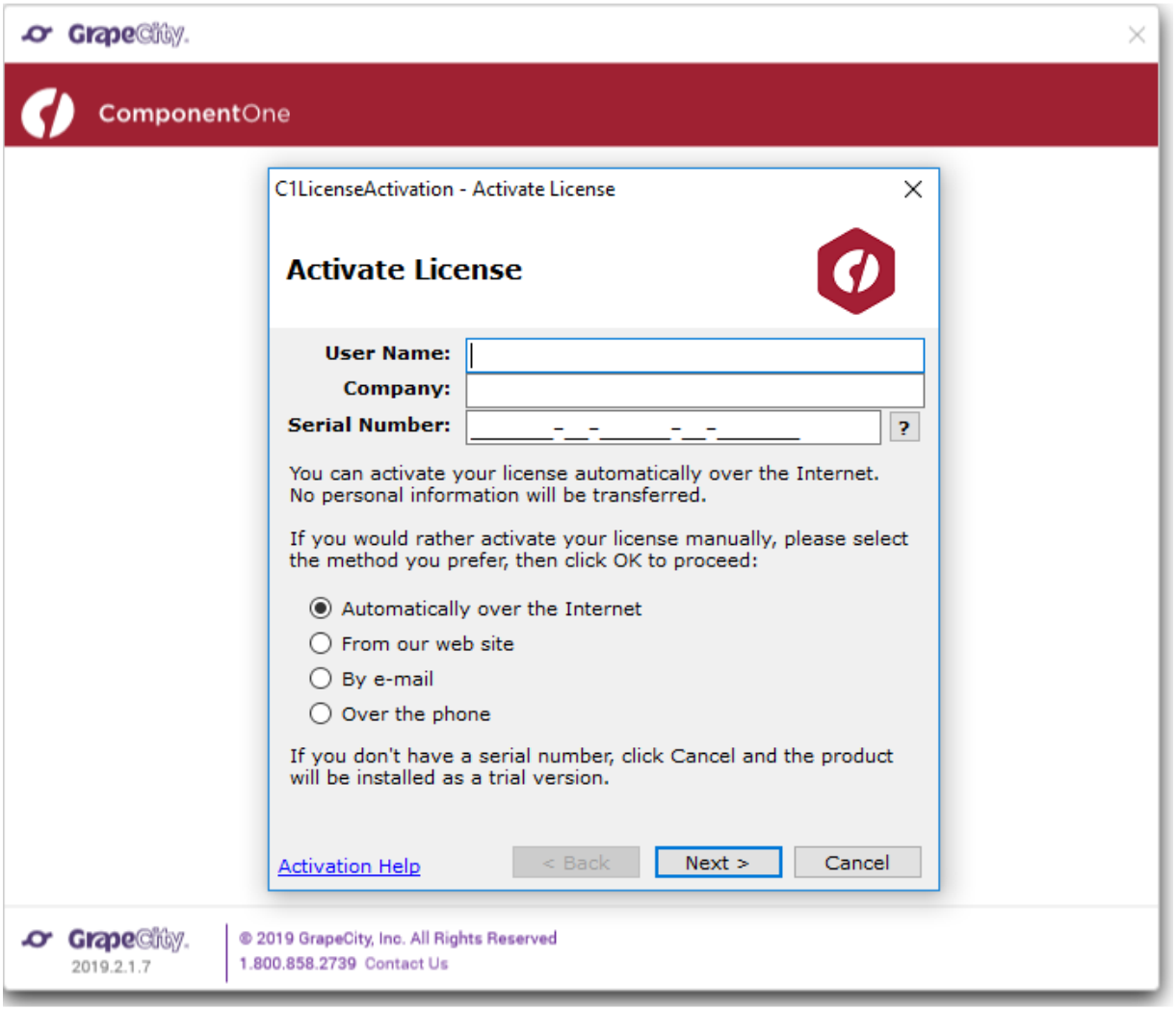


2. Select **GrapeCity License Manager**. The License Manager window appears as shown in the image below:

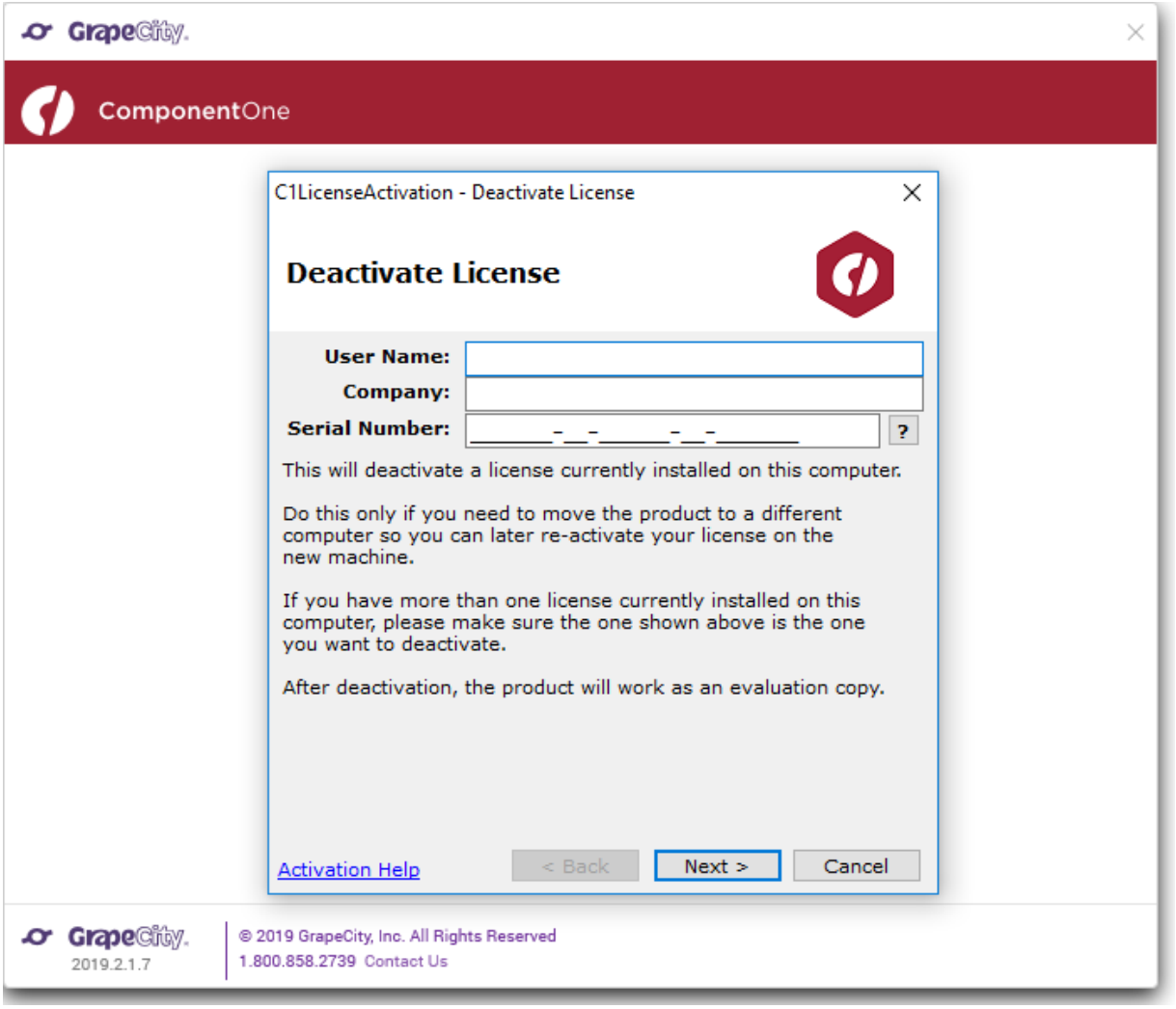


The **GrapeCity License Manager** displays the following options:

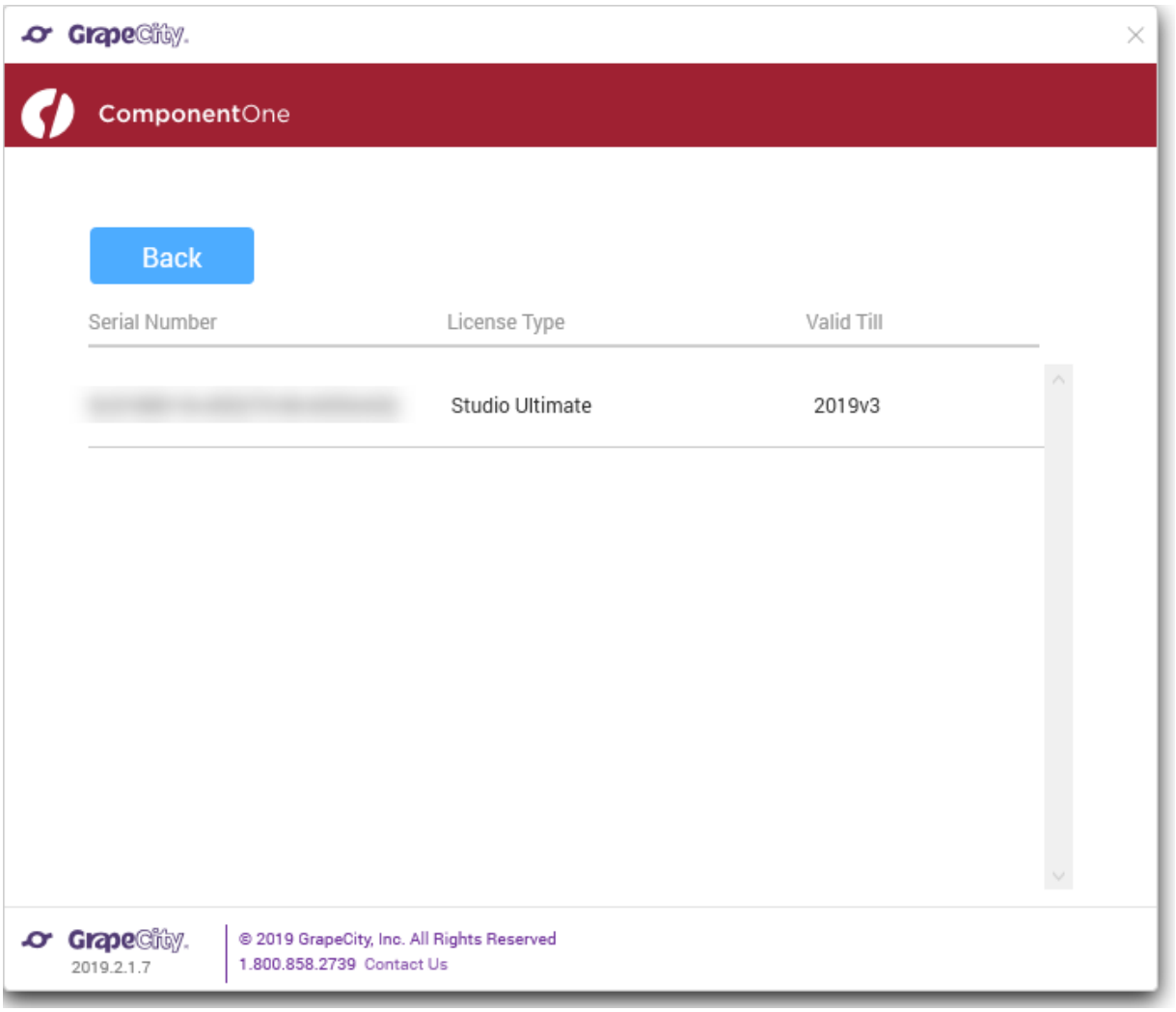
- **Activate License -** It allows the users to activate the License (**Serial Number**) using the Internet, ComponentOne website, e-mail, or over phone. On clicking this option, **C1LicenseActivation - Activate License** application window appears as shown below. Users can follow the wizard to do the license activation.



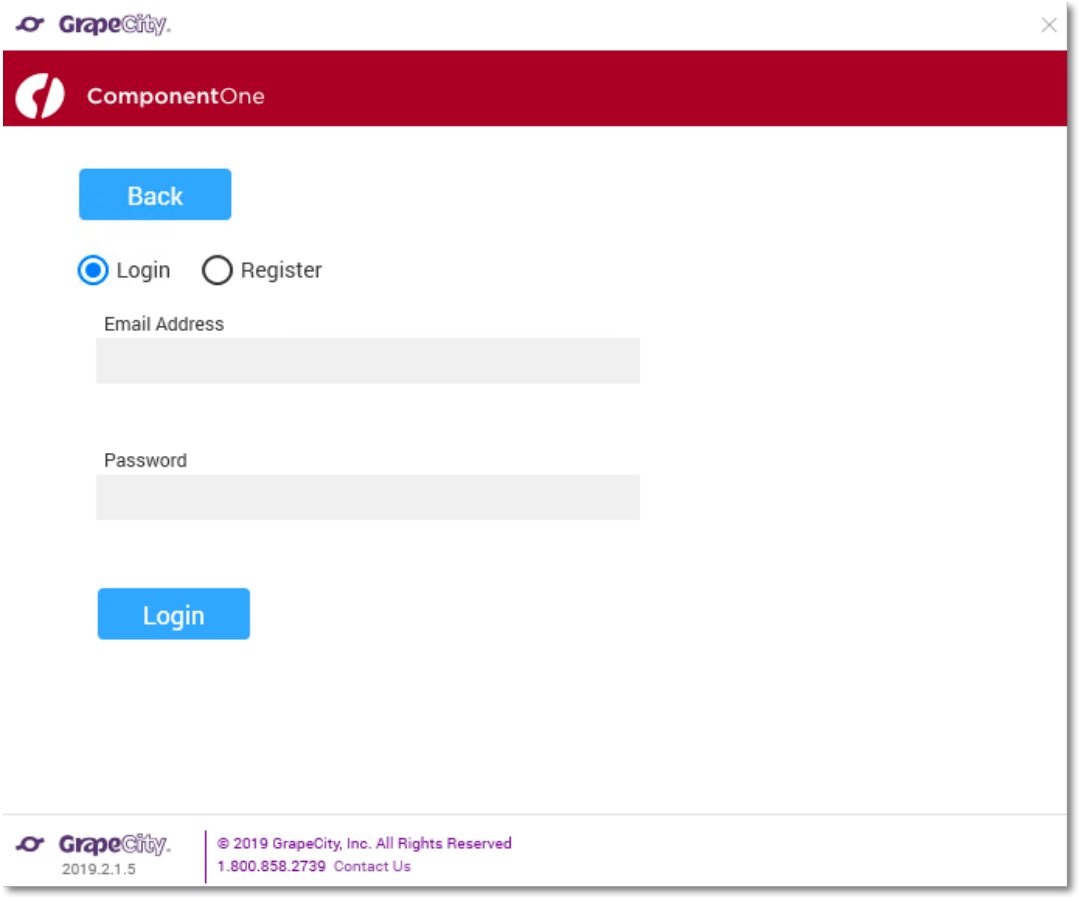
- **Deactivate License** – It allows the users to deactivate the License (**Serial Number**) using the Internet, ComponentOne website, e-mail, or over phone. On clicking this option, **C1LicenseActivation - Deactivate License** application will appear as shown below. Users can follow the wizard to do the license deactivation.



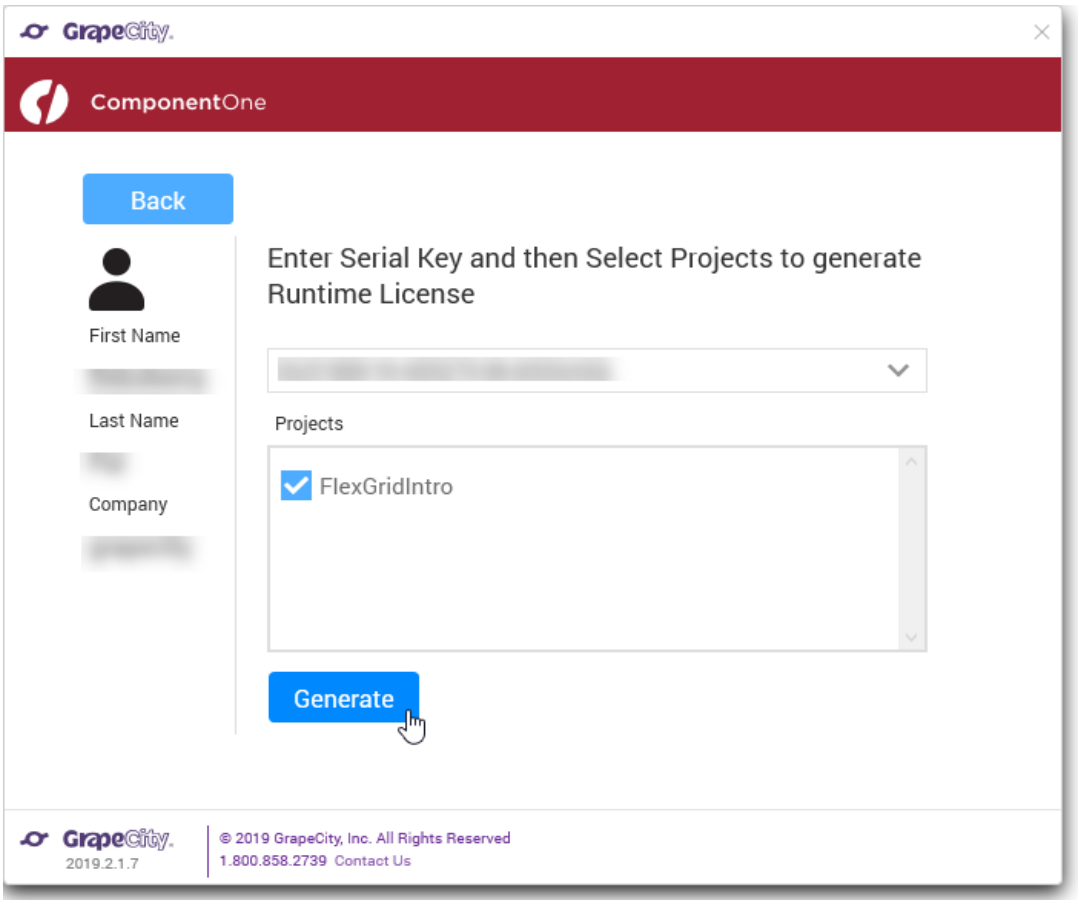
- **View Installed Licenses** - It allows the users to view the **Serial Number**, **License Type**, and **Validity** of the installed licenses on the system. The license can be viewed as shown below:



- **Create RunTime License** - It allows the users to generate runtime license for the project(s). On clicking this option, the following window appears:



To use this option, you need to **login** the **GrapeCity Account**. In case you don't have a GrapeCity Account, you can create it using the **Register** option. Once you Sign in, the following window appears:

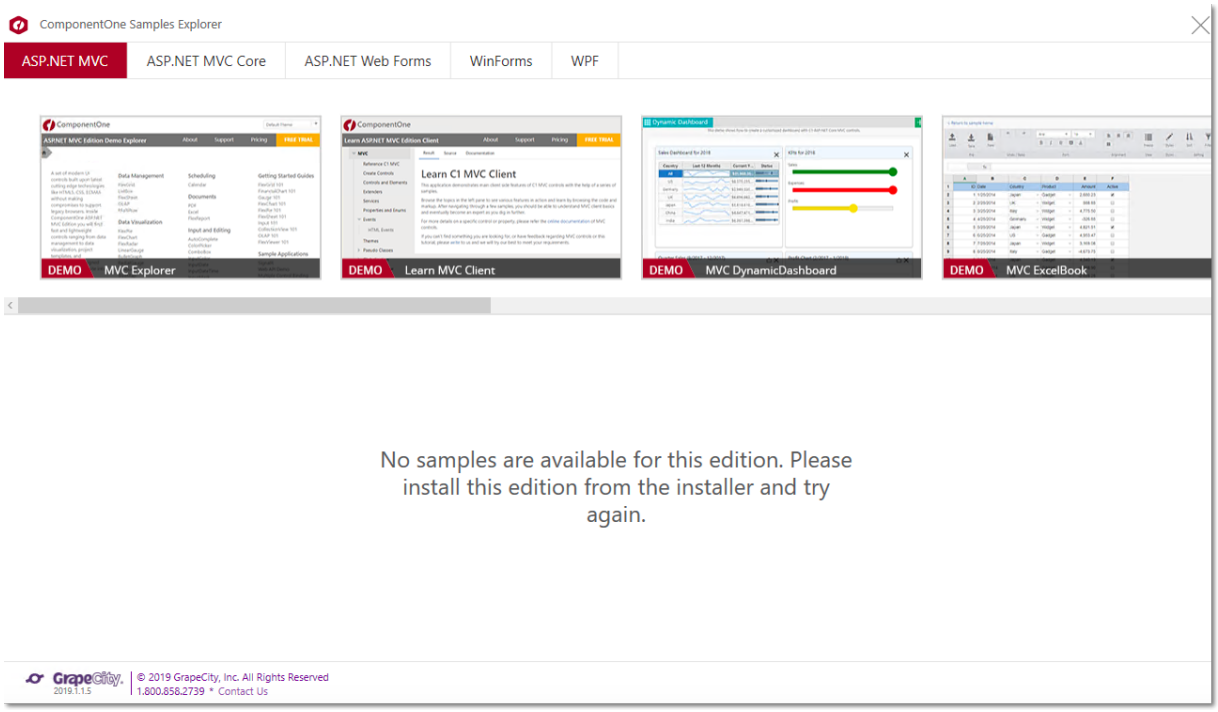


From the above window, you can select the license and the project for which license needs to be generated. On clicking the **Generate** button, a success message appears and a license file **GCDTLicenses.xml** is generated.

Samples Explorer

To view sample explorer, follow these steps:

1. From the **Tools** menu, select **GrapeCity**.
2. Select **Samples Explorer**. **ComponentOne Samples Explorer** window appears as shown in the image below:

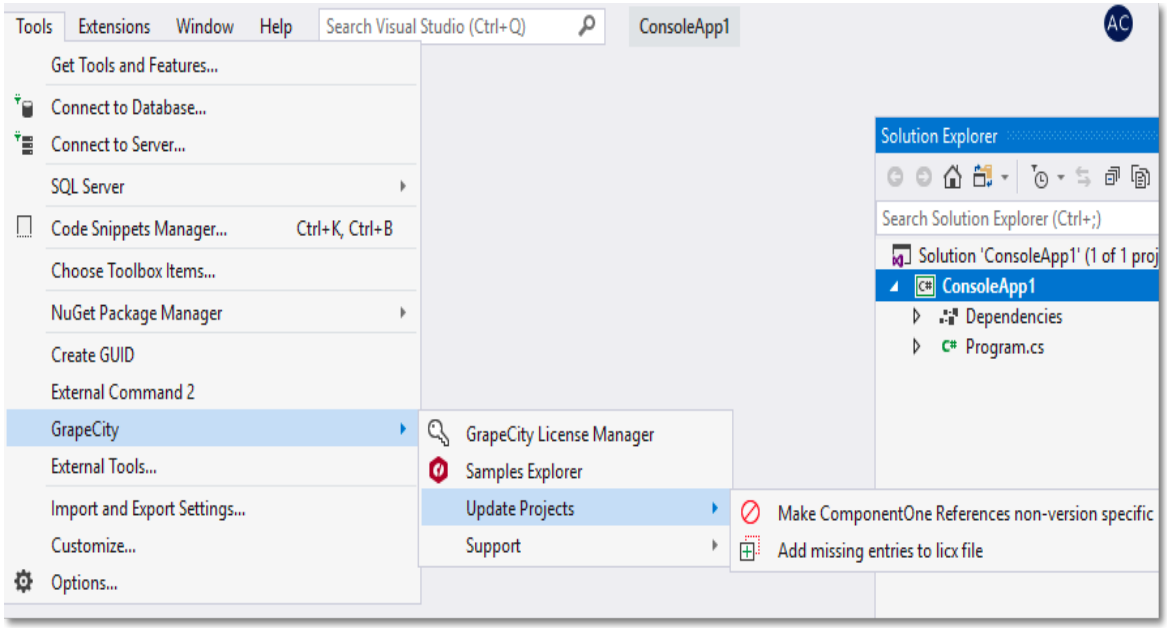


This option allows the users to open the Sample Explorer that has been installed on the system with the installer. From the displayed window, you can open any of the required samples or demos.

Update Projects

To update existing projects, follow these steps:

1. From the **Tools** menu, select **GrapeCity**.
2. Select **Update Projects**. Update Projects provides two options to update the project(s) as shown and listed below:

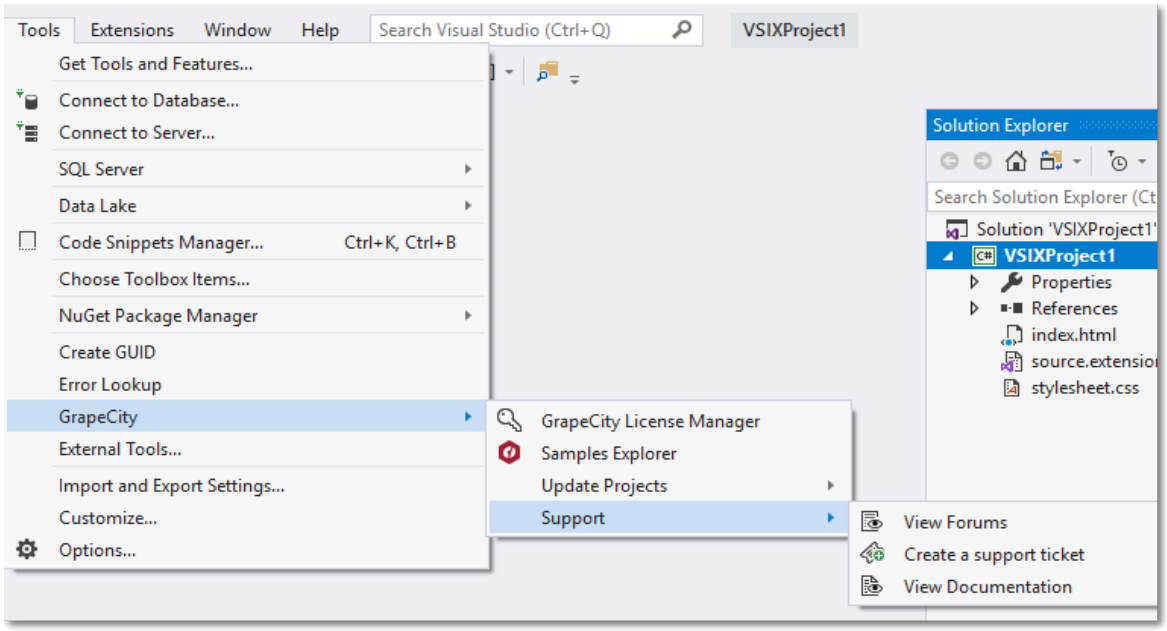


- **Make ComponentOne References non-version specific** - This option removes the version information from the Licenses.licx file and sets the **Specific Version** property of ComponentOne assemblies in the project(s) to **false**. On clicking this option, a window appears where you can select the **Project,** click **Update**, and click **Finish** to complete the update process.
- **Adding missing entries to Licx file -** This option adds missing license information to the license file (*. licx) in the project(s). On Clicking, a window appears where you can select the **Project,** click **Update**, and click **Finish** to update the license files.

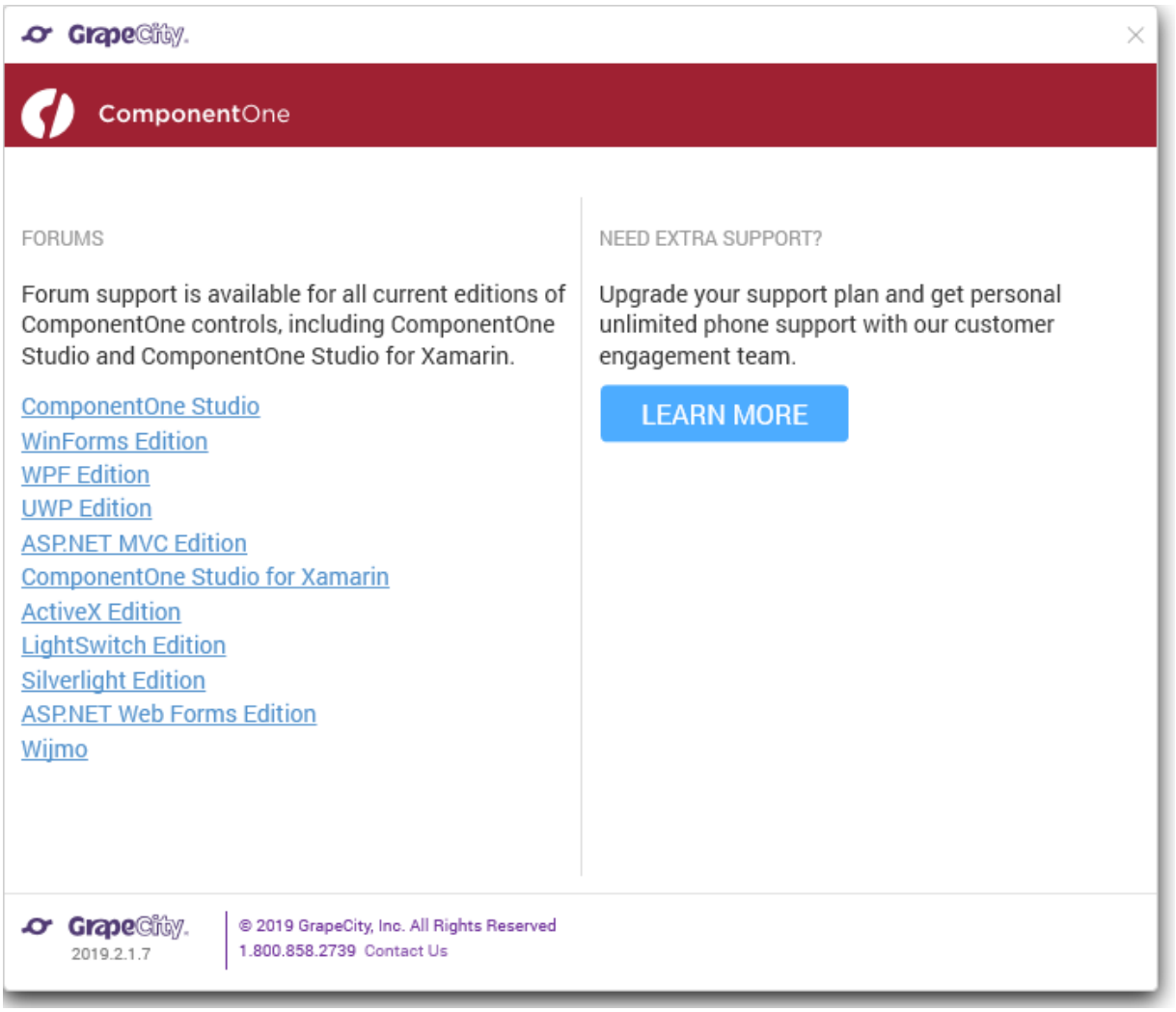
Support

To contact the Technical Support and view Documentation, follow these steps:

1. From the **Tools** menu, select **GrapeCity**.
2. Select **Support**. Support provides three options as shown and listed below.



- **View Forums** - This option opens the window shown below, which has the redirecting links to the GrapeCity ComponentOne Forums.



You can login to access the community forums for all GrapeCity products and post queries/doubts related to the products. You can also find information on how to contact the support team by clicking **Learn More**.

- **Create a support ticket -** This option redirects you to the Support Portal where you can login and get in touch with the support team directly. You can also post queries/doubts related to the products to get personalized support.
- **View Documentation -** This option redirects you to the Documentation page where you can access all the documents related to all the GrapeCity products.

Note: If you are using Visual Studio 2015 or previous versions of Visual Studio, you can use the GrapeCity License Manager add-in. The add-in is available in Tools menu in Visual Studio. For detailed information on using the GrapeCity License Manager add-in, see Generate License using GrapeCity License Manager Add-in.

Licensing 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 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.grapecity.com/en/my-account/create-app-key.

   > **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 Item...**. The **Add New Item** dialog appears.
   4. Under installed templates, select **Visual C# | Class**.
   5. Set the name of the class as `License.cs` and click **Add**.
   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.xaml.cs` and set the runtime license, inside the `constructor App()` method as shown below.

      C#

      ```
      C1.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.

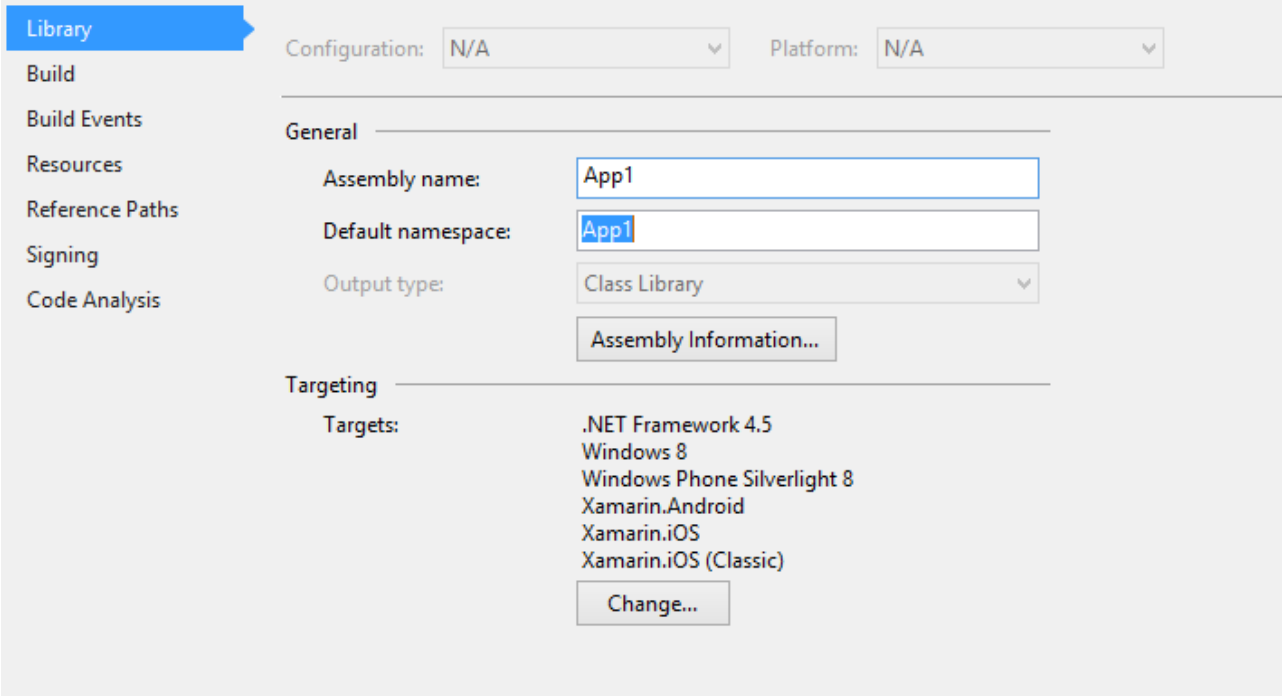
> The trial period is limited to 30 days, which begins when you generate your first runtime license. The controls will stop working after your 30 day trial has ended. You can extend your license by contacting our sales team.

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

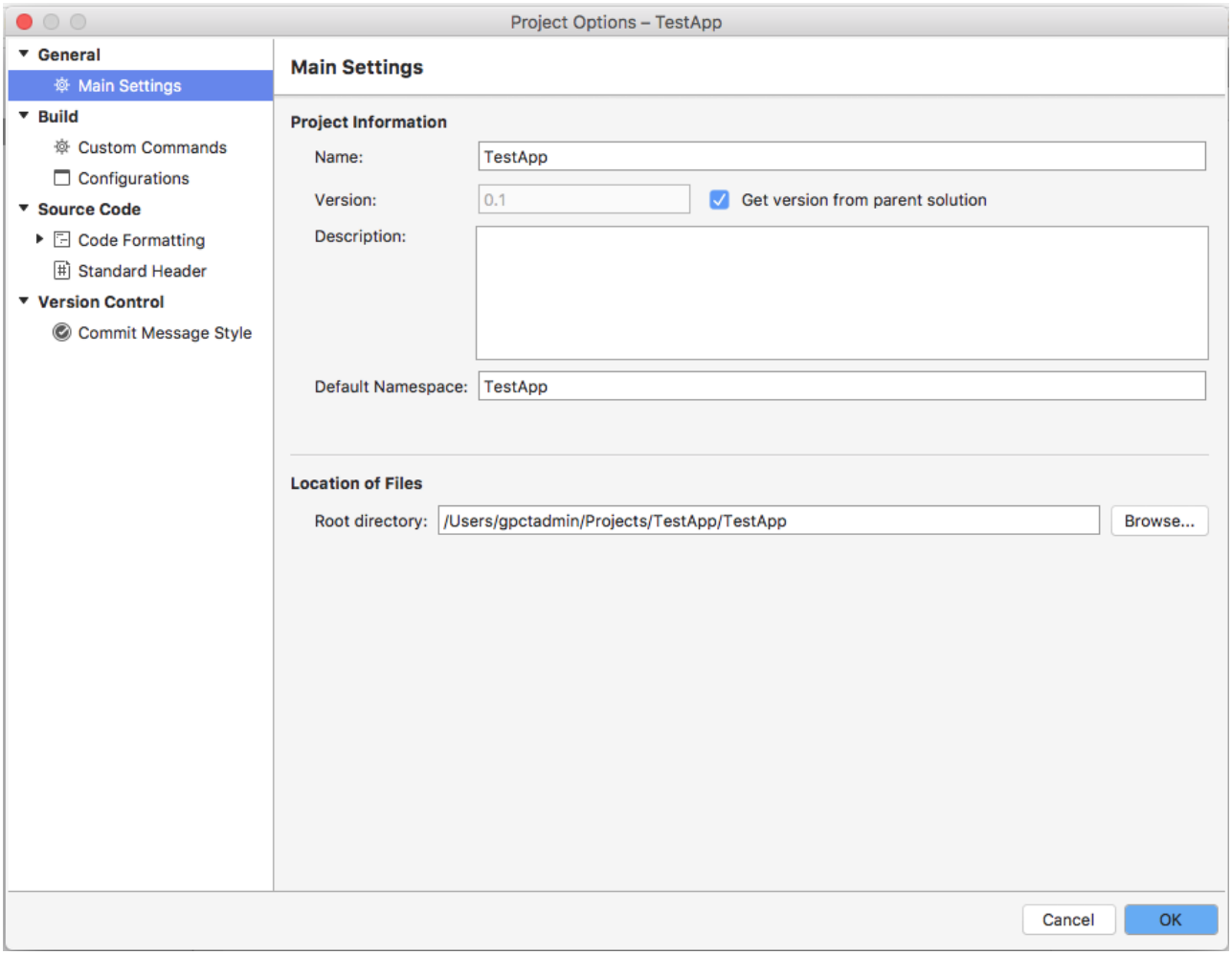
1. Open a pre-existing Mobile App.
2. In the **Solution Explorer**, right click the project YourAppName (Portable or Shared) and select **Properties**.
3. Open the **Library** tab.
4. The application name is the same as the **Default Namespace** displayed.



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

Visual Studio for Mac

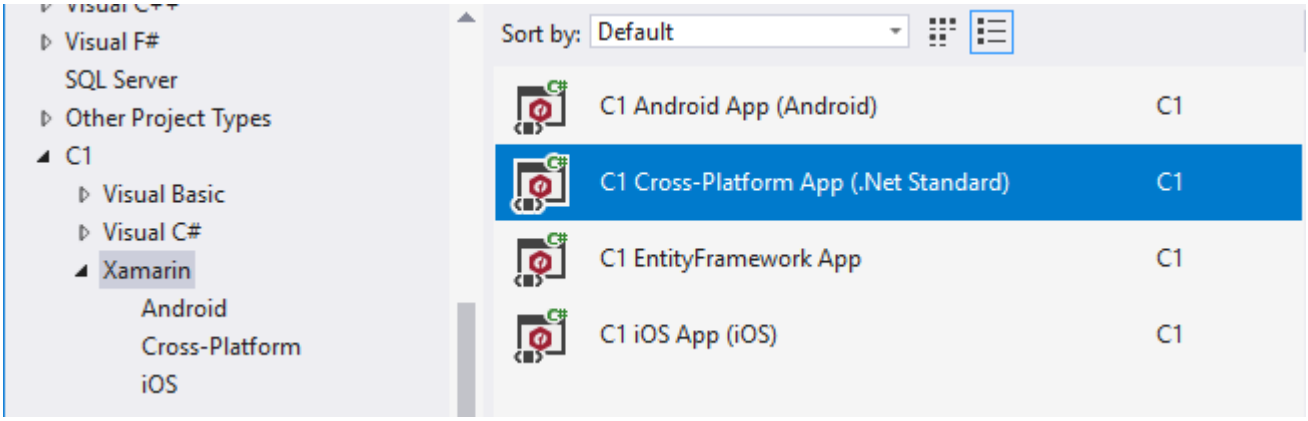
1. Open a pre-existing Mobile App.
2. In the **Solution Explorer**, right click the project YourAppName (Portable or Shared) and select **Options**.
3. The application name is displayed on the **Main Settings** tab.
4. The application name is the same as the **Default Namespace** displayed.



C1 Xamarin.Forms Project Templates

When you install **C1StudioInstaller.exe** on your system, we provide different project templates that are available for Xamarin.Forms, Android, and iOS. These templates help you to easily create a Xamarin project with renderer initializers already in place to make use of controls to make development easier. These templates will also help the new users to get started with Xamarin.Forms using C1 Xamarin controls.

- **Xamarin.Forms**
  - Android
  - Cross-Platform
  - iOS



These project templates are similar to the Xamarin.Forms template available in Visual Studio. However, our templates provide a simple mobile application that includes our control references. These generated applications contains all the necessary project files and assembly references to reduce manual work in project creation.

The table below list the C1 Xamarin.Forms project templates.

Project Template (Visual C#)	Description
C1 Android App (Android)	Creates a Xamarin.Android application using ComponentOne Studio for Xamarin. It includes the necessary project files, assembly references, and license information.
C1 Cross-Platform App (Xamarin.Forms Portable)	Creates a Xamarin.Forms application using a Portable Class Library and ComponentOne Studio for Xamarin. It includes the necessary project files, assembly references, and license information.
C1 EntityFramework App	Creates a Xamarin.Forms project using a data grid, Entity Framework, and a SQLite database that allows adding, removing, and updating records.
C1 iOS App (iOS)	Creates a Xamarin.iOS application using ComponentOne Studio for Xamarin. It includes the necessary project files, assembly references, and license information.

Creating a New Xamarin.Forms App

This topic demonstrates how to create a new cross platform app in Visual Studio or Visual Studio for MAC. See the System Requirements before proceeding.

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

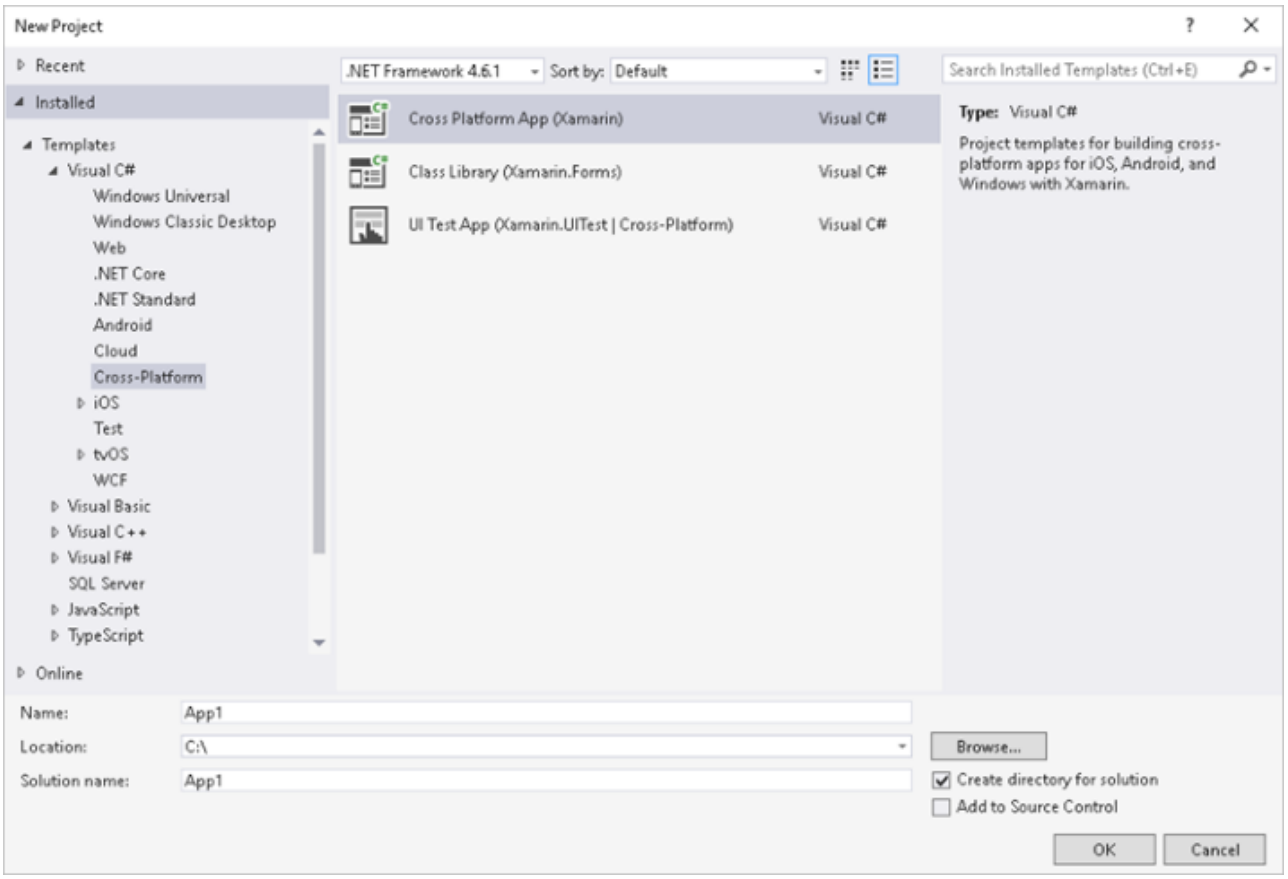
http://developer.xamarin.com/guides/cross-platform/application_fundamentals/building_cross_platform_applications/sharing_code_options/

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

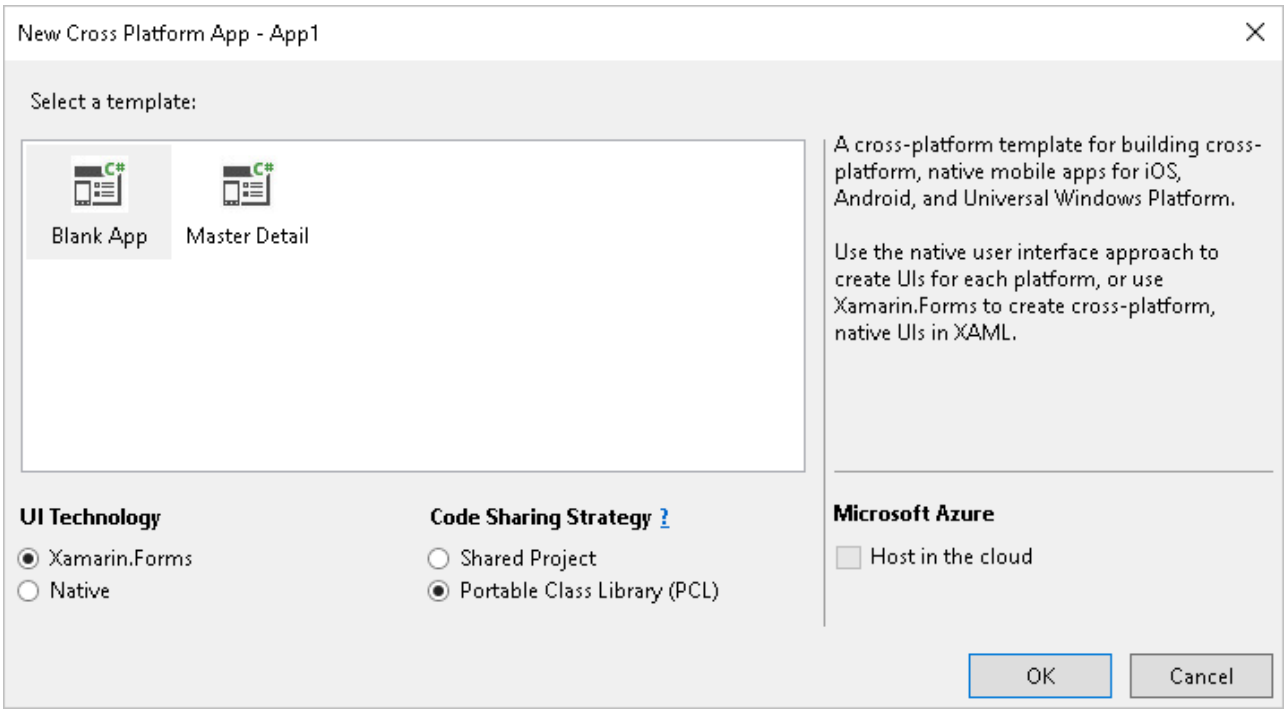
Visual Studio

1. Select **File | New | Project**.

2. Under **Installed** templates, select **Visual C# | Cross-Platform**.
3. In the center pane, select **Cross Platform App** (**Xamarin**).
4. Add a name for your app and select a location to save it.

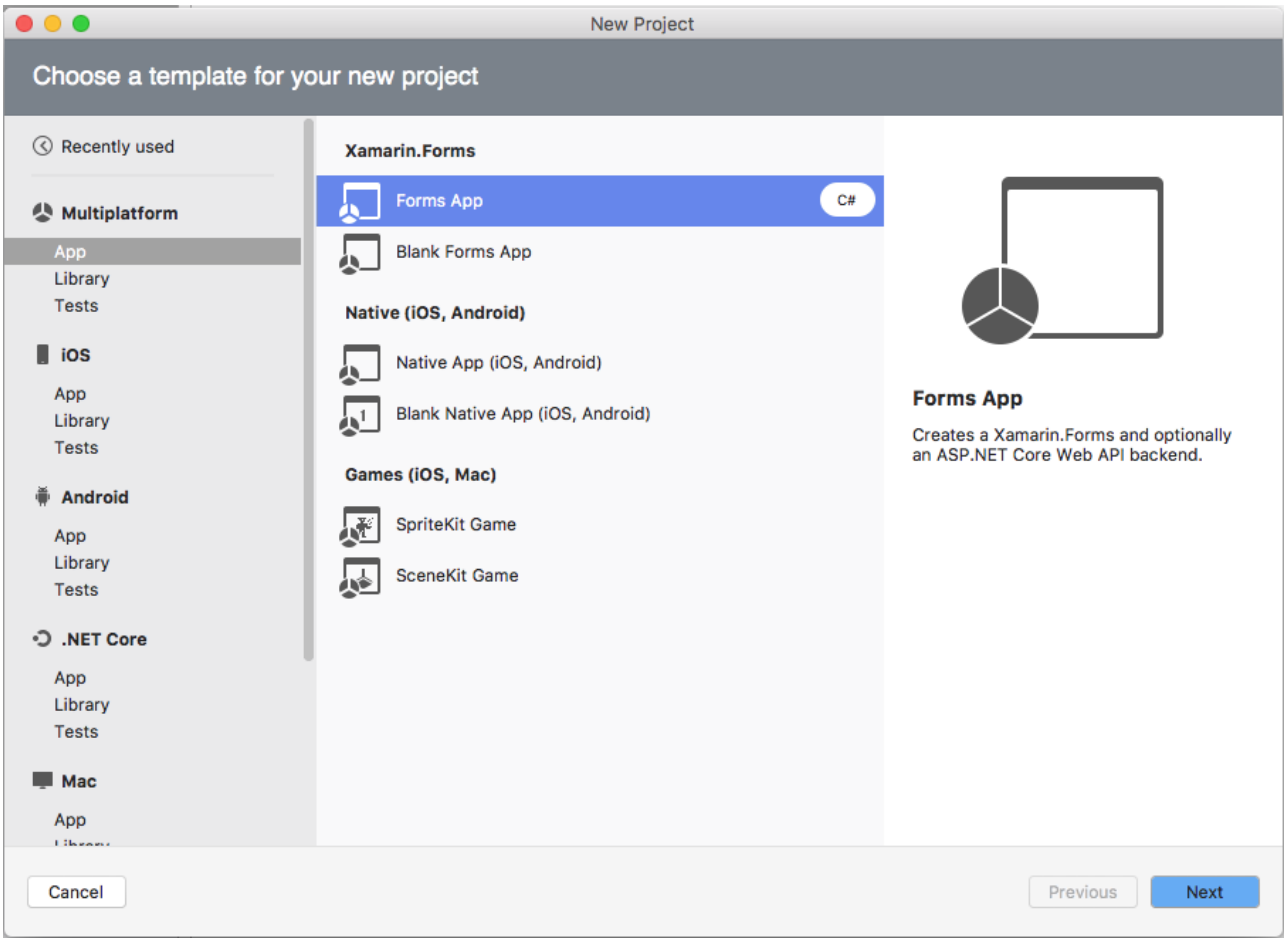


5. Click **OK**. A **New Cross Platform App** dialog appears.
6. Select **Blank App** from the available templates and choose either **Shared Project** or **Portable Class Library** from the **Code Sharing Strategy** options.
7. Click **OK**.



Visual Studio for Mac

1. Select **New Project.**
2. In the New Project window, select **Multiplatform | App**.
3. In the right pane, select **Forms App** or **Blank Forms App**, and then click **Next**.
4. Enter an **App Name** for your Forms App.
5. Click **Create** to create a new Forms App.



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.

Adding NuGet Packages to your App

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 **Cross Platform App** or create a new **Cross Platform App**.For more information, see Creating a New Xamarin.Forms App.
2. From the **Project** menu, select **Manage NuGet Packages**. The **NuGet Package** Manager appears.
3. Select **GrapeCity** from the **Package source** drop down.

4. Click **Browse** tab and select **C1.Xamarin.Forms.ControlName** (for example C1.Xamarin.Forms.FlexChart) from the left pane.
5. In the right pane, click **Install**. This adds the references for the Xamarin control.
6. In the **Preview** dialog, click **OK** to proceed with the changes.
7. Click **I Accept** to accept the license terms. The license gets installed.

Visual Studio for Mac

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 Visual Studio for MAC 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. Select the newly created package source and click the **Browse** tab. The available packages are displayed in the left pane where you can search for the control using the **Search** box.
4. In the right pane, click **Install**. This updates the references for the Xamarin control.
5. In the **Preview** dialog, click **OK** to proceed with the changes.
6. Click **I Accept** to accept the ComponentOne license for Xamarin and then click **Close** in the Manage NuGet Packages dialog box.

Visual Studio for Mac

1. From the **Projects** menu, select **Add NuGet 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 **GrapeCity**. 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 Item...**. The **Add New Item** dialog appears.
3. Under installed templates, select **Visual 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 C1.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 C1LinearGauge ReturnMyControl()
{
    // Instantiate LinearGauge and set its properties
    C1LinearGauge gauge = new C1LinearGauge();
    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.xaml.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:
   - **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();
        ```

   - **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 Content Page**
- **Step 2: Add the Control**
- **Step 3: Run the Program**

Step 1: Add a new Content Page

1. In the **Solution Explorer**, right click the project YourAppName (Portable or Shared).
2. Select **Add | New Item...**. The **Add New Item** dialog appears.
3. Under installed templates, select **C#** | **Content 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:c1="clr-namespace:C1.Xamarin.Forms.Gauge;assembly=C1.Xamarin.Forms.Gauge">
</ContentPage>
```

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>
  <c1:C1LinearGauge  Value="35" Min="0" Max="100" Thickness="0.1"
   HeightRequest="50" WidthRequest="50" 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 3: Run the Program

1. In the **Solution Explorer**, double click App.cs to open it.
2. In the class constructor App(), set 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#
        ```
        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

About this Documentation

Acknowledgements

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 and PDF files can be downloaded for offline viewing. If you have suggestions on how we can improve our documentation, please send feedback to support forum.

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. The control 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.
- **Animation**: By default, Calendar confers fast, animated transitions for enhanced user experience.

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 C1.Xamarin.Forms.Calendar;
```

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

C#

```
public static C1Calendar ReturnMyControl()
    {
        C1Calendar calendar = new C1Calendar();
        calendar.MaxSelectionCount = -1;
        calendar.HorizontalOptions = LayoutOptions.FillandExpand;
        calendar.FontSize = 20;
        return calendar;
    }
```

In XAML

1. Add a new Content 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"
```

```
xmlns:c1="clr-namespace:C1.Xamarin.Forms.Calendar;assembly=C1.Xamarin.Forms.Calendar"
x:Class="CalendarQuickStart.QuickStart"
Padding="20">
</ContentPage>
```

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

XAML

```
<Grid>
    <Label Text="{Binding MainText}" HorizontalOptions="Center" Font="Large" />
    <c1:C1Calendar x:Name="calendar" MaxSelectionCount="-1"/>
</Grid>
```

Step 2: Run the Project

1. In the Solution Explorer, double-click App.xaml.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 Content Page**: In the constructor App(), set the Content 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();
        ```

     3. (Optional) In case you compile your UWP application in **Release** mode, you need to explicitly add the following code to the **OnLaunched** method in your **App.xaml.cs** to include the correct assemblies with your application.

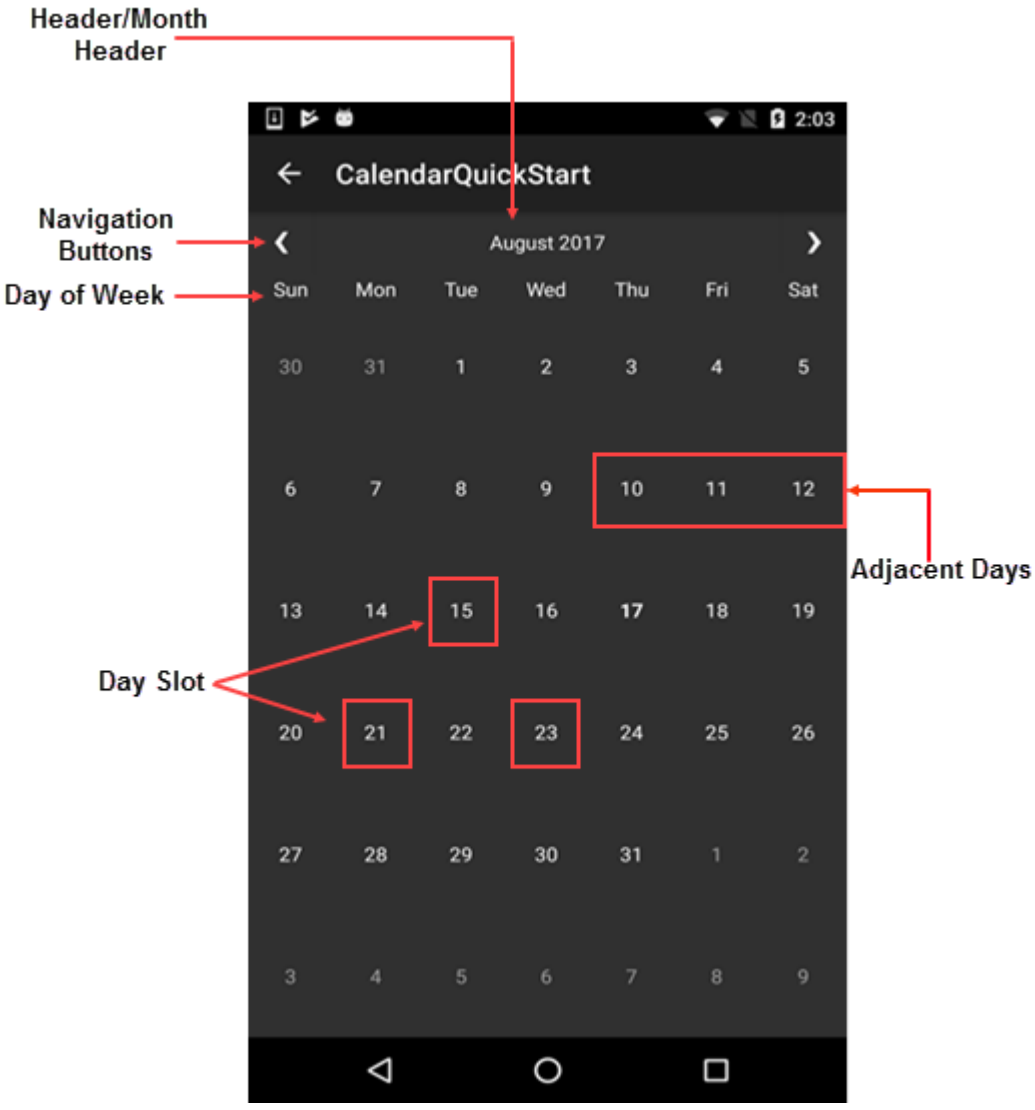
        C#

        ```
        var assembliesToInclude = new List<Assembly>();
        assembliesToInclude.Add(typeof(C1.Xamarin.Forms.Calendar.Platform.UWP.C1CalendarRenderer)
        .GetTypeInfo().Assembly);
        assembliesToInclude.Add(typeof(C1.UWP.Calendar.C1Calendar).GetTypeInfo().Assembly);
        Xamarin.Forms.Forms.Init(e, assembliesToInclude);
        ```

Interaction Guide

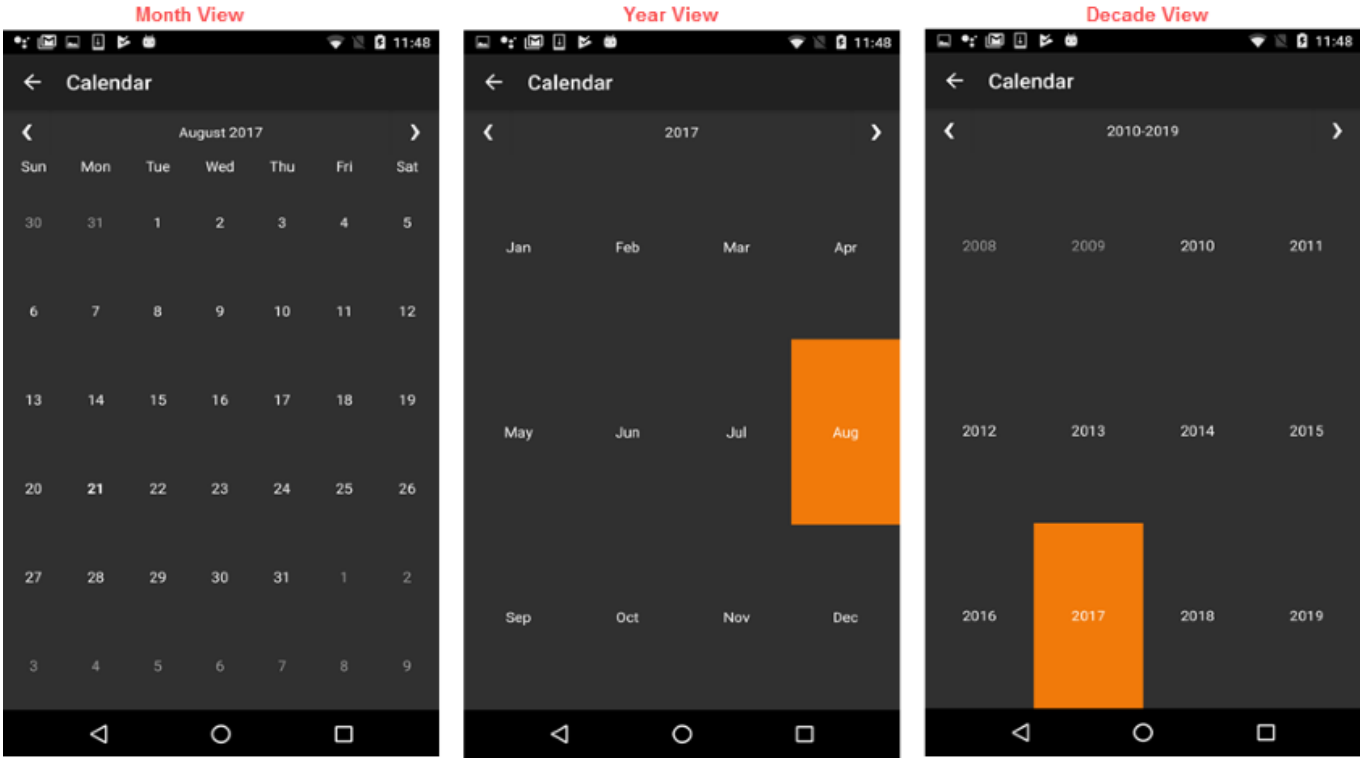
The Calendar control comprises of various functional parts such as Header, Day of Week, Day Slot, etc. These functional parts are illustrated below.

- **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.

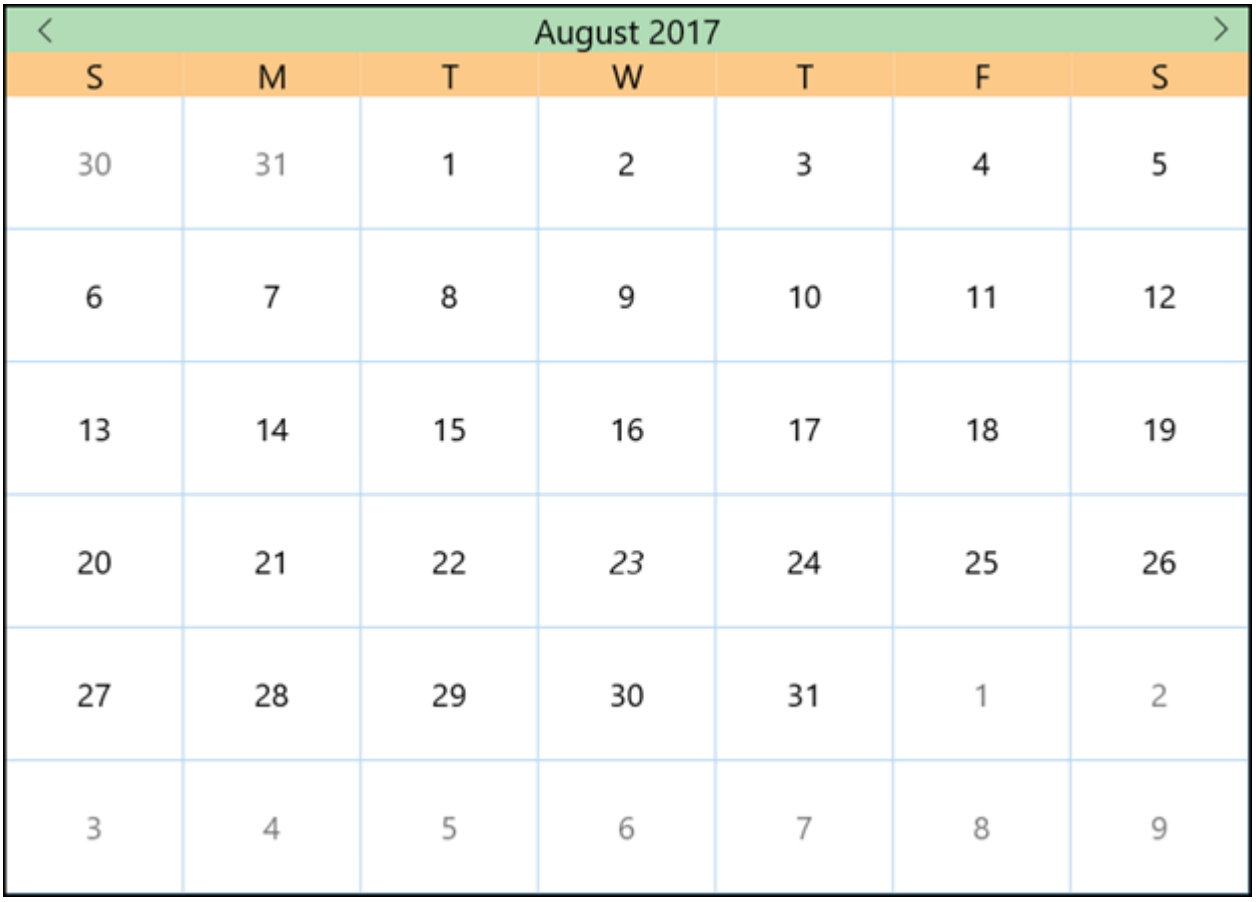


Features

Customizing Appearance

The C1Calendar 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.



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 section.

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.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

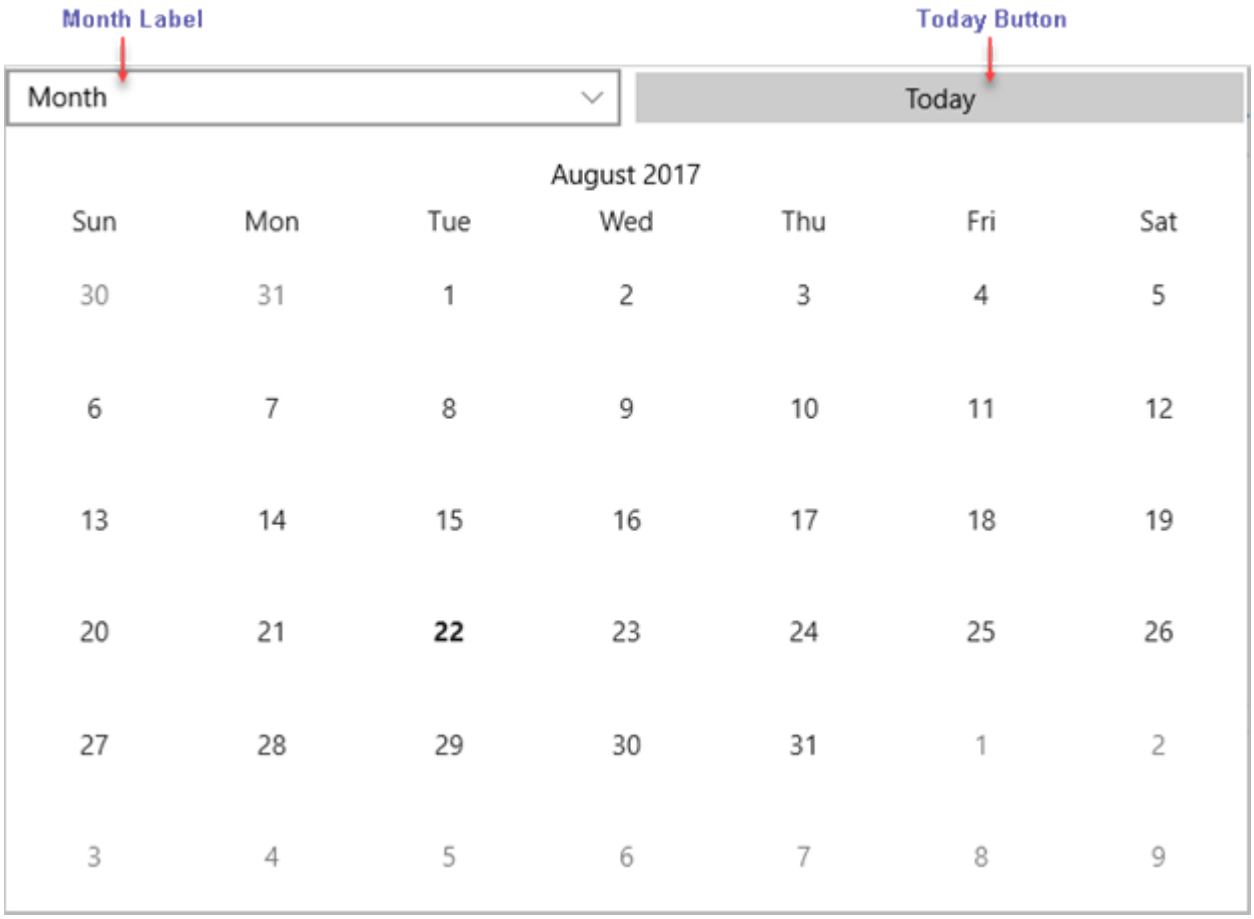
XAML

```
<c1:C1Calendar 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 section.

1. Add a new Content 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:c1="clr-
namespace:C1.Xamarin.Forms.Calendar;assembly=C1.Xamarin.Forms.Calendar"
             x:Class="C1CalendarCustomHeader.CustomizingHeader" x:Name="page">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
        </Grid.RowDefinitions>
        <Grid Grid.Row="0" Margin="0,30,0,0">
            <Grid.ColumnDefinitions>
                <ColumnDefinition />
```

```
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <Picker x:Name="modePicker"/>
            <Button x:Name="todayButton" Text="Today" Grid.Column="1"/>
        </Grid>
        <StackLayout Orientation="Vertical" Grid.Row="1" Margin="0,10,0,0">
            <Label x:Name="monthLabel" HorizontalOptions="Center"/>
            <c1:C1Calendar x:Name="calendar" ShowHeader="False"
VerticalOptions="FillAndExpand" />
        </StackLayout>
    </Grid>
```

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#

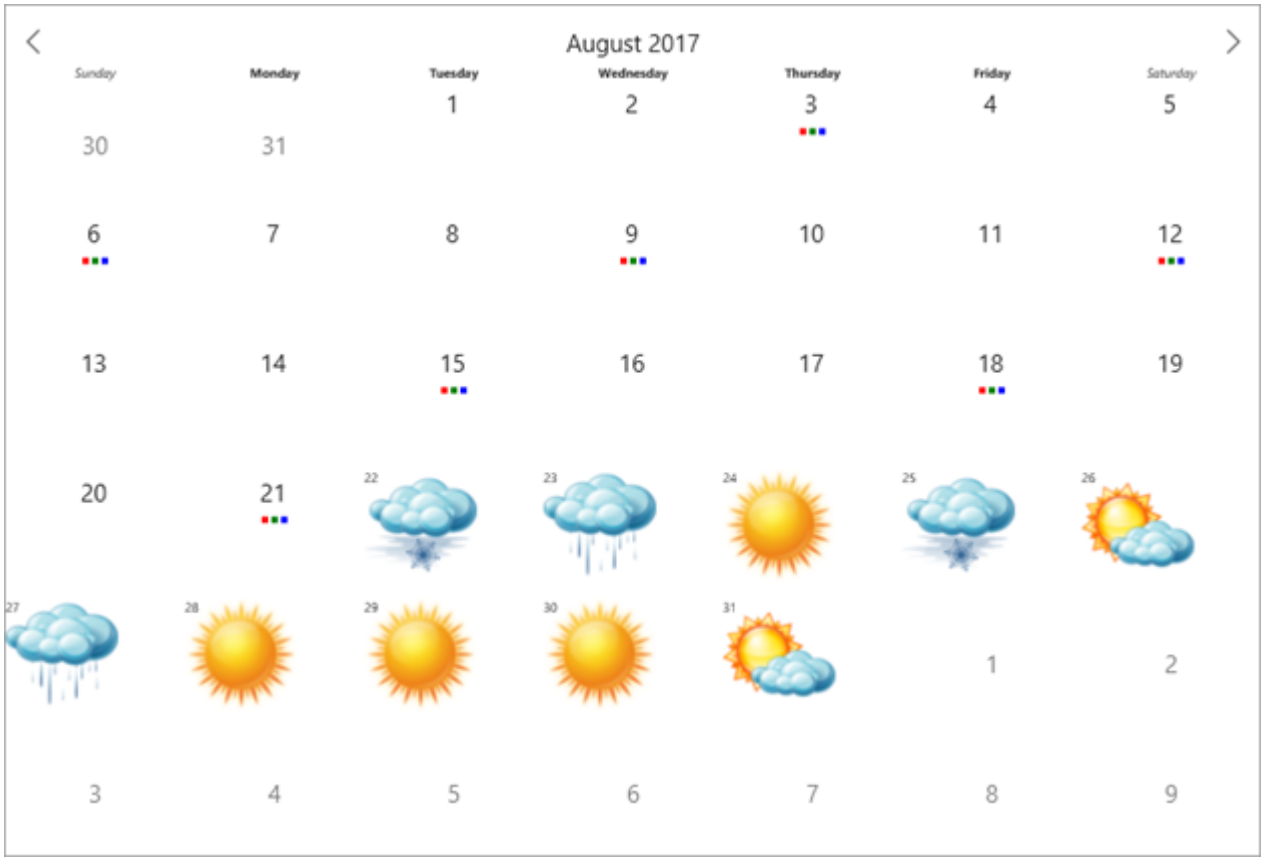
```
public partial class CustomizingHeader : ContentPage
    {
        public CustomizingHeader()
        {
            InitializeComponent();

            modePicker.Items.Add("Month");
            modePicker.Items.Add("YearLabel");
            modePicker.Items.Add("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 yyy}",
calendar.DisplayDate);
        }
    }
```

Customizing Day Content

The Calendar control allows users to add custom content to day slot. For this, all you need to do is subscribe to 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 Content Page, CustomDayContent.xaml, to your portable project.
2. 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:c1="clr-
namespace:C1.Xamarin.Forms.Calendar;assembly=C1.Xamarin.Forms.Calendar"
             x:Class="C1Calendar_Xamarin.CustomDayContent">

    <StackLayout Orientation="Vertical">
        <Grid  VerticalOptions="FillAndExpand">
            <c1:C1Calendar DayOfWeekFontSize="8" DayOfWeekFormat="dddd"
                                DayOfWeekFontAttributes="Italic"
DaySlotLoading="OnDaySlotLoading"
                                DayOfWeekSlotLoading="OnDayOfWeekSlotLoading"
VerticalOptions="FillAndExpand">
                <c1:C1Calendar.DaySlotTemplate>
                    <DataTemplate>
                        <StackLayout Padding="4" VerticalOptions="Center">
                            <Label Text="{Binding Day}"
HorizontalOptions="Center"/>
```

```
                            <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>
                        <Label Text="{Binding Day}" HorizontalOptions="Center"
                                VerticalOptions="Center"/>
                    </DataTemplate>
                </c1:C1Calendar.AdjacentDaySlotTemplate>
            </c1:C1Calendar>
        </Grid>
    </StackLayout>
</ContentPage>
```

3. Create a custom class, CalendarDaySlot.cs, for adding custom controls and images along with their properties to the project.
4. Add **Images** folder in the project to add the images of different weather icons. The images used in this project are available at the following location: **Documents\ComponentOne Samples\Xamarin\XF\C1Calendar101\C1Calendar101.Xamarin\Images**
5. Select an image and change the **Build Action** to **Embedded Resource**. Similarly, change the **Build Action** for all the images.
6. In the **Solution Explorer**, open CustomDayContent.xaml.cs file.
7. Add the following import statements in the CustomDayContent.xaml.cs file of your portable project.

C#

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

8. Add the following code in the CustomDayContent.xaml.cs to add custom content to day slots.

In Code

C#

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

```
        public CustomDayContent()
        {
            InitializeComponent();


_icons.Add(ImageSource.FromResource("C1Calendar_Xamarin.Images.partly-cloudy-
day-icon.png"));

_icons.Add(ImageSource.FromResource("C1Calendar_Xamarin.Images.Sunny-
icon.png"));
            _icons.Add(ImageSource.FromResource("C1Calendar_Xamarin.Images.rain-
icon.png"));
            _icons.Add(ImageSource.FromResource("C1Calendar_Xamarin.Images.snow-
icon.png"));

_icons.Add(ImageSource.FromResource("C1Calendar_Xamarin.Images.thunder-
lightning-storm-icon.png"));

_icons.Add(ImageSource.FromResource("C1Calendar_Xamarin.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 Label).FontAttributes = FontAttributes.Bold;
                (e.DayOfWeekSlot as Label).FontSize = 8;
            }
            else
            {
                (e.DayOfWeekSlot as Label).FontAttributes =
FontAttributes.Italic;
                (e.DayOfWeekSlot as Label).FontSize = 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 section.

In Code

C#

```
//Setting the Orientation
calendar.Orientation = CalendarOrientation.Vertical;
```

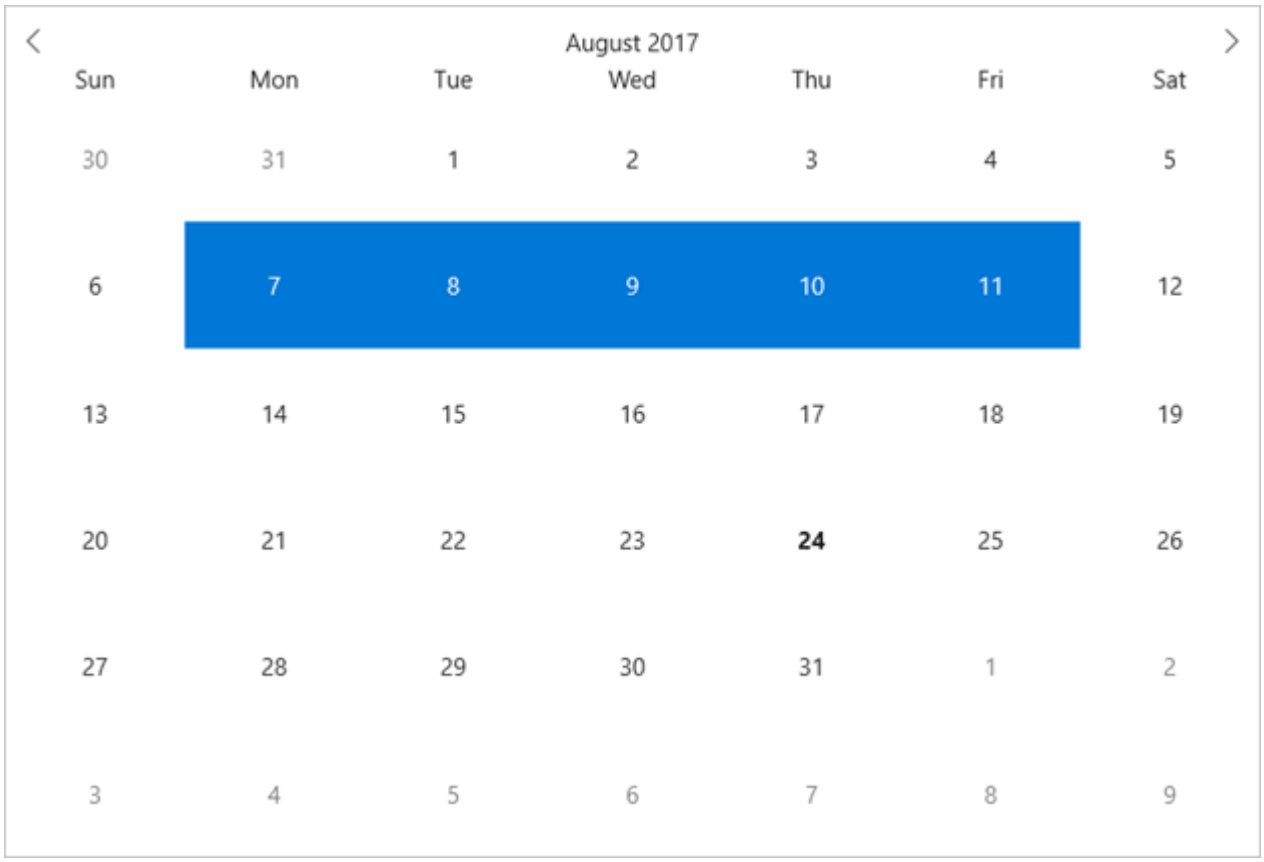
In XAML

XAML

```
<Grid>
     <c1:C1Calendar x:Name="calendar" 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 section.

In Code

C#

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

In XAML

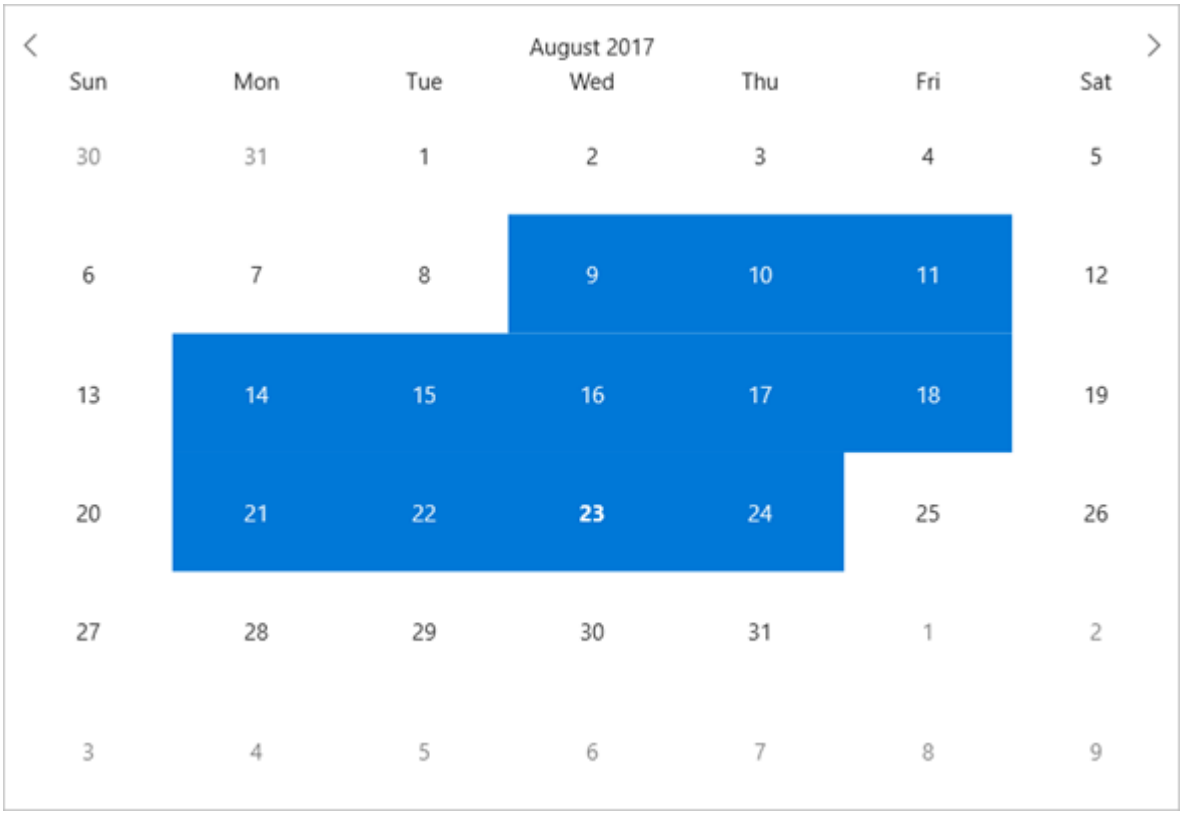
XAML

```
<Grid>
    <c1: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 section.

In Code

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

XAML

```
<Grid>
    <c1: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.

Quick Start: Add data to FlexGrid using CollectionView

This section describes how to add a **FlexGrid** control to your Xamarin application and add data to it using the CollectionView class. For more information on how to add Xamarin components in XAML, see Adding XamarinComponents 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 Project**

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

ID	Name	Address	Country	Country ID	Postal Code	First	La
0	Charlie Neiman	130 Main BLVD	United States	3	705635	Charlie	Ne
1	Gil Jammers	546 Fake ST	Thailand	5	169919	Gil	Ja
2	Steve Neiman	591 Main ST	United States	3	301845	Steve	Ne
3	Xavier Krause	706 Golden BLVD	Brazil	0	686787	Xavier	Kr
4	Steve Neiman	75 Green ST	Japan	4	956574	Steve	Ne
5	Xavier Frommer	346 Park BLVD	Japan	4	469262	Xavier	Fr
6	Herb Orsted	795 Park ST	Thailand	5	777931	Herb	Or
7	Charlie Krause	311 Panoramic BLVD	Colombia	1	332430	Charlie	Kr
8	Xavier Orsted	631 Park BLVD	United States	3	106841	Xavier	Or
9	Herb Frommer	279 Park BLVD	United States	3	308626	Herb	Fr
10	Xavier Neiman	242 Golden AVE	Brazil	0	254771	Xavier	Ne
11	Herb Neiman	227 Panoramic AVE	Thailand	5	687973	Herb	Ne
12	Xavier Frommer	907 Panoramic ST	Thailand	5	463449	Xavier	Fr
13	Larry Frommer	705 Panoramic AVE	Thailand	5	831314	Larry	Fr
14	Larry Krause	984 Green AVE	Colombia	1	201523	Larry	Kr
15	Oprah Orsted	368 Fake ST NE	Brazil	0	421158	Oprah	Or
16	Xavier Neiman	108 Main BLVD	Thailand	5	613916	Xavier	Ne
17	Herb Orsted	64 Golden BLVD	Colombia	1	268252	Herb	Or
18	Xavier Orsted	367 Golden AVE	Japan	4	863550	Xavier	Or
19	Herb Frommer	65 Green BLVD	Japan	4	264394	Herb	Fr
20	Herb Frommer	831 Golden BLVD	Thailand	5	358479	Herb	Fr

Step 1: Create a data source for FlexGrid

1. Add a new class file to the Xamarin.Forms application (Name: `Customer.cs`).
2. Add the following code to the `Customer.cs` file. We are using **Customer** class to represent data in the FlexGrid control.

Customer.cs

```
public class Customer
    {
        int _id, _countryID;
        string _first, _last, _address, _postalCode, _email;
        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|Colombia|Egypt|United
States|Japan|Thailand".Split('|');

        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('|');

        public Customer()
            : this(_rnd.Next(10000))
        {
        }
        public Customer(int id)
        {
            ID = id;
            First = GetString(_firstNames);
            Last = GetString(_lastNames);
            Address = GetRandomAddress();
            PostalCode = _rnd.Next(100000, 999999).ToString();
            CountryID = _rnd.Next() % _countries.Length;
            Email = string.Format("{0}@{1}.com", (First + Last.Substring(0, _rnd.Next(1,
Last.Length)).ToLower()), GetString(_emailServers));
        }
        public int ID
        {
            get { return _id; }
            set
            {
                if (value != _id)
                {
                    _id = value;
                }
            }
        }
        public string Name
        {
            get { return string.Format("{0} {1}", First, Last); }
        }

        public string Address
        {
            get { return _address; }
            set
            {
                if (value != _address)
                {
                    _address = value;
                }
            }
        }
        public string Country
        {
            get { return _countries[_countryID]; }
            set { _countries[_countryID] = value; }
        }

        public int CountryID
        {
            get { return _countryID; }
            set
            {
                if (value != _countryID)
```

```
            {
                _countryID = value;
            }
        }
    }
    public string PostalCode
    {
        get { return _postalCode; }
        set
        {
            if (_postalCode != value)
            {
                _postalCode = value;
            }
        }
    }
    public string First
    {
        get { return _first; }
        set
        {
            {
                _first = value;
            }
        }
    }

    public string Last
    {
        get { return _last; }
        set
        {
            if (_last != value)
            {
                _last = value;
            }
        }
    }
    public string Email
    {
        get { return _email; }
        set
        {
            {
                _email = value;
            }
        }
    }

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

Back to Top

Step 2: Add FlexGrid control

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

In XAML

1. Add a new content page (Name: `QuickStart.xaml`) to your .NET Standard or shared project and include references as shown below.

XAML

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:c1="clr-namespace:C1.Xamarin.Forms.Grid;assembly=C1.Xamarin.Forms.Grid"
             x:Class="CollectionViewXamarin.QuickStart">
</ContentPage>
```

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

XAML

```
<c1:FlexGrid x:Name="grid" ShowMarquee="True" SelectionMode="Cell"/>
```

3. Add the following references in the **QuickStart.xaml.cs** file.

C#

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

4. Add the following code in the QuickStart.xaml.cs file to create an instance of **C1CollectionView** class and set the **ItemSource** property of the FlexGrid control to CollectionView.

C#

```
namespace CollectionViewXamarin
```

```
{
    [XamlCompilation(XamlCompilationOptions.Compile)]
        public partial class QuickStart : ContentPage
        {
        public C1CollectionView<Customer> MyCollectionView { get; set; }
        public FlexGrid()
                {
            InitializeComponent();
            MyCollectionView = new C1CollectionView<Customer>(Customer.GetCustomerList(250));
            grid.ItemsSource = MyCollectionView;
            grid.RowHeaders.Columns.Clear();
            grid.AutoSizeColumns(0, grid.Columns.Count - 1);
            grid.AllowDragging = GridAllowDragging.None;
        }
    }
}
```

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 FlexGrid control.
    - **To return a Content 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.Chart.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.Chart.Platform.UWP.FlexGridRenderer.Init();
           ```

        4. (Optional) In case you compile your UWP application in **Release** mode, you need to explicitly add the following code to the **OnLaunched** method in your **App.xaml.cs** to include the correct assemblies within your application.

           C#

           ```
           var assembliesToInclude = new List<Assembly>();
           assembliesToInclude.Add(typeof(C1.Xamarin.Forms.Grid.Platform.UWP.FlexGridRenderer)
           .GetTypeInfo().Assembly);
           assembliesToInclude.Add(typeof(C1.UWP.Grid.FlexGrid).GetTypeInfo().Assembly);
           Xamarin.Forms.Forms.Init(e, assembliesToInclude);
           ```
4. Press **F5** to run the project.

Back to Top

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.

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 Content 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="App1.Grouping"
 xmlns:c1="clr-namespace:C1.Xamarin.Forms.Grid;assembly=C1.Xamarin.Forms.Grid">
  <StackLayout>
    <Grid VerticalOptions="FillAndExpand">
      <c1:FlexGrid x:Name="grid" AutoGenerateColumns="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 or on-demand loading is a powerful feature for mobile applications where data is loaded in chunks as the user scroll down a list in real time. Xamarin CollectionView supports incremental loading for data bound controls, such as FlexGrid and UITableView.

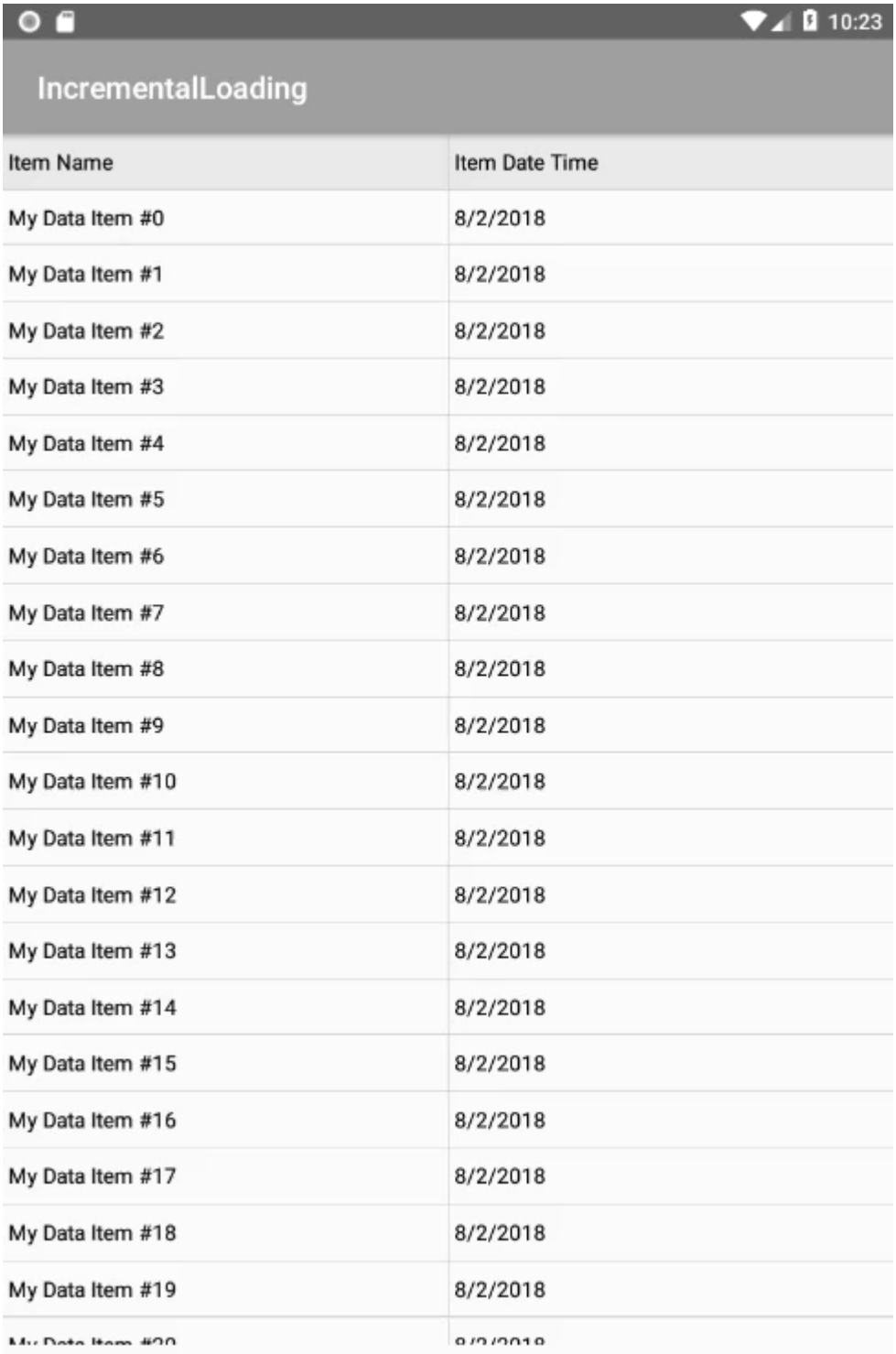
Adding incremental loading to ListView control is accomplished in two basic steps.

1. Implement your own collection view class, for example, OnDemandCollectionView, that extends C1CursorCollectionView class and overrides GetPageAsync. Once, this is completed you need to add logic that loads the data in pages or chunks.

2. Extend UITableView, FlexGrid, or another UI control to determine when the user has reached the bottom of a

page.

The following GIF image shows incremental loading in FlexGrid control.



The following code example demonstrates how to implement incremental loading using the CollectionView class.

1. Add a new Content 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

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamarinSample.IncrementalLoading"
             Title="Incremenatl Loading">
    <ListView x:Name="list">
        <ListView.ItemTemplate>
            <DataTemplate>
                <TextCell Text="{Binding ItemName}" Detail="{Binding
ItemDateTime,
           StringFormat='Created: {0}'}" />
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</ContentPage>
```

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 IncrementalLoading : ContentPage
        {
                public IncrementalLoading ()
                {
                        InitializeComponent ();

            var myCollectionView = new SimpleOnDemandCollectionView();
            list.ItemsSource = myCollectionView;

            list.LoadItemsOnDemand(myCollectionView);
        }
        }
    public class SimpleOnDemandCollectionView :
C1CursorCollectionView<MyDataItem>
    {
        const int MAX = 100;
        int current;
        public SimpleOnDemandCollectionView()
        {
            PageSize = 20;
        }

        public int PageSize { get; set; }
```

```
        protected override async Task<Tuple<string, IReadOnlyList<MyDataItem>>>
GetPageAsync(int startingIndex, string pageToken, int? count = null,
IReadOnlyList<SortDescription> sortDescriptions = null, FilterExpression
filterExpression = null, CancellationToken cancellationToken =
default(CancellationToken))
        {
            if (startingIndex >= MAX)
                return new Tuple<string, IReadOnlyList<MyDataItem>>(null, null);

            var newItems = new ObservableCollection<MyDataItem>();
            await Task.Run(() =>
            {
                for (int i = 0; i < this.PageSize; i++)
                {
                    if (current >= MAX)
                        break;
                    newItems.Add(new MyDataItem(startingIndex + i));
                    current++;
                }
            });
            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; }

    }
    public static class ListViewEx
    {
        public static void LoadItemsOnDemand<T>(this ListView listview,
         C1CursorCollectionView<T> collectionView) where T : class
        {
            listview.ItemAppearing += (s, e) =>
            {
                var index = collectionView.IndexOf((T)e.Item);
                if (index == collectionView.Count - 1)
                {
                    if (collectionView.HasMoreItems)
                    {
                        collectionView.LoadMoreItemsAsync();
                    }
```

```
            }
        };
        listview.Refreshing += async (s, e) =>
        {
            listview.IsRefreshing = true;
            await collectionView.RefreshAsync();
            listview.IsRefreshing = false;
        };
        if (collectionView.HasMoreItems)
        {
            collectionView.LoadMoreItemsAsync();
        }
    }
}
```

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	Country ID
	9	Charlie Neiman	Congo	1
	1	Herb Jammers	Egypt	2
	8	Larry Krause	Congo	1
	2	Oprah Orsted	Egypt	2
	5	Oprah Orsted	Egypt	2
	4	Steve Frommer	Egypt	2
	6	Steve Orsted	Congo	1
	7	Xavier Krause	Thailand	5
	0	Xavier Neiman	Congo	1
	3	Xavier Neiman	United States	3

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);
    C1CollectionView<Customer> cv = new C1CollectionView<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;
}
```

In XAML

XAML

```
<C1.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.
- **Animation**: Add effects to your chart with a variety of built in animations.
- **Scrolling & Zooming**: Enable scrolling and touch based zooming for your charts, for a better view of bigger charts on a smaller screen.
- **Touch Based Labels**: Display chart values using touch based labels.
- **Multiple Series**: Add multiple series on a single chart.

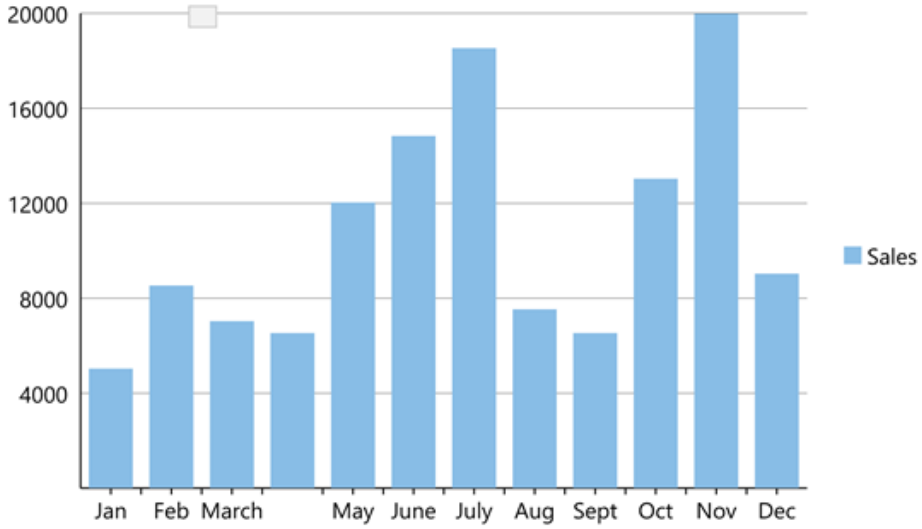
Quick Start: Add Data to FlexChart

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 C1.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 Content 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:c1="clr-namespace:C1.Xamarin.Forms.Chart;assembly=C1.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>
  <c1:FlexChart x:Name="chart"  ItemsSource="{Binding Data}" BindingX="Name" ChartType="Column"
   Grid.Row="1" Grid.ColumnSpan="2" VericalOptions="FillAndExpand">
    <c1:FlexChart.Series>
      <c1:ChartSeries x:Name="Sales2015" SeriesName ="Sales" Binding="Sales" ></c1:ChartSeries>
    </c1:FlexChart.Series>
  </c1: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 Content Page:** In the class constructor `App()`, set the Content 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.FlexChartRenderer.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.Chart.Platform.UWP.FlexChartRenderer.Init();
```

        4. (Optional) In case you compile your UWP application in **Release** mode, you need to explicitly add the following code to the **OnLaunched** method in your **App.xaml.cs** to include the correct assemblies within your application.

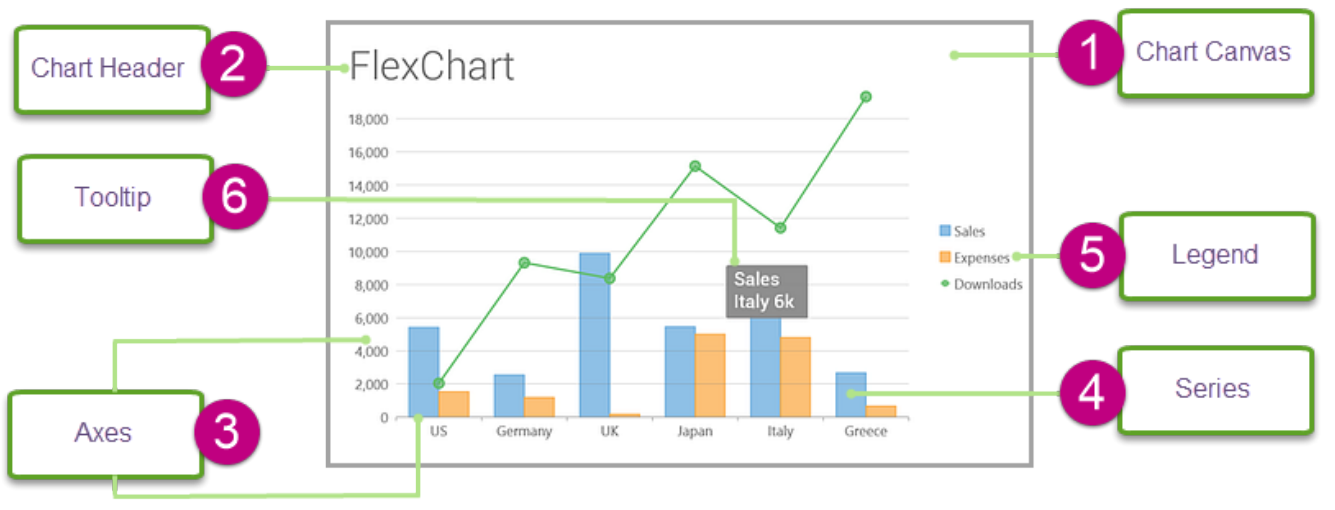
```
C#
var assembliesToInclude = new List<Assembly>();
assembliesToInclude.Add(typeof(C1.Xamarin.Forms.Chart.Platform.UWP.FlexChartRenderer)
.GetTypeInfo().Assembly);
assembliesToInclude.Add(typeof(C1.UWP.Chart.FlexChart).GetTypeInfo().Assembly);
Xamarin.Forms.Forms.Init(e, assembliesToInclude);
```

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

Back to Top

Chart Elements

FlexChart is composed of several elements as shown below:



1	**Chart Canvas**	Area on which all chart elements including the series, axes title and legend are placed.
2	**Chart Header**	Text that you want to display at the top of your chart, basically a title that serves as a heading for your chart.
3	**Axes**	Two primary axes, X and Y. Although in some cases you may add secondary axes as well.

4	**Series**	Collection of data that is plotted on the chart.
5	**Legend**	Name of the series added in the chart along with predefined symbols and colors used to plot data for that series.
6	**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

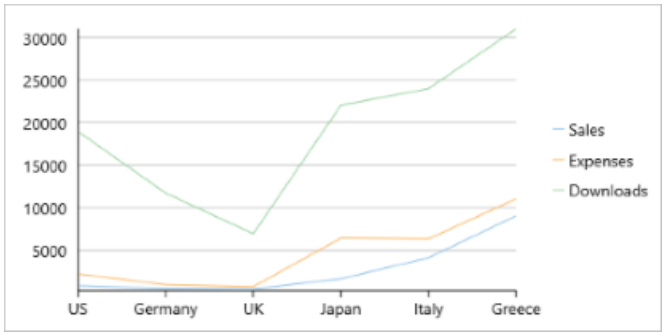
XAML

```
<c1:FlexChart x:Name="chart" ChartType="LineSymbols" ItemsSource="{Binding Data}" BindingX="Name">
  <c1:FlexChart.Series>
    <c1:ChartSeries x:Name="Sales2014" Binding="Sales" ></c1:ChartSeries>
  </c1:FlexChart.Series>
</c1:FlexChart>
```

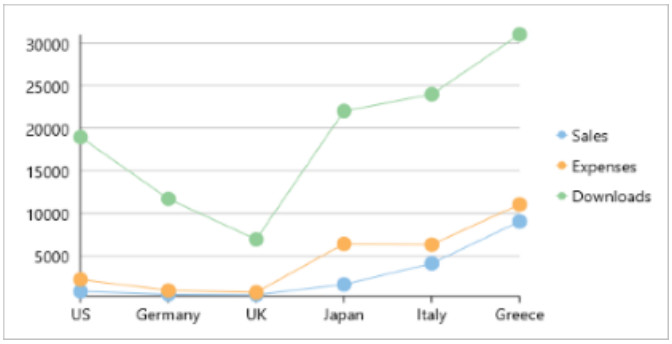
Line and LineSymbol chart

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

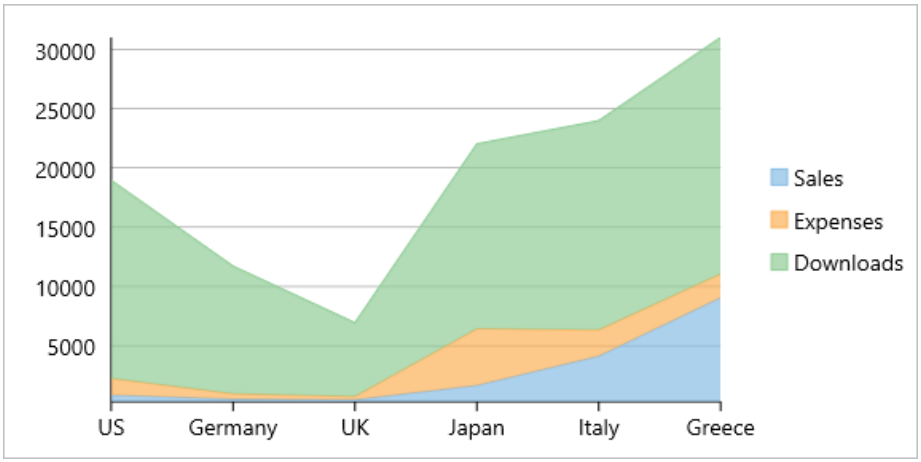


LineSymbol Chart

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.

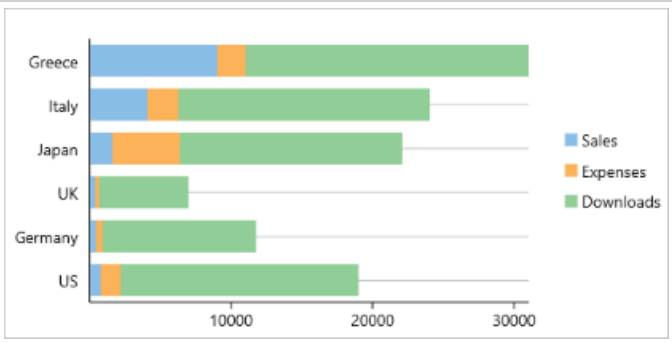
These charts are commonly used to show trends between associated attributes over time.



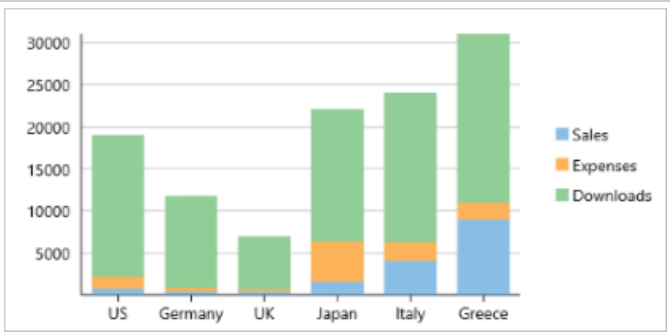
Bar and Column chart

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.



Bar Chart

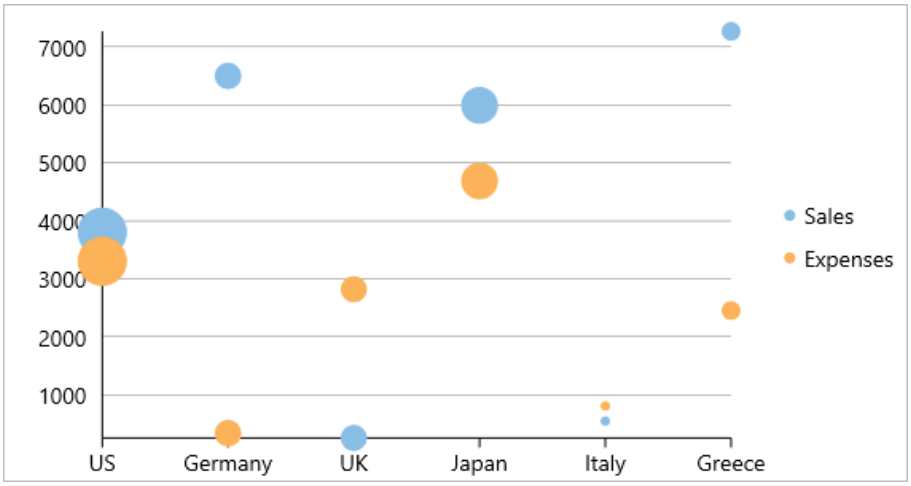


Column Chart

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.

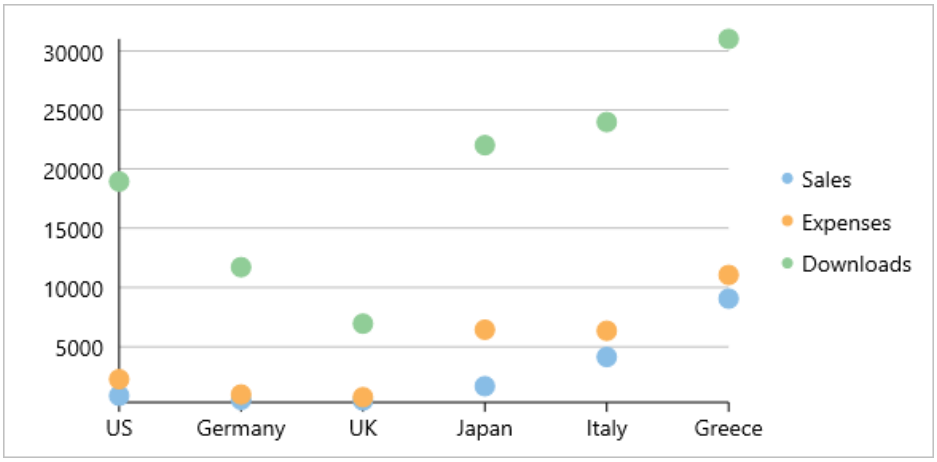
These charts are used to compare entities based on their relative positions on the axis as well as their size.



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

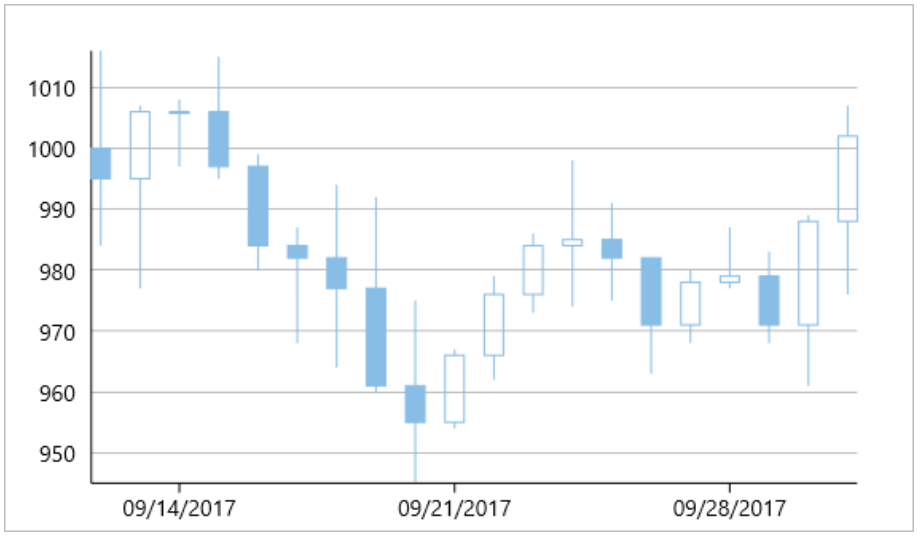
The Candlestick chart is made up of the following elements: **candle**, **wick**, and **tail**.

- **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.

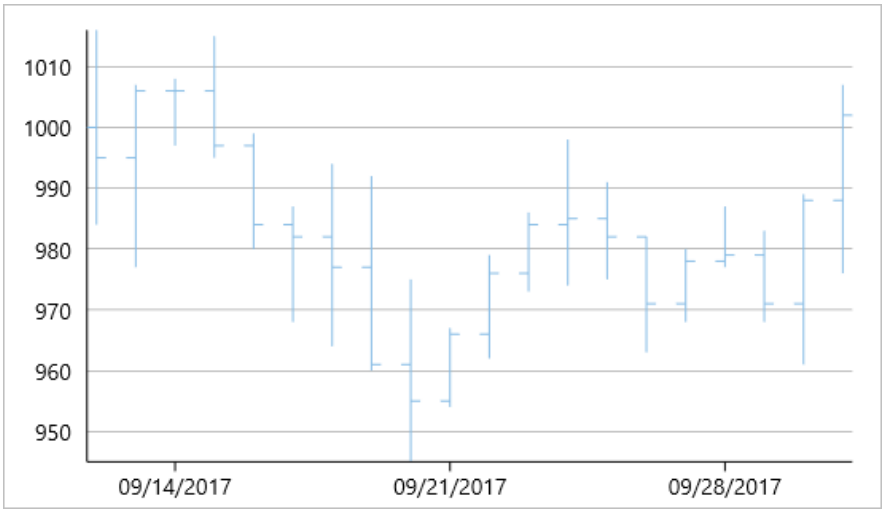
The following image shows a candlestick chart displaying stock prices.



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

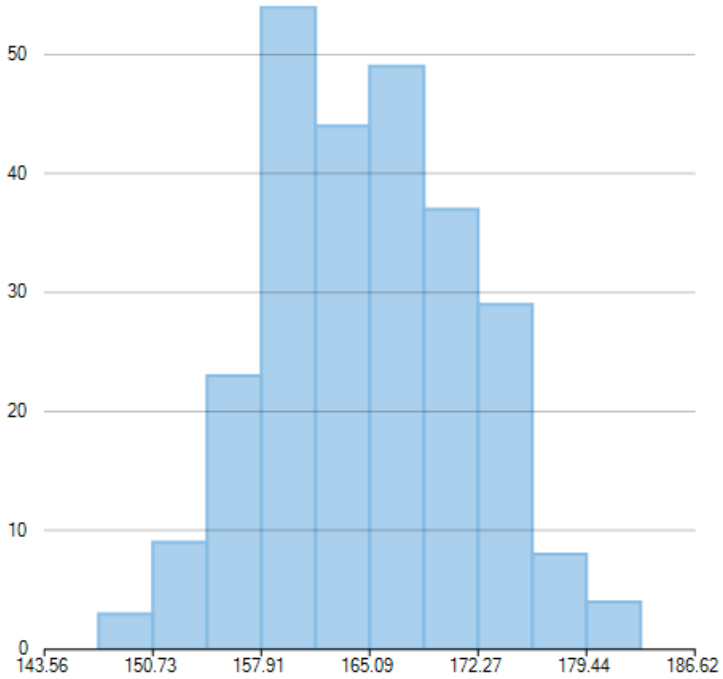
to showing the high and low value of a stock, the Y2 and Y3 array elements represent the stock's opening and closing price respectively.



Histogram Chart

Histogram chart plots the frequency distribution of data against the defined class intervals or bins. These bins are created by dividing the raw data values into a series of consecutive and non-overlapping intervals. Based on the number of values falling in a particular bin, frequencies are then plotted as rectangular columns against continuous x-axis.

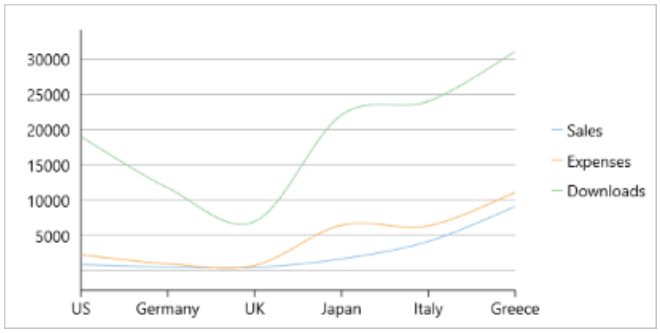
These charts are commonly used for visualizing distribution of numerical data over a continuous, or a certain period of time.



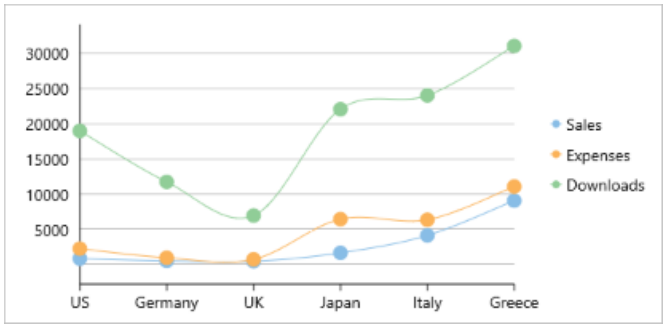
Spline and SplineSymbol chart

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.



Spline Chart



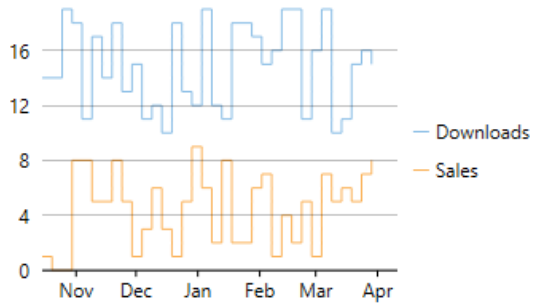
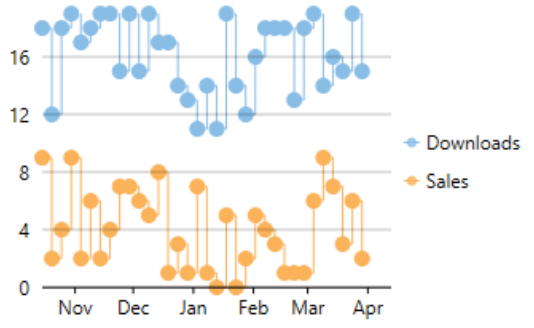
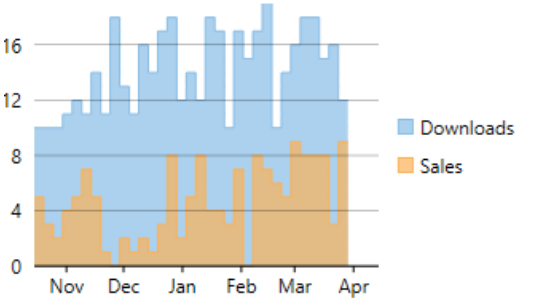
SplineSymbol Chart

Step Chart

Step charts use horizontal and vertical lines to present data that show sudden changes along y-axis by discrete amount. These charts help display changes that are sudden and irregular but stay constant till the next change. Step charts enable judging trends in data along with the duration for which the trend remained constant.

Consider a use case where you want to visualize and compare weekly sales and units downloaded of a software. As both of these values vary with discrete amounts, you can use step chart to visualize them. As shown in the image below, apart from depicting the change in sales these charts also show the exact time of change and the duration for which sales were constant. Moreover, you can easily identify the magnitude of respective changes by simply looking at the chart.

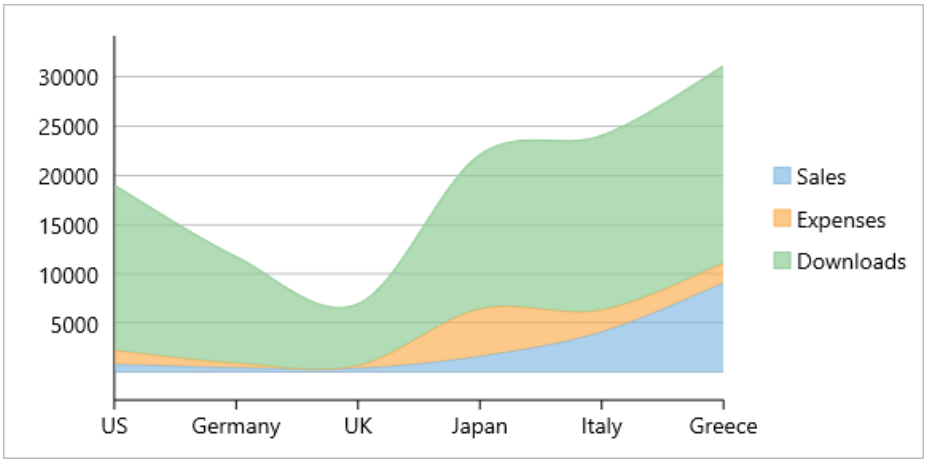
FlexChart supports Step chart, StepSymbols chart, and StepArea or filled step chart. The following table gives detailed explanation of these chart types.

 **Step Chart**	Step chart is similar to the Line chart, except that Line chart uses shortest distance to connect consecutive data points, while Step chart connects them with horizontal and vertical lines. These horizontal and vertical lines give the chart step-like appearance. While the line charts depict change and its trend, the Step charts also help in judging the magnitude and the intermittent pattern of the change.
 **StepSymbols Chart**	StepSymbols chart combines the Step chart and the Scatter chart. FlexChart plots data points by using symbols and connects those data points with horizontal and vertical step lines. Here, the data points are marked using symbols and, therefore, help mark the beginning of an intermittent change.
	StepArea chart combines the Step chart and the Area chart. It is similar to Area chart with the difference in the manner in which data points are connected. FlexChart plots the data points using horizontal and vertical step lines, and then fills the area between x-axis and the step lines. These are based on Step charts, and are commonly used to compare discrete and intermittent changes between two or more quantities. This gives the chart stacked appearance, where related data points of the multiple series seem stacked above the other. For example, number of units downloaded and sales of a software for a particular time duration can be easily compared as shown in the image.

StepArea Chart	

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.



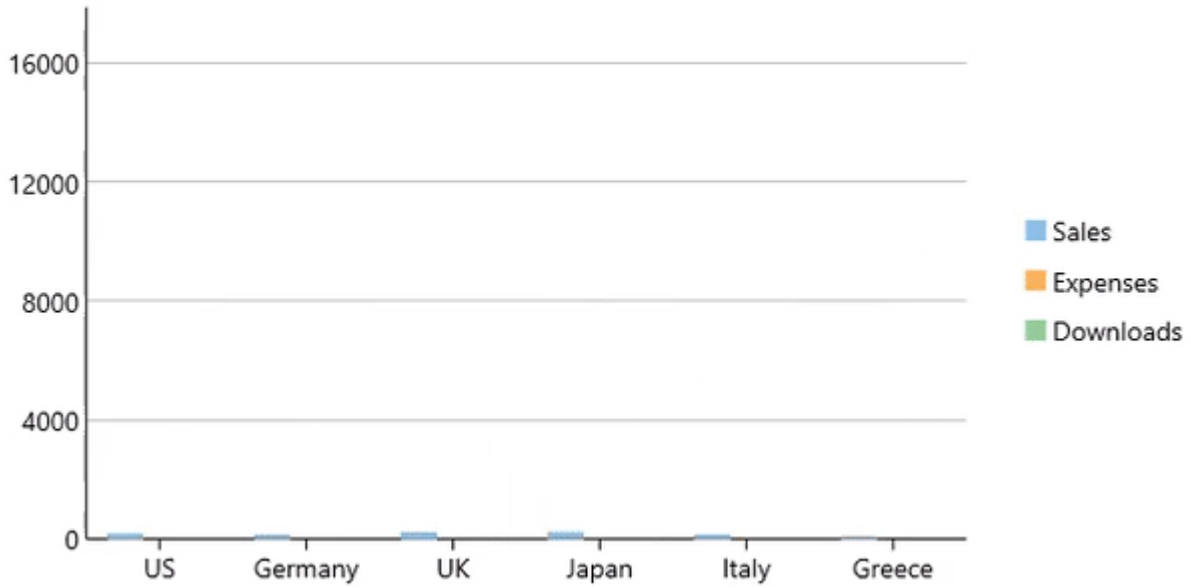
SplineArea Chart

Features

Animation

FlexChart allows you to enable animation effects using one of the two ways, either on loading when the chart is drawn or on updating when the chart is redrawn after modifications. It supports animation in charts through C1Animation class available in the C1.Xamarin.Forms.Core namespace.

The following GIF shows animation in FlexChart.



You can also set the duration of animation in chart using Duration property of the C1Animation class and interpolate values of animation using Easing property of the C1Animation class, which accepts values from the C1Easing class. This class supports a collection of standard easing functions such as CircleIn, CircleOut, and Linear.

- **CircleIn:** Easing function that starts slow and speeds up in the form of a circle.
- **CircleOut:** Easing function that starts fast and slows down in the form of a circle.
- **Linear:** Easing function with constant speed.

C#

```
C1Animation animate = new C1Animation();
// set update animation duration
animate.Duration = new TimeSpan(3000 * 10000);
// interpolate the values of animation
animate.Easing = C1Easing.Linear;
```

In addition to easing functions of C1Easing class, FlexChart supports built in easing functions of Xamarin.Forms.Easing class. For more information, refer Xamarin Easing Class.

You can show animation while loading or updating a chart. To show animation while loading the chart, use LoadAnimation property of the ChartBase class, which gets the load animation from the object of C1Animation class. Similarly, to animate the chart when underlying data collection changes on adding, removing, or modifying a value, you can use UpdateAnimation property of the ChartBase class.

C#

```
// set the loading animation
chart.LoadAnimation = animate;
```

You can apply the animation effect by setting the AnimationMode property which accepts values from AnimationMode enumeration. This enumeration supports four different animation modes: All, None, Series, and Point.

- **All**: All plot elements animate at once from the bottom of the plot area.
- **None**: Does not display any animation.
- **Series**: Each series animates one at a time from the bottom of the plot area.
- **Point**: The plot elements appear one at a time from left to right.

C#

```
// set the animation mode
chart.AnimationMode = AnimationMode.Series;
```

Annotations

Annotations are used to mark important news or events that can be attached to a specific data point on FlexChart. Annotations can also be used to place arbitrary elements such as images, shapes and text onto the chart. The FlexChart control supports various built-in annotations such as Polygon, Line, Ellipsis, Rectangle, Image and Text.

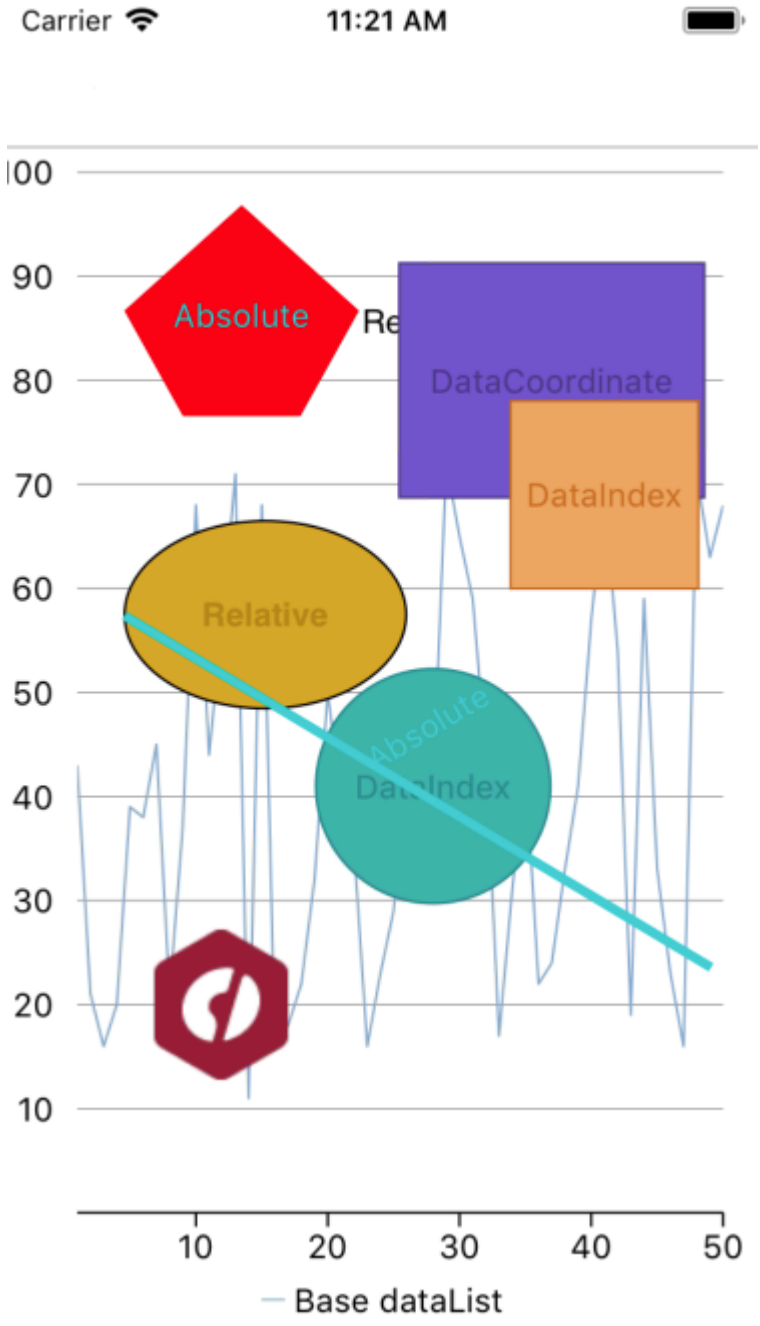
You can specify the position of an annotation on FlexChart by setting the **Position** property to Bottom, Center, Left, Right or Top. To specify the attachment of annotation on FlexChart, you can use the **Content** property and set its value to:

- **Absolute**: The coordinates of the annotation are specified by the annotation's shape data in pixels.
- **DataCoordinate**: The coordinates of the annotation are specified in data coordinates.
- **DataIndex**: The coordinates of the annotation are specified by the data series index and the data point index.
- **Relative**: The coordinates of the annotation are specified as a relative position within the control, where (0, 0) is the top left corner and (1, 1) is the bottom right corner.

This topic demonstrates how to set various types of annotations on the FlexChart control in three steps:

- **Step1: Create a Data Source for FlexChart**
- **Step 2: Initialize a FlexChart control and add Annotations to it in XAML**
- **Step 3: Set Bindings in Code**

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



Step 1: Create a Data Source for FlexChart

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

C#

```
class ChartSampleData
    {
        public class AnnotationViewModel
        {
            List<DataItem> _data;
            List<DataItem> _simpleData;
            Random rnd = new Random();

            public List<DataItem> Data
            {
                get
                {
                    if (_data == null)
                    {
                        _data = new List<DataItem>();
                        for (int i = 1; i < 51; i++)
                        {
                            _data.Add(new DataItem()
                            {
                                X = i,
                                Y = rnd.Next(10, 80)
                            });
                        }
                    }

                    return _data;
                }
            }

            public List<DataItem> SimpleData
            {
                get
                {
                    if (_simpleData == null)
                    {
                        _simpleData = new List<DataItem>();
                        _simpleData.Add(new DataItem() { X = 1, Y = 30 });
                        _simpleData.Add(new DataItem() { X = 2, Y = 20 });
                        _simpleData.Add(new DataItem() { X = 3, Y = 30 });
                        _simpleData.Add(new DataItem() { X = 4, Y = 65 });
                        _simpleData.Add(new DataItem() { X = 5, Y = 70 });
                        _simpleData.Add(new DataItem() { X = 6, Y = 60 });
                    }

                    return _simpleData;
                }
            }
        }

        public class DataItem
```

```
        {
            public int X { get; set; }
            public int Y { get; set; }
        }
    }
```

Step 2: Initialize a FlexChart control and add Annotations to it in XAML

1. Right-click your project in the **Solution Explorer** and select **Add | New Item**. The **Add New Item** dialog appears.
2. Select **Content Page** from the installed templates and provide a name to it, for example, Annotations.
3. Click **Add** to add the page to your project.
4. Initialize the FlexChart control in the Annotation.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:Chart="clr-
namespace:C1.Xamarin.Forms.Chart;assembly=C1.Xamarin.Forms.Chart"
              xmlns:Annotation="clr-
namespace:C1.Xamarin.Forms.Chart.Annotation;assembly=C1.Xamarin.Forms.Chart"
             x:Class="Annotations.App">
    <Chart:FlexChart x:Name="flexChart" BindingX="Name">
        <Chart:FlexChart.AxisY>
            <Chart:ChartAxis Min="0" Max="100" MajorUnit="10" AxisLine="False"
MajorGrid="True" MajorTickMarks="None"/>
        </Chart:FlexChart.AxisY>
        <Chart:FlexChart.Series>
            <Chart:ChartSeries SeriesName="Base dataList" Binding="Y"
BindingX="X"/>
        </Chart:FlexChart.Series>

        <Chart:FlexChart.Layers>
            <Annotation:AnnotationLayer>
                <Annotation:AnnotationLayer.Annotations>
                    <Annotation:Text Content="Relative" Location="0.55, 0.15"
Attachment="Relative" >
                        <Annotation:Text.AnnotationStyle>
                            <Chart:ChartStyle FontSize="14" Stroke="Black"
FontFamily="GenericSansSerif" />
                        </Annotation:Text.AnnotationStyle>
                    </Annotation:Text>

                    <Annotation:Ellipse Content="Relative" Location="0.4, 0.45"
Width="120" Height="80" Attachment="Relative">
                        <Annotation:Ellipse.AnnotationStyle>
                            <Chart:ChartStyle Fill="Goldenrod"/>
                        </Annotation:Ellipse.AnnotationStyle>
                        <Annotation:Ellipse.ContentStyle>
                            <Chart:ChartStyle  Stroke="DarkGoldenrod"
FontAttributes="Bold" FontSize="14" FontFamily="GenericSansSerif" />
                        </Annotation:Ellipse.ContentStyle>
```

```
                    </Annotation:Ellipse>

                    <Annotation:Circle Content="DataIndex" Radius="50"
SeriesIndex="0" PointIndex="27" Attachment="DataIndex" >
                        <Annotation:Circle.AnnotationStyle>
                            <Chart:ChartStyle Fill="LightSeaGreen" FontSize="14"
Stroke="DarkCyan" FontFamily="GenericSansSerif" FontAttributes="Bold" />
                        </Annotation:Circle.AnnotationStyle>
                    </Annotation:Circle>

                    <Annotation:Rectangle Content="DataCoordinate" Width="130"
Height="100" Location="37, 80" Attachment="DataCoordinate">
                        <Annotation:Rectangle.AnnotationStyle>
                            <Chart:ChartStyle Fill="SlateBlue" FontSize="14"
Stroke="DarkSlateBlue" FontFamily="GenericSansSerif" FontAttributes="Bold" />
                        </Annotation:Rectangle.AnnotationStyle>
                    </Annotation:Rectangle>

                    <Annotation:Square Content="DataIndex" Length="80"
SeriesIndex="0" PointIndex="40" Attachment="DataIndex" >
                        <Annotation:Square.AnnotationStyle>
                            <Chart:ChartStyle Fill="SandyBrown" FontSize="14"
Stroke="Chocolate" FontFamily="GenericSansSerif" FontAttributes="Bold" />
                        </Annotation:Square.AnnotationStyle>
                    </Annotation:Square>

                    <Annotation:Polygon x:Name="polygonAnno" Content="Absolute"
Attachment="Absolute">
                        <Annotation:Polygon.AnnotationStyle>
                            <Chart:ChartStyle Fill="Red" StrokeThickness="3"
FontSize="14" Stroke="DarkTurquoise" FontFamily="GenericSansSerif"
FontAttributes="Bold" />
                        </Annotation:Polygon.AnnotationStyle>
                    </Annotation:Polygon>

                    <Annotation:Line Content="Absolute" Start="50, 200"
End="300, 350" Attachment="Absolute" >
                        <Annotation:Line.AnnotationStyle>
                            <Chart:ChartStyle StrokeThickness="4" FontSize="14"
Stroke="DarkTurquoise" FontFamily="GenericSansSerif" FontAttributes="Bold" />
                        </Annotation:Line.AnnotationStyle>
                    </Annotation:Line>

                    <Annotation:Image Location="12, 20" x:Name="imageAnno"
Width="64" Height="64" Attachment="DataCoordinate" />

                </Annotation:AnnotationLayer.Annotations>
            </Annotation:AnnotationLayer>
        </Chart:FlexChart.Layers>

    </Chart:FlexChart>
```

```
</ContentPage>
```

Step 3: Set Bindings in Code

1. In the **Solution Explorer**, expand the Annotations.xaml node and open the Annotations.xaml.cs to open the C# code behind.
2. In the Annotations() class constructor, set the BindingContext for the FlexChart to SampleViewModel and other annotation bindings as illustrated in the code below.

C#

```
public partial class AnnotationSample : ContentPage
    {
        public AnnotationSample()
        {
            InitializeComponent();
            Title = AppResources.GettingStartedTitle;
            flexChart.ItemsSource = new AnnotationViewModel().Data;
            flexChart.ChartType = ChartType.Line;
                        flexChart.BindingX = "X";
                        flexChart.LegendPosition = ChartPositionType.Bottom;
                        var pngImage = ImageSource.FromResource("
<ApplicationName>.Images.butterfly.png");
            imageAnno.Source = pngImage;
            polygonAnno.Points = CreatePoints();
        }
        private System.Collections.ObjectModel.ObservableCollection<Point>
CreatePoints()
        {
            System.Collections.ObjectModel.ObservableCollection<Point> points =
new System.Collections.ObjectModel.ObservableCollection<Point>();
            points.Add(new Point(100, 25));
            points.Add(new Point(50, 70));
            points.Add(new Point(75, 115));
            points.Add(new Point(125, 115));
            points.Add(new Point(150, 70));

            return points;
        }
    }
```

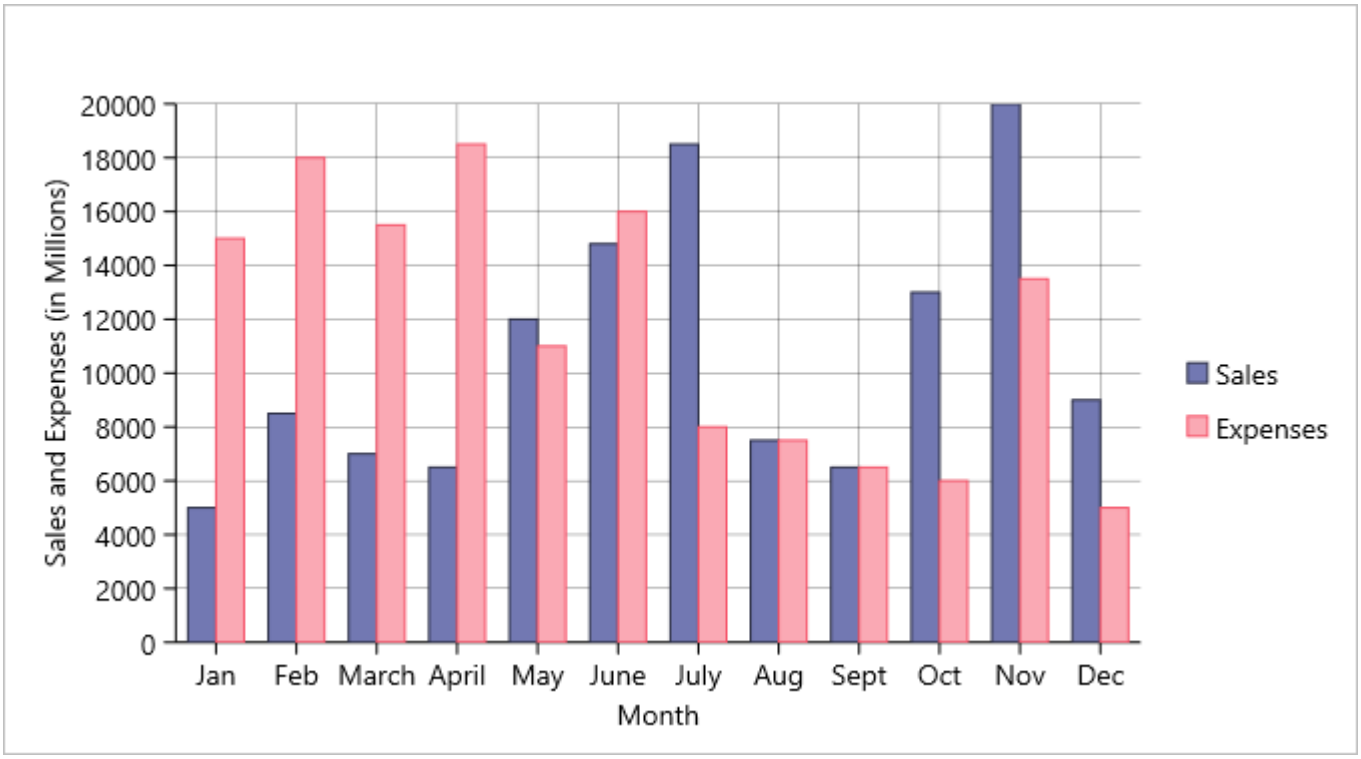
Axes

An axis is composed of several elements, such as formats, lines, axis line, origin, and titles. There are several properties available in FlexChart that let you customize these elements, for both X and Y axes.

See https://msdn.microsoft.com/en-us/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

```
<c1:FlexChart.AxisY>
  <c1:ChartAxis MajorGrid="true" AxisLine="true" Title="Sales and Expenses (in
Millions)"
      MajorUnit="2000" Format="D"></c1:ChartAxis>
</c1:FlexChart.AxisY>
<c1:FlexChart.AxisX>
```

```
  <c1:ChartAxis AxisLine="true" Title="Month" MajorGrid="true"></c1:ChartAxis>
</c1:FlexChart.AxisX>
```

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.

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.

In Code

C#

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

In XAML

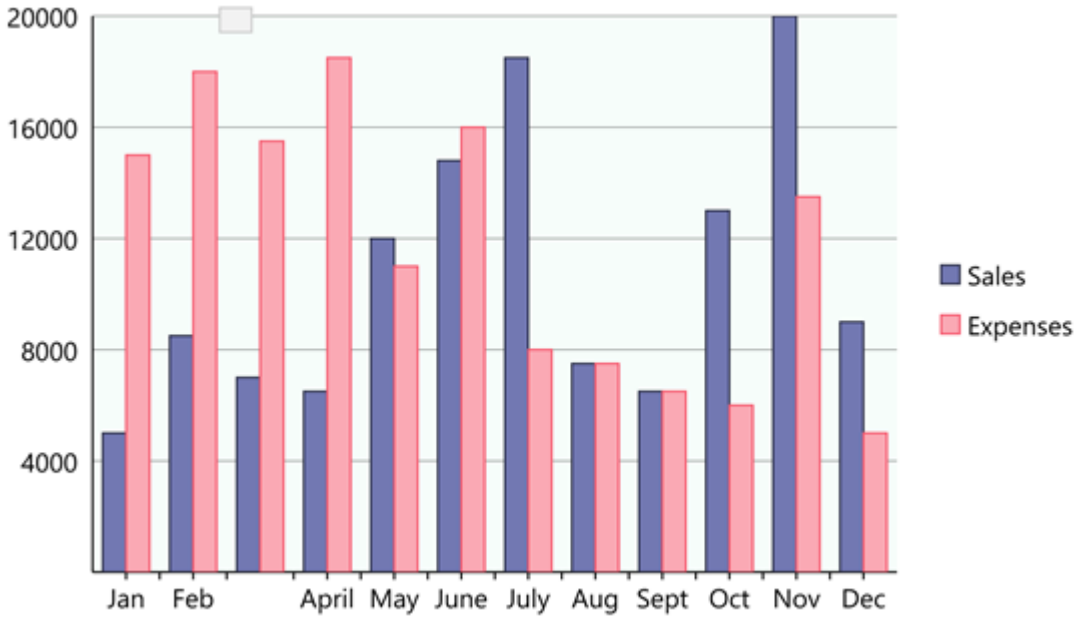
XAML

```
<c1:FlexChart.AxisX>
  <c1:ChartAxis AxisLine="true" Title="Month" MajorGrid="true" Origin="12000">
</c1:ChartAxis>
</c1:FlexChart.AxisX>
```

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.



The following code examples demonstrate 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.

In Code

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;
```

In XAML

XAML

```
<c1:FlexChart x:Name="chart" ItemsSource="{Binding Data}" BindingX="Name"
ChartType="Column"
    Grid.Row="1" Grid.ColumnSpan="2" VerticalOptions="FillAndExpand">
  <c1:FlexChart.Series>
    <c1:ChartSeries x:Name="Sales2015" SeriesName="Sales" Binding="Sales">
      <c1:ChartSeries.Style>
        <c1:ChartStyle Fill="#7278B2" Stroke="#2D3047" />
      </c1:ChartSeries.Style>
    </c1:ChartSeries>
    <c1:ChartSeries x:Name="Expenses2015" SeriesName="Expenses" Binding="Expenses">
      <c1:ChartSeries.Style>
        <c1:ChartStyle Fill="#FAA9B4" Stroke="#F6546A" />
      </c1:ChartSeries.Style>
    </c1:ChartSeries>
  </c1:FlexChart.Series>
  <c1:FlexChart.PlotStyle>
        <c1:ChartStyle Fill="#F6FDFA" StrokeThickness="0"></c1:ChartStyle>
  </c1:FlexChart.PlotStyle>
</c1:FlexChart>
```

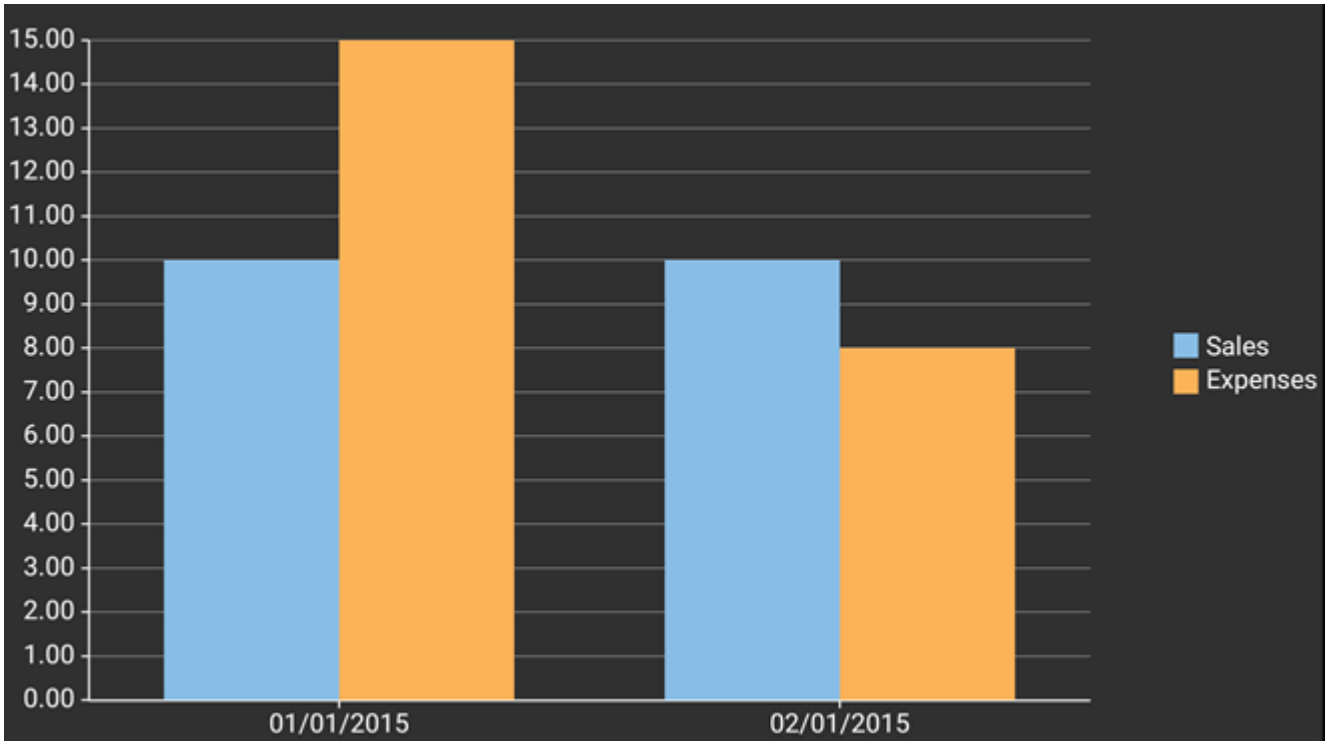
Data Binding

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
Binding	Property for binding Y-axis values
BindingX	Property for binding X-axis values
ItemsSource	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

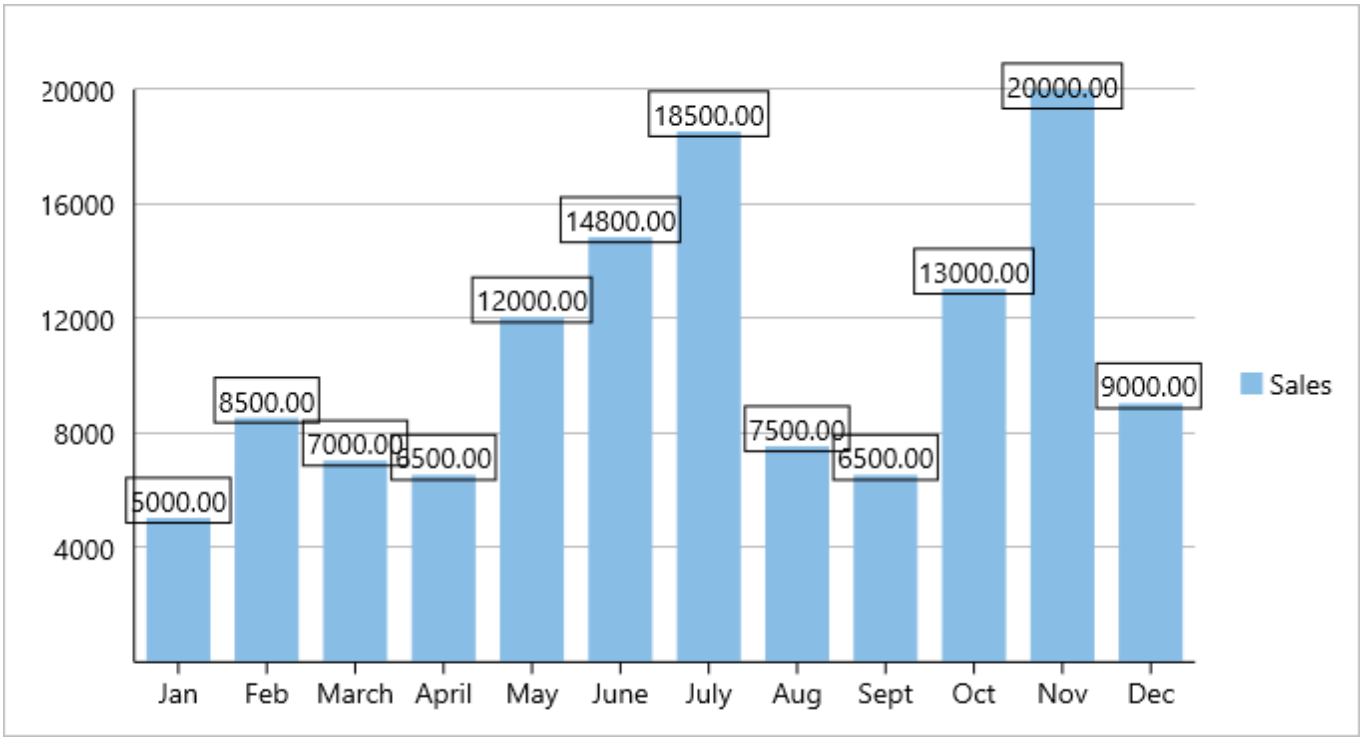
```
<c1:FlexChart x:Name="chart" ItemsSource="{Binding FlexChartDataSource}"
BindingX="Time">
  <c1:FlexChart.Series>
    <c1:ChartSeries Binding="Sales" Name="Sales" />
    <c1:ChartSeries Binding="Expenses" Name="Expenses" />
  </c1:FlexChart.Series>
</c1:FlexChart>
```

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

The following code example demonstrates how to use data labels in FlexChart and set their position. This examples uses the sample created in the Quick Start section.

XAML

```
<c1:FlexChart.DataLabel>
      <c1:ChartDataLabel Position="Top" Border="True" Content="{}{value:F2}"/>
</c1:FlexChart.DataLabel>
```

In Code

To add the content and change the label position in code, use the following code.

C#

```
//Setting the data labels
chart.DataLabel.Content = "{}{value:F2}";
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. You can also customize the appearance of the data labels.

In XAML

XAML

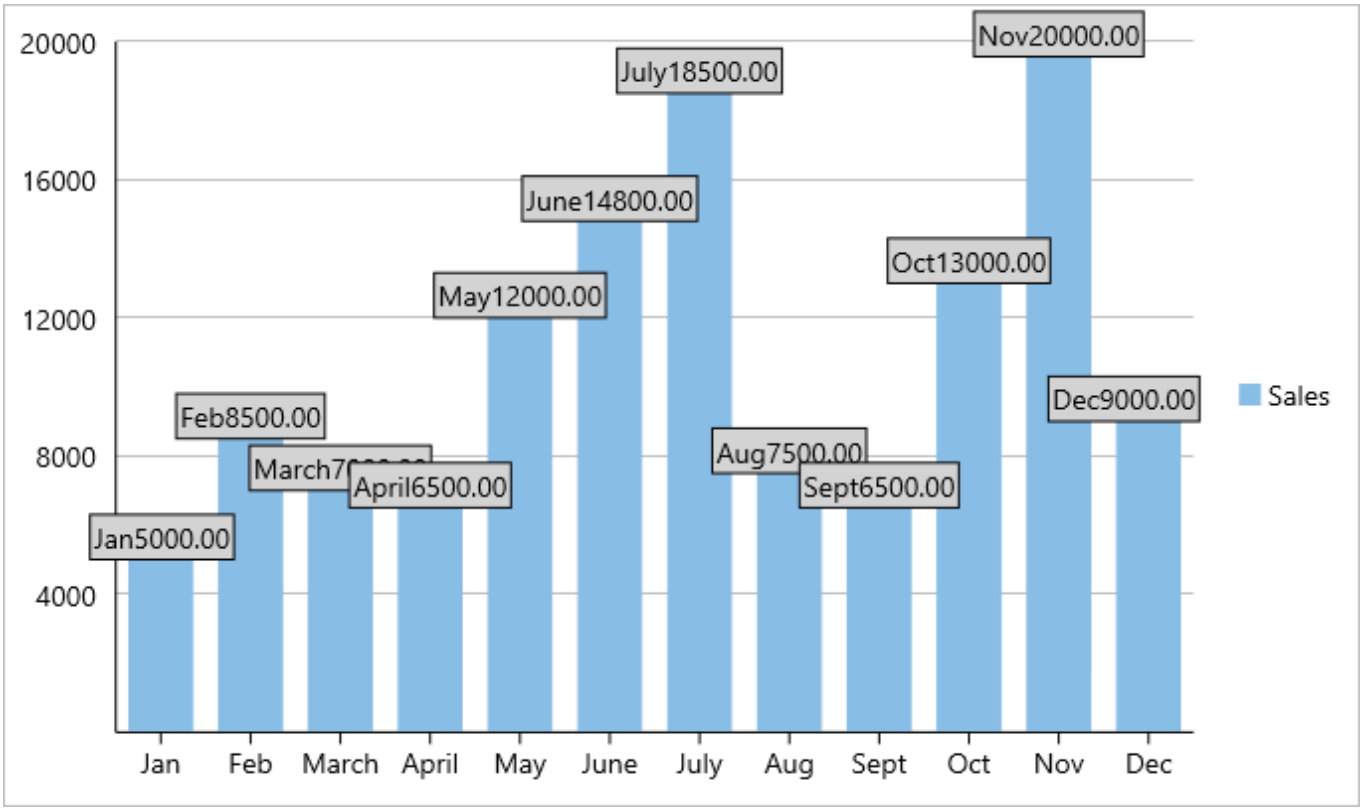
```
<c1:FlexChart.DataLabel>
       <c1:ChartDataLabel Border="True" Position="Top" Content="{}{x}{value:F2}">
              <c1:ChartDataLabel.Style>
                    <c1:ChartStyle FontSize="14" Fill="LightGray"></c1:ChartStyle>
               </c1:ChartDataLabel.Style>
            </c1:ChartDataLabel>
         </c1:FlexChart.DataLabel>
```

In Code

C#

```
chart.DataLabel.Content = "{}{x}{value:F2}";
chart.DataLabel.Position = ChartLabelPosition.Top;
chart.DataLabel.Style.Fill = Color.LightGray;
chart.DataLabel.Style.FontSize = 14;
```

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



Manage Overlapped Data Labels

A common issue pertaining to charts is the overlapping of data labels that represent data points. In most cases, overlapping occurs due to long text in data labels or large numbers of data points.

In case of overlapped data labels in FlexChart, it provides the following ways to manage the overlapping.

- **Auto Arrangement of Data Labels**
- **Hide Overlapped Labels**
- **Control Appearance of Overlapped Labels**
- **Rotate Data Labels**
- **Trim or Wrap Data Labels**

Auto Arrangement of Data Labels

The easiest way to handle overlapping of data labels is to set the FlexChart to position the data labels automatically. For automatic positioning of data labels, you can set the Position property to Auto. Moreover, you can also set the MaxAutoLabels property to set the maximum number of labels that can be positioned automatically.

When the Position property is set to Auto, the number of created data labels is limited by MaxAutoLabels property which is 100 by default. You can increase the value of MaxAutoLabels property if necessary, but it may slow down the chart rendering since the label positioning algorithm becomes expensive in terms of performance when number of labels is large.

This approach may not provide an optimal layout when working with large data set and when there is no enough space for all data labels. In this case, it's recommended to reduce the number of data labels. For example, create a series with limited number of data points that should have labels, that is, chose to hide the labels at the individual series level.

In Code

C#

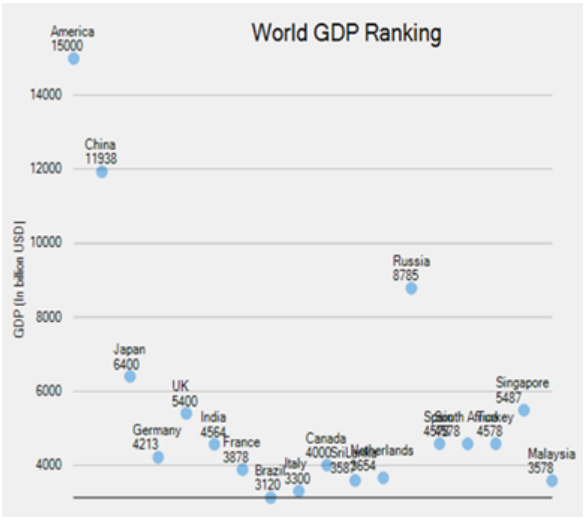
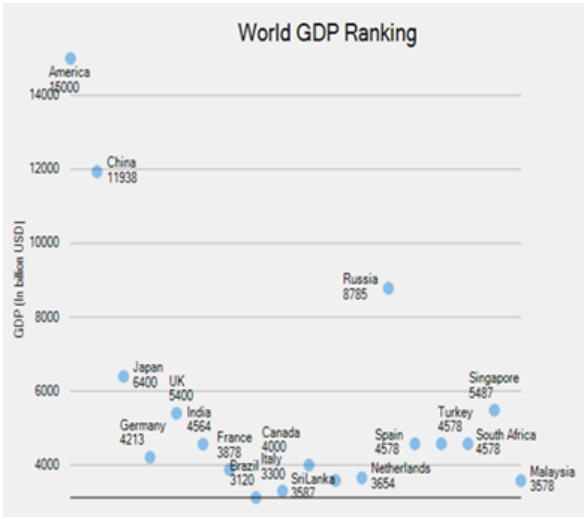
```
// Set Position and MaxAutoLabels property
flexChart1.DataLabel.Position = LabelPosition.Auto;
flexChart1.DataLabel.MaxAutoLabels = 150;
```

In XAML

XAML

```
<c1:ChartDataLabel Content="{}{type}" MaxAutoLabels="150" Position="Auto">
</c1:ChartDataLabel>
```

The image below shows how FlexChart appears after setting the Position and MaxAutoLabels property.

LabelPosition.Top	LabelPostion.Auto
	

Hide Overlapped Labels

In case of overlapped data labels in FlexChart, you can use the Overlapping property provided by the DataLabel class. This approach is helpful when developer wants to completely hide or show the overlapped data labels.

In Code

C#

```
// Set Overlapping property
flexChart1.DataLabel.Overlapping = LabelOverlapping.Hide;
```

In XAML

XAML

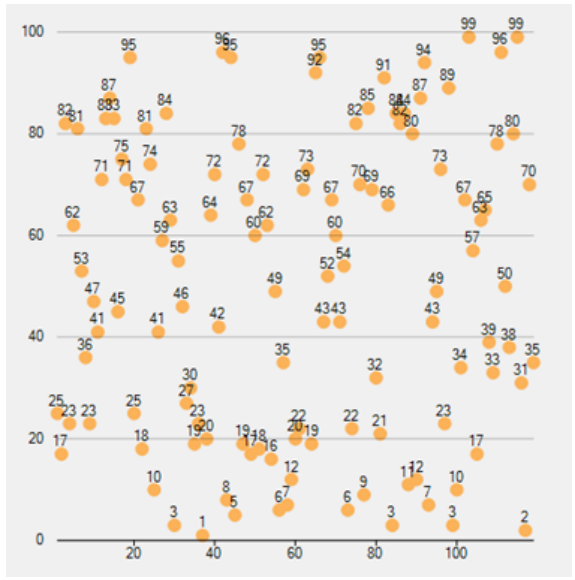
```
<c1:ChartDataLabel Content="{}{value}" Overlapping="Hide" Position="Top"></c1:ChartDataLabel>
```

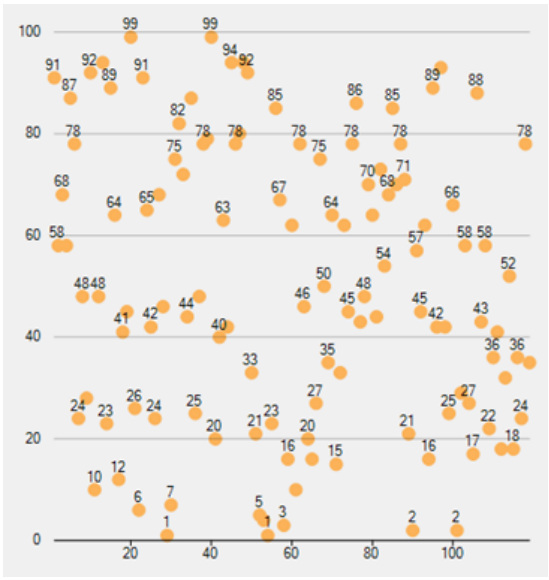
The Overlapping property accepts the following values in the LabelOverlapping enumeration.

Enumeration	Description
Hide	Hide overlapped data labels.
Show	Show overlapped data labels.

The image below shows how FlexChart appears after setting the Overlapping property.

LabelOverlapping.Show	LabelOverlapping.Hide





Control Appearance of Overlapped Labels

Furthermore, you can use the OverlappingOptions property to specify additional label overlapping options that will help the user to effectively manage overlapping of data labels.

In Code

C#

```
// Set OverlappingOptions property
flexChart1.DataLabel.OverlappingOptions = LabelOverlappingOptions.OutsidePlotArea;
```

In XAML

XAML

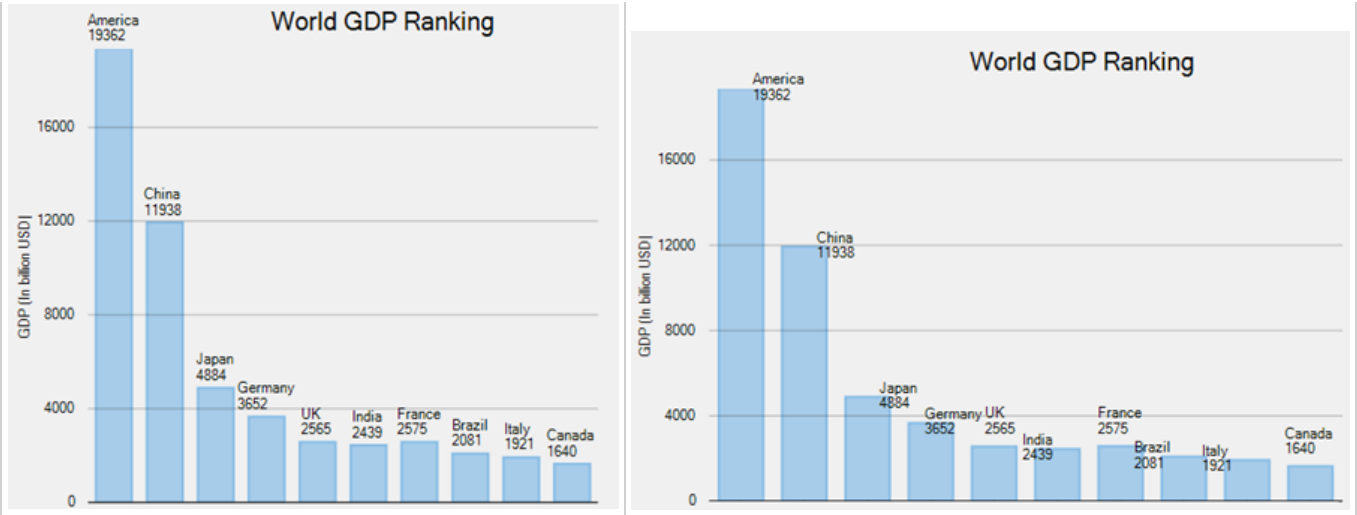
```
<c1:ChartDataLabel Content="{}{type}" OverlappingOptions="OutsidePlotArea" Position="Top">
</c1:ChartDataLabel>
```

The OverlappingOptions property accepts the following values in the LabelOverlappingOptions enumeration.

Enumeration	Description
None	No overlapping is allowed.
OutsidePlotArea	Allow labels outside plot area.
OverlapDataPoints	Allow overlapping with data points.

The image below shows how FlexChart appears after setting the OverlappingOptions property.

LabelOverlappingOptions.None	**LabelOverlappingOptions.OutsidePlotArea**



Rotate Data Labels

Another option to manage overlapping of data labels in FlexChart is to use the Angle property. The Angle property enables the user to set a specific rotation angle for data labels.

In Code

C#

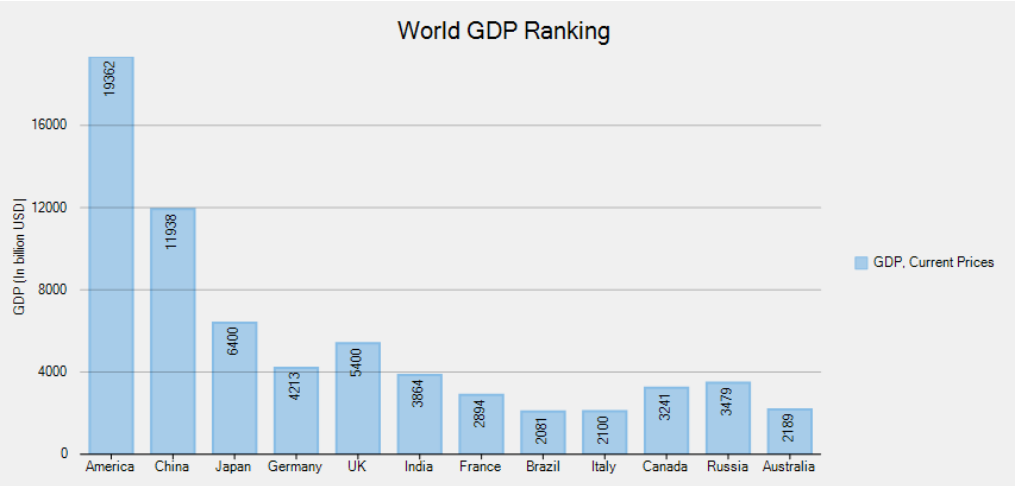
```
// Set the Angle property
flexChart1.DataLabel.Angle = 90;
```

In XAML

XAML

```
<c1:ChartDataLabel Content="{}{type}" Angle="90" Position="Top">
</c1:ChartDataLabel>
```

The image below shows how FlexChart appears after setting the Angle property.



Trim or Wrap Data Labels

To manage the content displayed in the data labels, in case of overlapping, you can either trim the data labels or wrap the data labels using ContentOptions property. Managing of data labels using the ContenOptions property is dependent on MaxWidth and MaxLines property.

The MaxWidth property allows you to set the maximum width of a data label. In case the width of data label text exceeds the specified width, then you can either trim the data labels or wrap the data labels using the ContentOptions property.

The MaxLines property allows you to set the maximum number of lines in data label. This property helps you to limit the wrapped text to grow vertically. In case the wrapped text does not fit within the specified MaxWidth and MaxLines property values, then the last line gets trimmed with an ellipses(...).

In Code

C#

```
// Set MaxWidth property
flexChart1.DataLabel.MaxWidth = 25;
// Set ContentOptions property
flexChart1.DataLabel.ContentOptions = ContentOptions.Trim;
```

In XAML

XAML

```
<c1:ChartDataLabel Content="{}{type}" MaxWidth="25" ContentOptions="Trim" Position="Top">
</c1:ChartDataLabel>
```

The image below shows how FlexChart appears after setting the ContentOptions property.

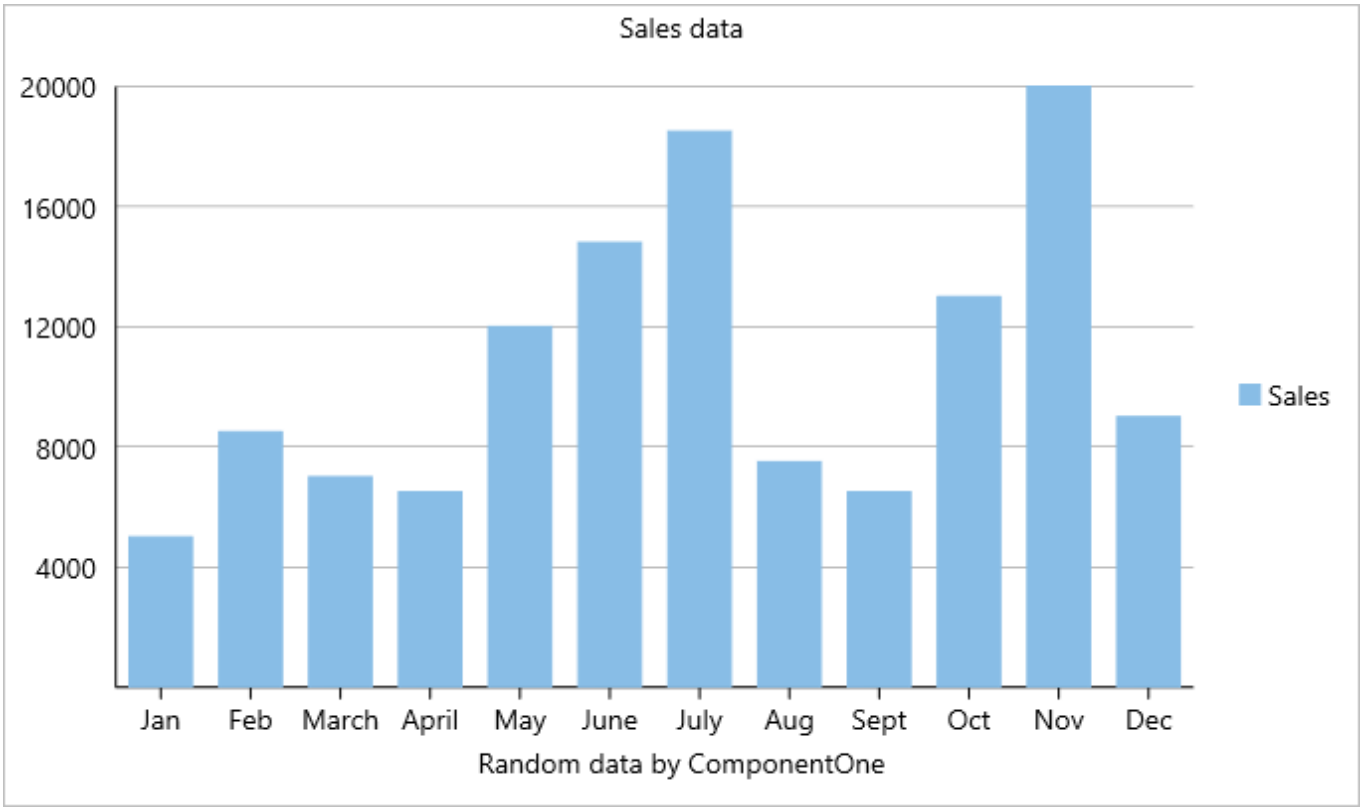
ContentOptions.None	ContentOptions.Trim
	

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.



The following code examples demonstrate how to set header and footer properties in C# and XAML. This example uses the sample created in the Quick Start section.

In Code

C#

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

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

In XAML

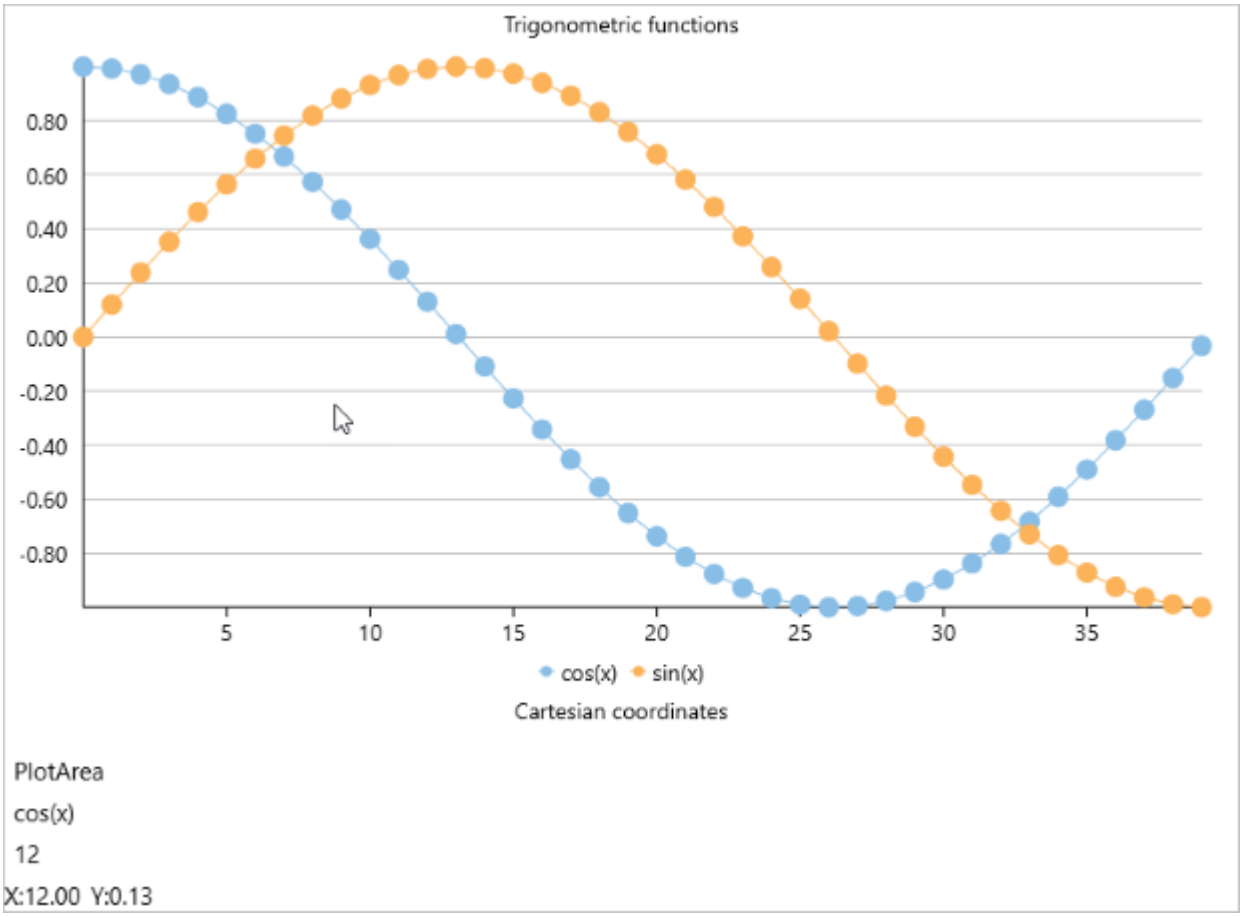
XAML

```
<c1:FlexChart x:Name="chart" ItemsSource="{Binding Data}" BindingX="Name"
ChartType="Column"
Header="Sales Data" HeaderAlignment="Fill" Footer="Random data by ComponentOne"
FooterAlignment="Fill">
</c1:FlexChart>
```

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 lies outside the series of the FlexChart.

The following image shows a LineSymbol chart with the index of the tapped point on the chart.



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.

1. Create a new **Cross Platform App** and open the MainPage.xaml page.
2. In the MainPage.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:c1="clr-
namespace:C1.Xamarin.Forms.Chart;assembly=C1.Xamarin.Forms.Chart"
             xmlns:local="clr-namespace:Flexchart_HitTest"
             x:Class="Flexchart_HitTest.MainPage">
  <StackLayout>
    <Grid VerticalOptions="FillAndExpand">
      <c1:FlexChart x:Name="flexChart" Tapped="flexChart_Tapped"
                Header="Trigonometric functions" Footer="Cartesian coordinates"
                BindingX="X" ChartType="LineSymbols" LegendPosition="Bottom" >
        <c1:FlexChart.Series>
          <c1:ChartSeries x:Name="seriesCosX"  Binding="Y" SeriesName="cos(x)"
/>
          <c1:ChartSeries x:Name="seriesSinX"  Binding="Y" SeriesName="sin(x)"
/>
        </c1:FlexChart.Series>
      </c1: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>
```

3. Open the MainPage.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;
```

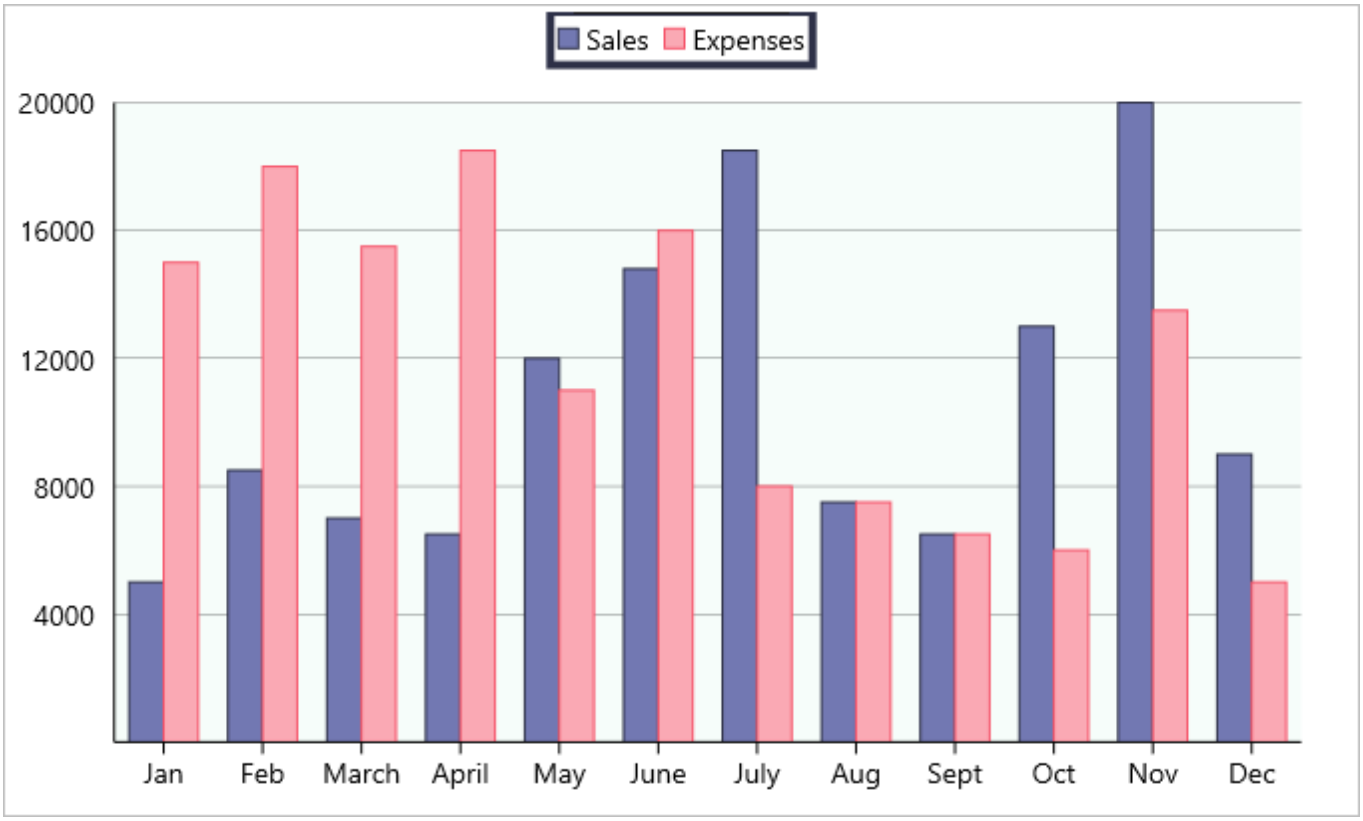
```
        }

        void flexChart_Tapped(object sender,
C1.Xamarin.Forms.Core.C1TappedEventArgs 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. By default, the position of legend is 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.

The following code examples demonstrate how to set these properties in C# and XAML. This example uses the sample created in the Customize Appearance section.

In Code

C#

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

In XAML

XAML

```
<c1:FlexChart x:Name="chart" ItemsSource="{Binding Data}" BindingX="Name"
ChartType="Column"
    Grid.Row="1" Grid.ColumnSpan="2" VerticalOptions="FillAndExpand"
LegendToggle="True"
    LegendPosition="Top" LegendOrientation="Horizontal">
        <c1:FlexChart.LegendStyle>
            <c1:ChartStyle Stroke="#2D3047" StrokeThickness="4"></c1:ChartStyle>
        </c1:FlexChart.LegendStyle>
</c1:FlexChart>
```

Line Marker

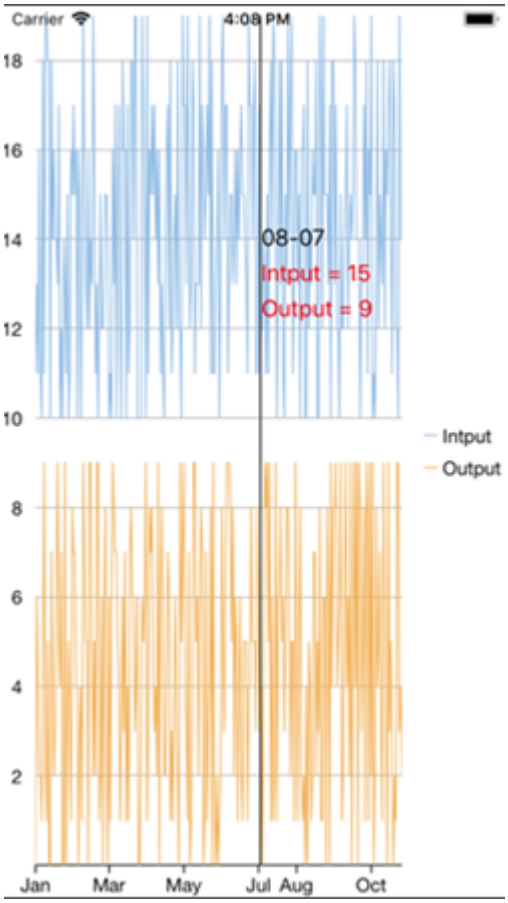
Line marker is a line that is drawn on chart plot and bound to some value on an axis. It may be used to show a trend or mark an important value on the chart. It displays the lines over the plot with an attached label. It is useful in scenarios, where a user has a lot of data in a line or area chart, or if a user wants to display data from multiple series in a single label. With built-in interactions, such as Drag and Move, a user can drag the line marker to select the data point on the chart more precisely.

If you set the Interaction property to Drag, you need to set the **DragContent** and the **DragLines** property to specify whether the content and values linked with the line marker lines are draggable or not.

To implement LineMarkers in the FlexChart, we use the following key properties.

- **Interaction** - It specifies that how a user can interact with the line marker: drag, move or none. It is an important property that defines how you can interact move or drag the marker to get the data precisely on the FlexChart.
- **Contents** - It allows the user to display UI elements in the label. These elements can be bound to values from the DataPoints collection.
- **Lines** - It allows you to choose the type of Lines you want to display: vertical, horizontal, or both (cross-hair) line.
- **Alignment** - It allows you to set the position of the label relative to VerticalPosition or HorizontalPosition.

The image below shows how the FlexChart appears after LineMarker is added to it.



The following code examples demonstrate how to set these properties in C# and in XAML. This example uses the sample created in the Quick Start section.

In Code

Add the following code in the **QuickStart.cs** file to display line marker for FlexChart control.

XAML

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

            this.flexChart.ItemsSource = new LineMarkerViewModel().Items;
            initMarker();
        }
        StackLayout layout = new StackLayout();
        Label xLabel = new Label();
        IList<Label> yLabels = new List<Label>();
        private void initMarker()
        {
            xLabel.VerticalOptions = LayoutOptions.FillAndExpand;
            xLabel.HorizontalOptions = LayoutOptions.FillAndExpand;
```

```
            layout.Children.Add(xLabel);

            for (int index = 0; index < flexChart.Series.Count; index++)
            {
                var series = flexChart.Series[index];
                var fill = (int)((IChart)flexChart).GetColor(index);
                Label yLabel = new Label();
                yLabel.VerticalOptions = LayoutOptions.FillAndExpand;
                yLabel.HorizontalOptions = LayoutOptions.FillAndExpand;
                var bytes = BitConverter.GetBytes(fill);
                yLabel.TextColor = Color.FromRgba(bytes[2], bytes[1], bytes[0],
bytes[3]);
                yLabels.Add(yLabel);
                layout.Children.Add(yLabel);
            }
            lineMarker.Content = layout;
        }
        private void OnLineMarkerPositionChanged(object sender, PositionChangedArgs
e)
        {
            if (flexChart != null)
            {
                var info = flexChart.HitTest(new Point(e.Position.X, double.NaN));
                int pointIndex = info.PointIndex;
                xLabel.Text = string.Format("{0:dd-MM}", info.X);

                for (int index = 0; index < flexChart.Series.Count; index++)
                {
                    var series = flexChart.Series[index];
                    var value = series.GetValues(0)[pointIndex];

                    var fill = (int)((IChart)flexChart).GetColor(index);
                    string content = string.Format("{0} = {1}", series.SeriesName,
string.Format("{0:f0}", value));
                    Label yLabel = yLabels[index];
                    yLabel.Text = content;
                }
            }
        }
        public class LineMarkerViewModel
        {
            const int Count = 300;
            Random rnd = new Random();

            public List<LineMarkerSampleDataItem> Items
            {
                get
                {
                    List<LineMarkerSampleDataItem> items = new
List<LineMarkerSampleDataItem>();
                    DateTime date = new DateTime(2016, 1, 1);
```

```
                    for (var i = 0; i < Count; i++)
                    {
                        var item = new LineMarkerSampleDataItem()
                        {
                            Date = date.AddDays(i),
                            Input = rnd.Next(10, 20),
                            Output = rnd.Next(0, 10)
                        };
                        items.Add(item);
                    }
                    return items;
                }
            }
            public List<string> LineType
            {
                get
                {
                    return Enum.GetNames(typeof(LineMarkerLines)).ToList();
                }
            }
            public List<string> LineMarkerInteraction
            {
                get
                {
                    return Enum.GetNames(typeof(LineMarkerInteraction)).ToList();
                }
            }
        }
        public class LineMarkerSampleDataItem
        {
            public int Input { get; set; }
            public int Output { get; set; }
            public DateTime Date { get; set; }
        }
    }
```

XAML Code

XAML

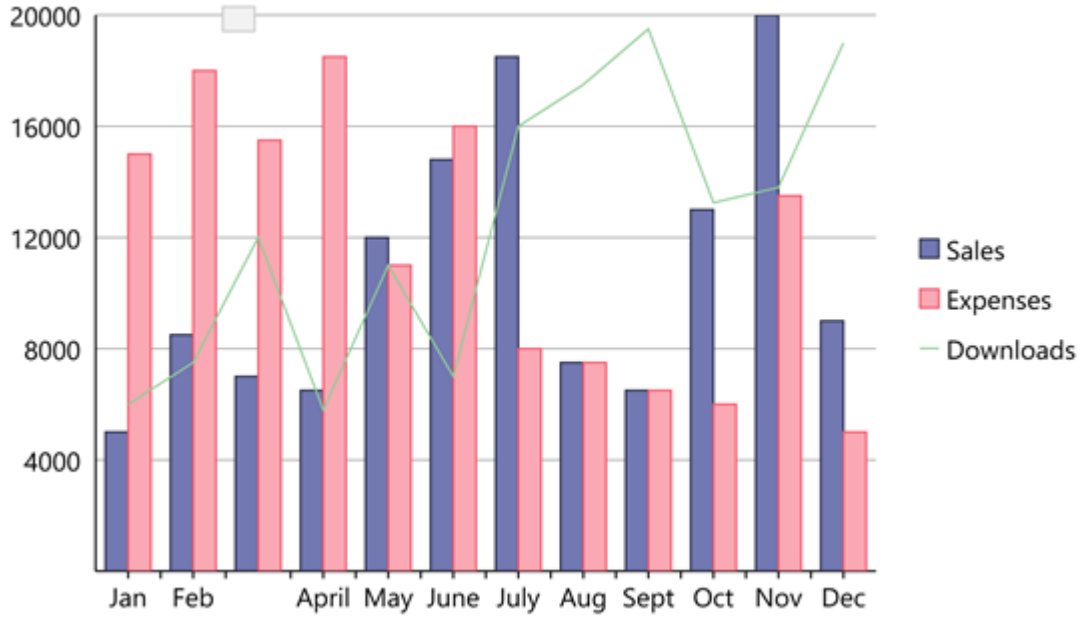
```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:chartFeatures"
             xmlns:c1="clr-
namespace:C1.Xamarin.Forms.Chart;assembly=C1.Xamarin.Forms.Chart"
             xmlns:interaction="clr-
namespace:C1.Xamarin.Forms.Chart.Interaction;assembly=C1.Xamarin.Forms.Chart"
             x:Name=""
             x:Class="chartFeatures.MainPage">
<StackLayout>
        <Grid  VerticalOptions="FillAndExpand">
```

```
            <c1:FlexChart x:Name="flexChart" ChartType="Line" BindingX="Date"
VerticalOptions="FillAndExpand" ShowTooltip="False">
                <c1:FlexChart.Series>
                    <c1:ChartSeries Binding="Input" SeriesName="Intput"/>
                    <c1:ChartSeries Binding="Output" SeriesName="Output"/>
                </c1:FlexChart.Series>
                <c1:FlexChart.AxisY>
                    <c1:ChartAxis MajorUnit="2" Position="Left" MajorGrid="True"
AxisLine="False" MajorTickMarks="None" />
                </c1:FlexChart.AxisY>
                <c1:FlexChart.Layers>
                    <interaction:C1LineMarker x:Name="lineMarker"  DragContent="True"
PositionChanged="OnLineMarkerPositionChanged">
                        <interaction:C1LineMarker.VerticalLineStyle>
                            <c1:ChartStyle Stroke="Gray" />
                        </interaction:C1LineMarker.VerticalLineStyle>
                        <interaction:C1LineMarker.HorizontalLineStyle>
                            <c1:ChartStyle Stroke="Gray" />
                        </interaction:C1LineMarker.HorizontalLineStyle>
                    </interaction:C1LineMarker>
                </c1:FlexChart.Layers>
            </c1:FlexChart>
        </Grid>
    </StackLayout>
</ContentPage>
```

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.



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

In Code

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);
```

In XAML

XAML

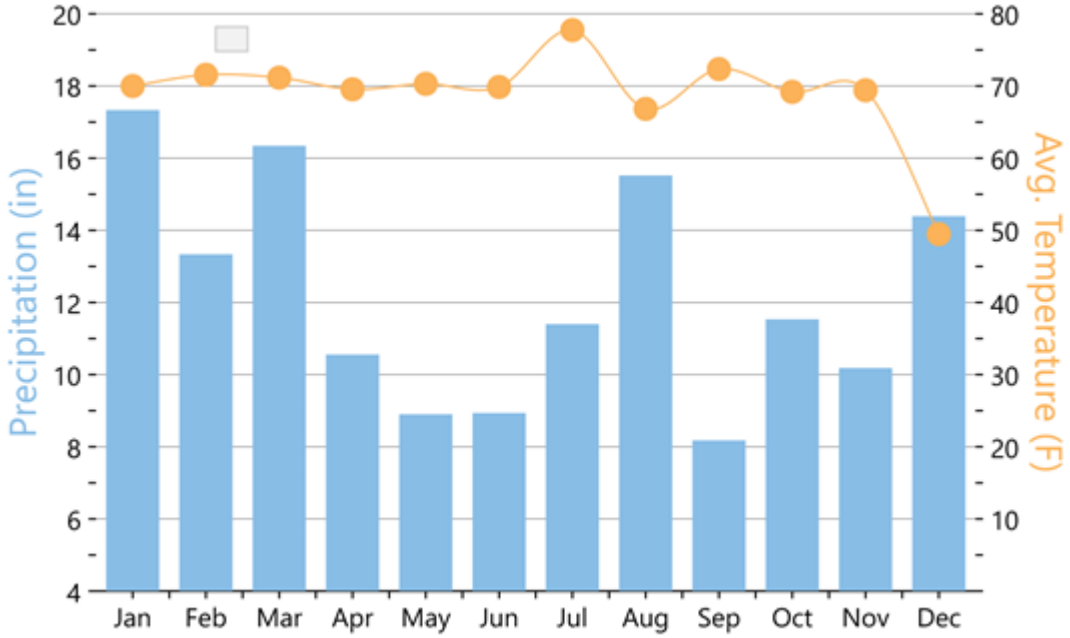
```
<c1:FlexChart.Series>
    <c1:ChartSeries x:Name="Sales2015" SeriesName ="Sales" Binding="Sales" >
        <c1:ChartSeries.Style>
            <c1:ChartStyle Fill="#7278B2" Stroke="#2D3047" />
        </c1:ChartSeries.Style>
    </c1:ChartSeries>
    <c1:ChartSeries x:Name="Expenses2015" SeriesName="Expenses" Binding="Expenses">
        <c1:ChartSeries.Style>
            <c1:ChartStyle Fill="#FAA9B4" Stroke="#F6546A" />
        </c1:ChartSeries.Style>
    </c1:ChartSeries>
    <c1:ChartSeries x:Name="Downloads2015" SeriesName="Downloads" Binding="Downloads"
        ChartType="Line"/>
</c1:FlexChart.Series>
```

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 displays two different series for Temperature and Precipitation, respectively.



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

1. Create a new **Cross Platform App** and open the MainPage.xaml page.
2. In the MainPage.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:c1="clr-
namespace:C1.Xamarin.Forms.Chart;assembly=C1.Xamarin.Forms.Chart"
             x:Class="QuickstartChart.MultipleYAxes">
  <c1:FlexChart x:Name="flexChart" BindingX="MonthName">
    <c1:FlexChart.Series>
      <c1:ChartSeries Binding="Precipitation" SeriesName="Precip"/>
      <c1:ChartSeries Binding="Temp" ChartType="SplineSymbols" SeriesName="Avg.
Temp">
        <c1:ChartSeries.AxisY>
          <c1:ChartAxis Position="Right" Min="0" Max="80" MajorUnit="10"
Title="Avg.Temperature (F)" AxisLine="False" MajorGrid="False">
            <c1:ChartAxis.TitleStyle>
              <c1:ChartStyle Stroke="#fbb258" FontSize="20"/>
            </c1:ChartAxis.TitleStyle>
          </c1:ChartAxis>
        </c1:ChartSeries.AxisY>
      </c1:ChartSeries>
    </c1:FlexChart.Series>

    <c1:FlexChart.AxisY>
```

```
      <c1:ChartAxis Min="4" Max="20" MajorUnit="2" MajorGrid="True"
Title="Precipitation (in)" AxisLine="False">
        <c1:ChartAxis.TitleStyle>
          <c1:ChartStyle Stroke="#88bde6" FontSize="20"/>
        </c1:ChartAxis.TitleStyle>
      </c1:ChartAxis>
    </c1:FlexChart.AxisY>
  </c1:FlexChart>
</ContentPage>
```

3. 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; }
    }
```

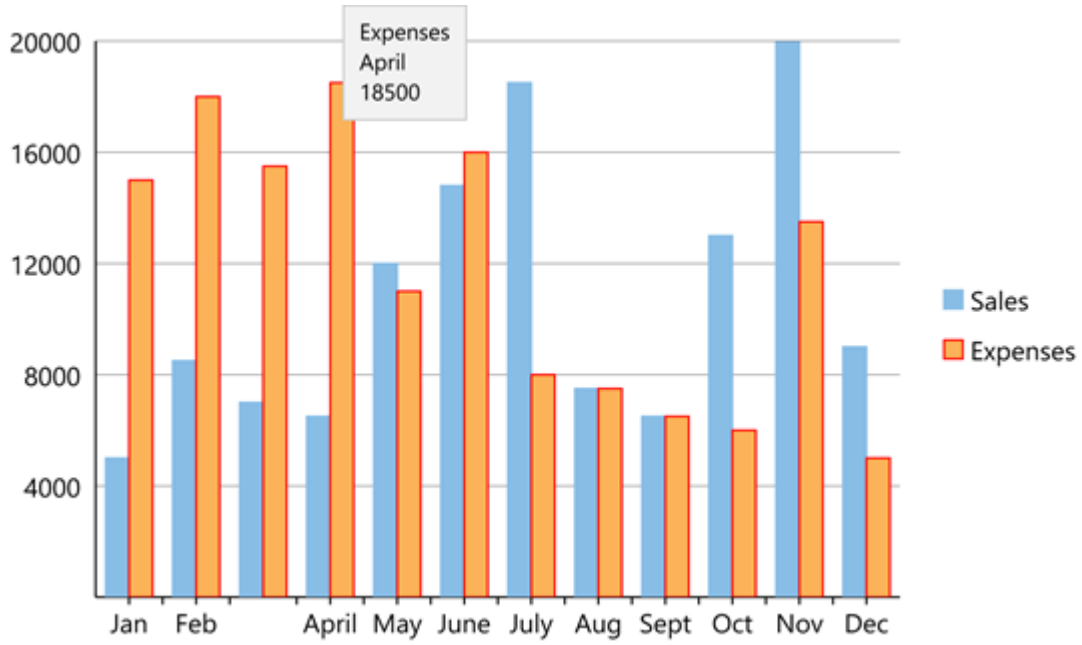
4. Press **F5** to run the project and view two Y axes against a common X axis.

Selection

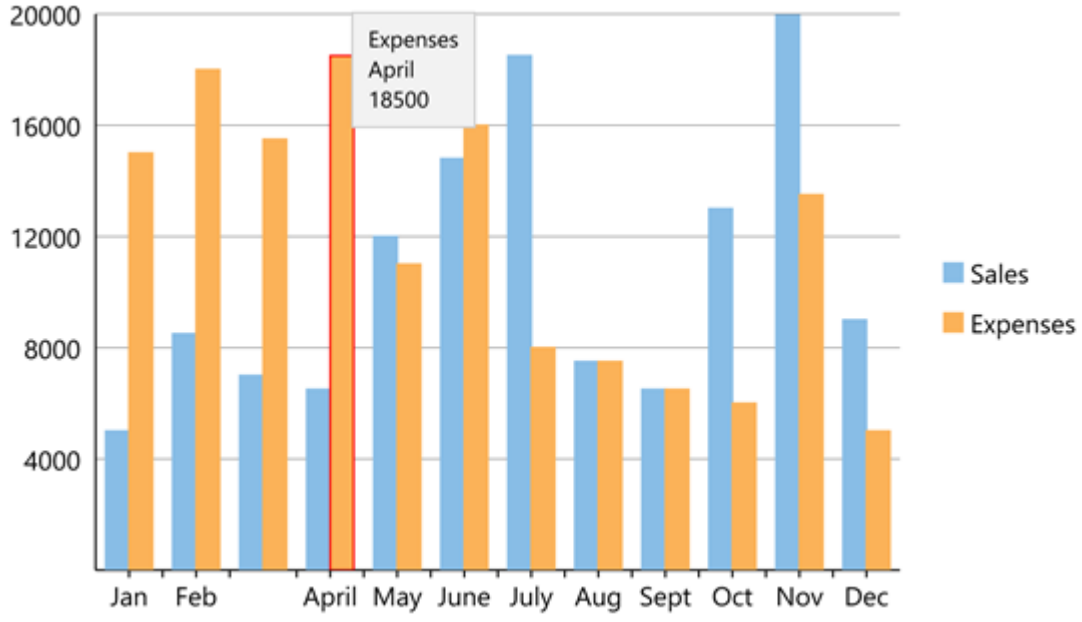
You can choose what element of the FlexChart should be selected when the user taps on any region in a FlexChart by setting the SelectionMode property. This property accepts one of the following values from ChartSelectionModeType enum:

- **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

The following code examples demonstrate how to set these properties in C# and XAML. This example uses the sample

created in the Quick Start section.

In Code

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 = ChartSelectionModeType.Point;
```

In XAML

XAML

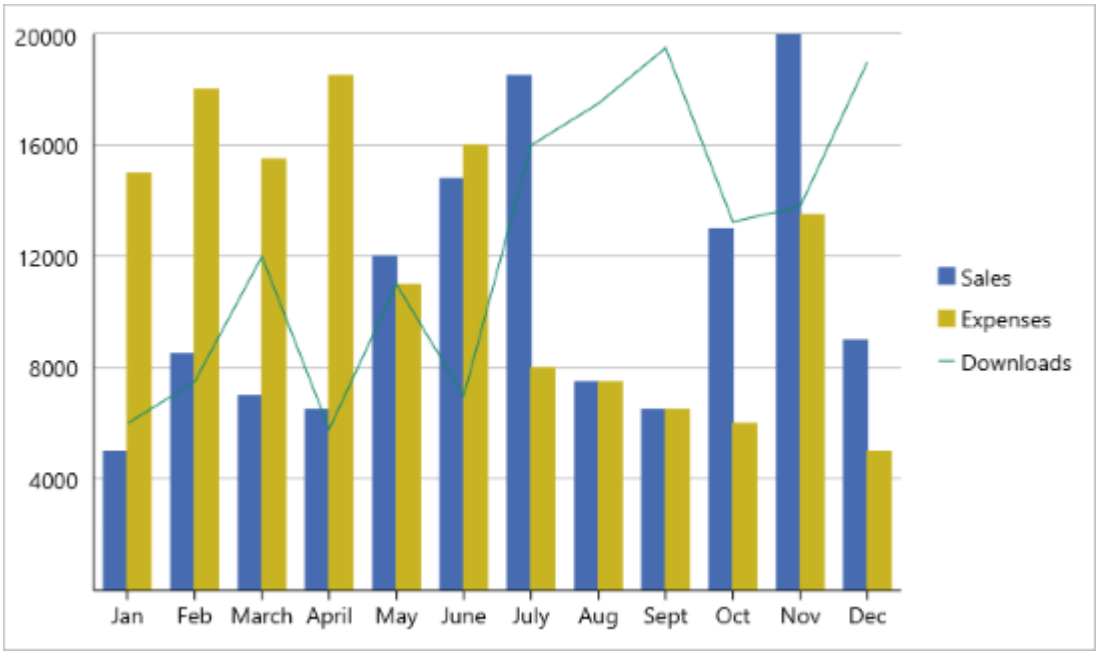
```
<c1:FlexChart x:Name="chart" ItemsSource="{Binding Data}" BindingX="Name"
ChartType="Column"
    Grid.Row="1" Grid.ColumnSpan="2" VerticalOptions="FillAndExpand"
SelectionMode="Point">
    <c1:FlexChart.Series>
        <c1:ChartSeries x:Name="Sales2015" SeriesName ="Sales" Binding="Sales" />
        <c1:ChartSeries x:Name="Expenses2015" SeriesName="Expenses"
Binding="Expenses"/>
    </c1:FlexChart.Series>
</c1:FlexChart>
```

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.



The following code examples demonstrate how to apply themes in C# and XAML.

In Code

C#

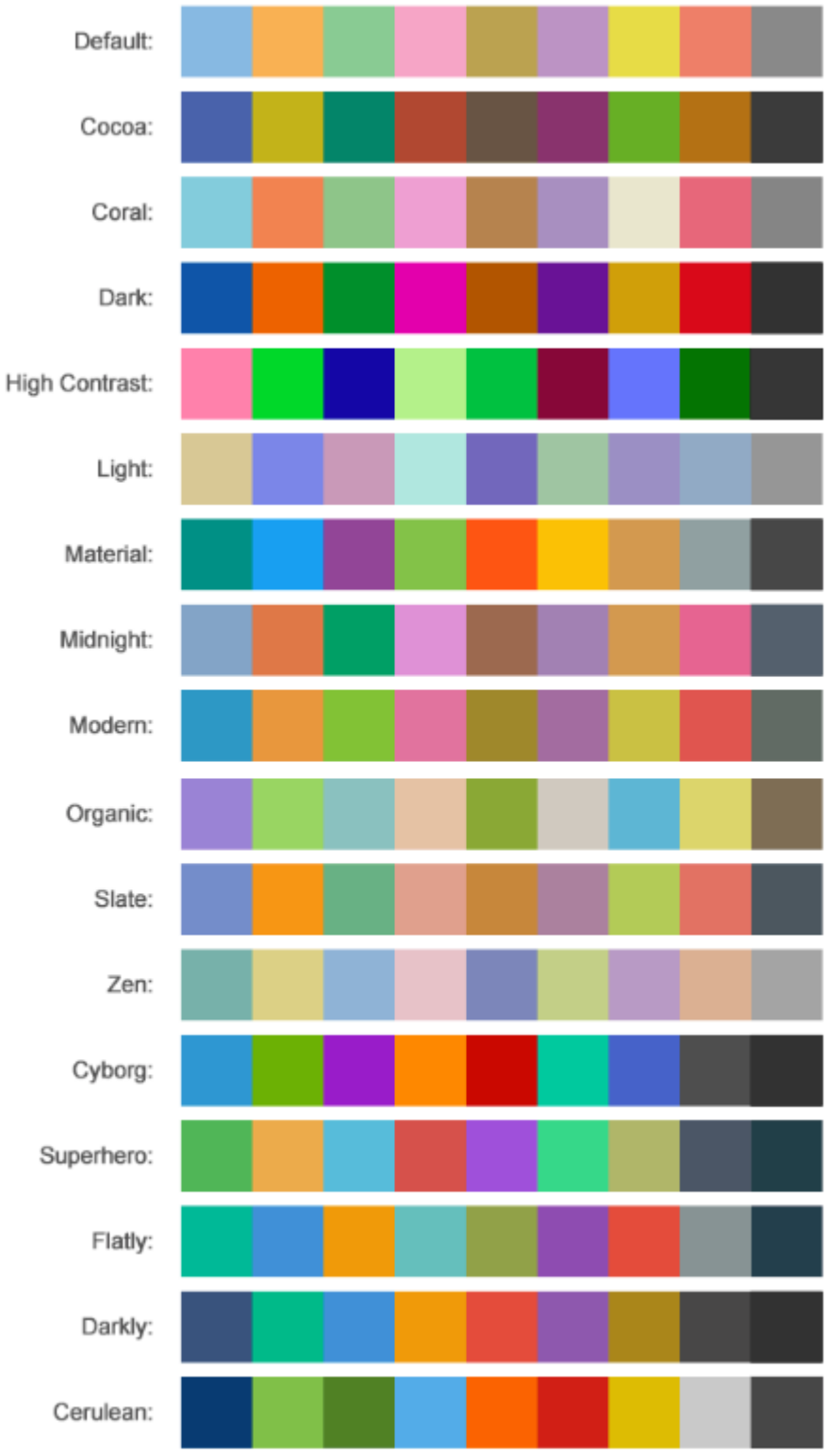
```
//setting the Palette
chart.Palette = Palette.Cocoa;
```

In XAML

XAML

```
<c1:FlexChart x:Name="chart" ItemsSource="{Binding Data}" BindingX="Name"
ChartType="Column"
    Grid.Row="1" Grid.ColumnSpan="2" VerticalOptions="FillAndExpand"
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.



Tooltip

Tooltips are the labels that appear when users hover over data points on your chart. They appear on FlexChart by default, and can display any combination of data values and text. FlexChart allows you to display content from predefined string values in the tooltip and display custom views that may contain labels or images using the **Tooltip.Content** property.

The easiest way to display content in Tooltip is by using the Content property. You need to configure the content property as per the type of entry you want to display in tooltip.

The table below lists the pre-defined parameters applicable for tooltip content.

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 chart slice. (FlexPie only)

To implement the Content property to display content from predefined string values, use the following code example given below.

Xaml

```
<c1:FlexChart x:Name="flexChart" BindingX="Name" ToolTipContent="{}{seriesName}{name}
{value}"/>
```

To implement the Content property to display custom view, use the following code examples given below. In the code below, Content property accepts a Xamarin layout and displays it in the form of a tooltip, when the user taps at any point on the FlexChart control.

Data Source

C#

```
public static class ChartSampleFactory
    {
       public static IEnumerable<SalesExpensesDownloadsEntity> CreateEntityList()
        {o
            List<SalesExpensesDownloadsEntity> entityList = new
List<SalesExpensesDownloadsEntity>();
            string[] countries = new string[] { "US", "Germany", "UK", "Japan",
"Italy", "Greece" };

            Random random = new Random();

            for (int i = 0; i < countries.Length; i++)
            {
                double sales = random.NextDouble() * 10000;
                double expenses = random.NextDouble() * 5000;
                double downloads = Math.Round(random.NextDouble() * 20000);

                entityList.Add(new SalesExpensesDownloadsEntity(countries[i], sales,
expenses, downloads, DateTime.Today.AddDays(i)));
            }
```

```
            return entityList;
        }
    }
```

XAML Code

Add the following code in the **CustomTooltip.xaml** file to initiate the FlexChart control and add tooltips. The images used in this sample are available at the following location: *Documents\ComponentOne Samples\Xamarin\XF\FlexChart101\FlexChart101.XF\Images*.

Xaml

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="FlexChart101.CustomTooltipsSample"
             xmlns:c1="clr-
namespace:C1.Xamarin.Forms.Chart;assembly=C1.Xamarin.Forms.Chart"
             xmlns:common="clr-namespace:FlexChart101;assembly=FlexChart101.XF"
x:Name="page">
<ContentPage.Resources>
    <ResourceDictionary>
      <common:FlagConverter x:Key="converter" />
    </ResourceDictionary>
  </ContentPage.Resources>
  <StackLayout>
    <Grid  VerticalOptions="FillAndExpand">
    <c1:FlexChart x:Name="chart" ItemsSource="{Binding Data}" BindingX="Name"
ChartType="Column" VerticalOptions="FillAndExpand">
      <c1:FlexChart.Series>
        <c1:ChartSeries Binding="Sales" SeriesName="Sales" />
        <c1:ChartSeries Binding="Expenses" SeriesName="Expenses" />
        <c1:ChartSeries Binding="Downloads" SeriesName="Downloads" />
      </c1:FlexChart.Series>

      <c1:FlexChart.ToolTip>
        <c1:ChartTooltip IsOpen="false" BackgroundColor="Blue" Padding="0">
          <StackLayout BackgroundColor="#FFFFCA" Padding="5">
            <StackLayout Orientation = "Horizontal">
              <Image x:Name="image" Source="{Binding ValueX, Mode=OneWay, Converter=
{StaticResource converter}"/>
              <Label x:Name="label1" Text="{Binding SeriesName}" TextColor="Black"
FontAttributes="Bold" FontSize="15"></Label>
            </StackLayout>
            <Label x:Name="label2" Text="{Binding DataY, StringFormat='{0:c2}'}"
TextColor="Black" FontSize="15" HorizontalOptions="Center"></Label>
          </StackLayout>
        </c1:ChartTooltip>
      </c1:FlexChart.ToolTip>
    </c1:FlexChart>
    </Grid>
  </StackLayout>
</ContentPage>
```

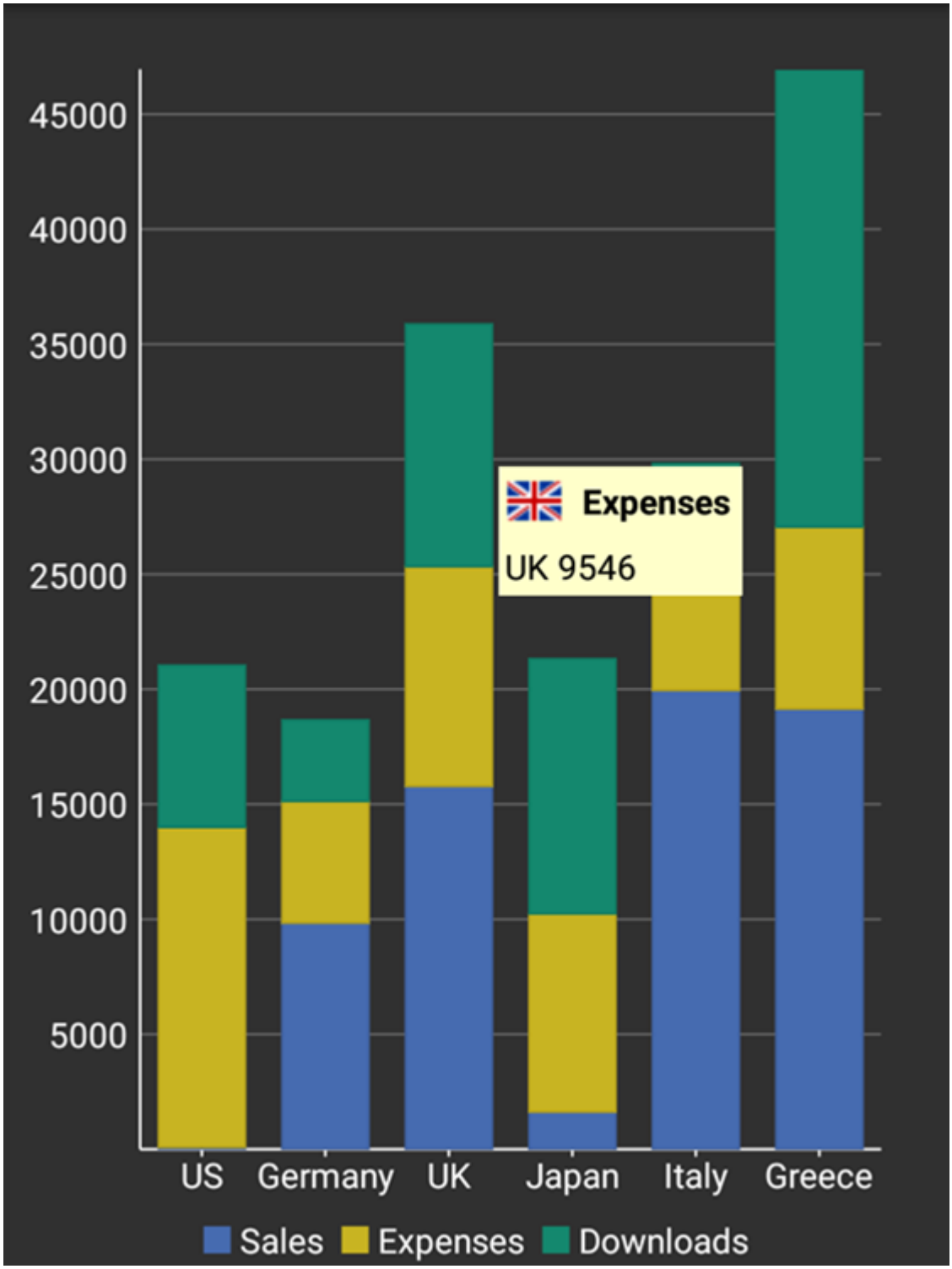
In Code

Add the following code in the **CustomTooltip.xaml.cs** file to bind the FlexChart control to data source.

C#

```
public CustomTooltips()
        {
            InitializeComponent();

            this.chart.ItemsSource = ChartSampleFactory.CreateEntityList();
            this.chart.Palette = Palette.Zen;
            this.chart.LegendPosition = ChartPositionType.Bottom;
            this.chart.Stacking = ChartStackingType.Stacked;
            chart.Palette = Palette.Cocoa;
        }
```

The image below shows how a custom view tooltip appears on FlexChart.

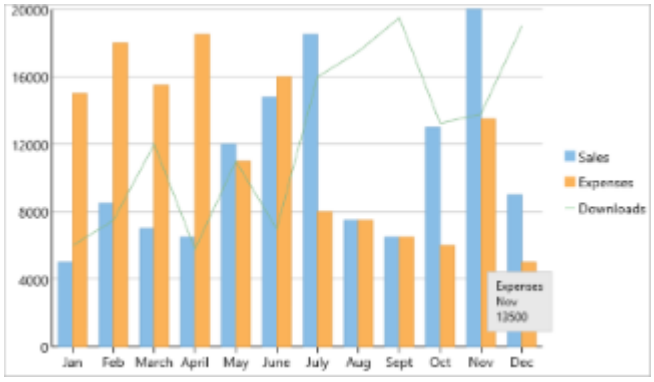
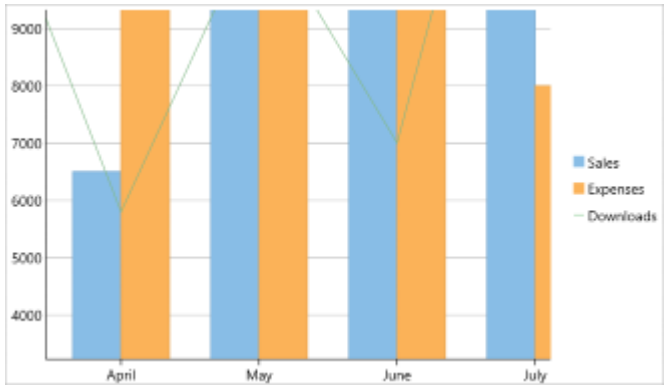


Zooming and Panning

Zooming can be performed in FlexChart using ZoomBehavior class. To implement zooming, you need to create an object of ZoomBehavior class available in the C1.Xamarin.Forms.Chart.Interaction namespace and pass it as a parameter to the Add method. This method adds zoom behavior to the behavior collection by accessing it through Behaviors property of the ChartBase class. In addition, you can use the ZoomMode property to enable touch based zooming in FlexChart. This property sets the gesture direction of zoom behavior through GestureMode enumeration which provides four zoom modes as given below:

- **None -** Disables zooming.
- **X -** Enables zooming along x-axis.
- **Y -** Enables zooming along y-axis.
- **XY -** Enables zooming along x and y axes.

The image below shows how FlexChart appears on zooming.

Before Zooming	After Zooming
	

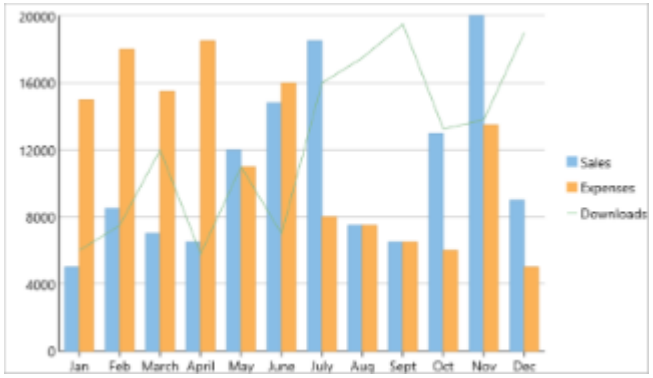
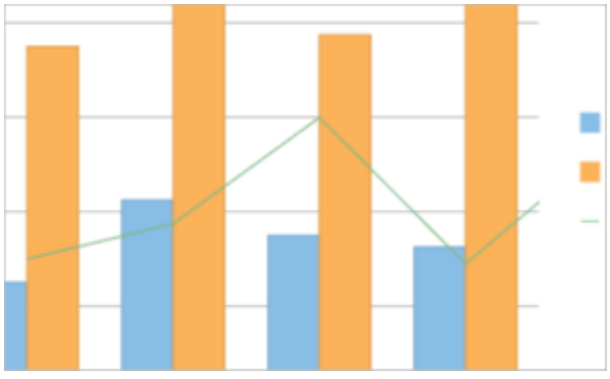
The following code examples demonstrate how to implement zooming in C#. These examples use the sample created in the Quick Start section.

C#

```
ZoomBehavior z = new ZoomBehavior();
z.ZoomMode = GestureMode.X;
chart.Behaviors.Add(z);
```

Similarly, panning can be implemented in FlexChart by creating an object of TranslateBehavior class available in the C1.Xamarin.Forms.Chart.Interaction namespace and passing it as a parameter to the Add method. This method adds translation behavior to the behavior collection by accessing it through Behaviors property of the ChartBase class.

The image below shows how FlexChart appears on panning.

Before Panning	After Panning
	

The following code examples demonstrate how to implement panning in C#. These examples use the sample created in the Quick Start section.

C#

```
TranslateBehavior t = new TranslateBehavior();
```

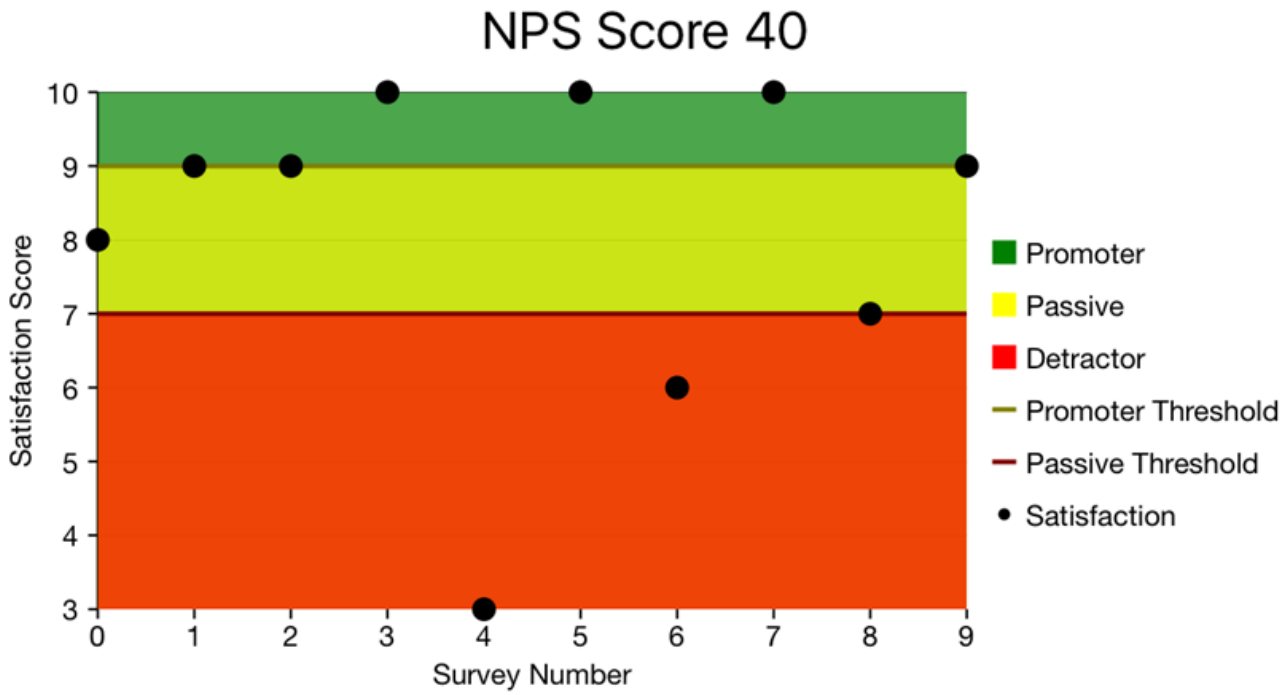
```
chart.Behaviors.Add(t);

chart.Scale = 1.5;
chart.AxisX.DisplayedRange = 4;
```

In addition to zooming and panning, FlexChart allows you to customize the relative range of values displayed in the view through Scale property, so if you set it to 0.5 it will display 50% of the axis in view (you can pan to the other 50%). Alternatively, FlexChart also allows you to set the absolute range of values displayed in the view through DisplayRange property. You can set this property on the chart axis or on the x or y axes separately.

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 **Content 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:c1="clr-
namespace:C1.Xamarin.Forms.Chart;assembly=C1.Xamarin.Forms.Chart"
             x:Class="QuickstartChart.SurveyZones">
  <StackLayout>
    <Label x:Name="lbl" HorizontalOptions="Center" Font="Large"/>
    <Grid VerticalOptions="FillAndExpand" >
      <c1:FlexChart x:Name="flex" BindingX="surveyNumber"
VerticalOptions="FillAndExpand"
                    ChartType="Scatter" Grid.Row="0">
        <c1:FlexChart.AxisX>
          <c1:ChartAxis AxisLine="false" MajorGrid="false" Title="Survey Number"
/>
        </c1:FlexChart.AxisX>
        <c1:FlexChart.AxisY>
          <c1:ChartAxis Title="Satisfaction Score" Max="10"/>
        </c1:FlexChart.AxisY>
      </c1:FlexChart>
    </Grid>
  </StackLayout>
</ContentPage>
```

Step 3: Bind data and create zones

1. Expand the SurveyZones.xaml node and open SurveyZones.xaml file.
2. Add the following import statements.

C#

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

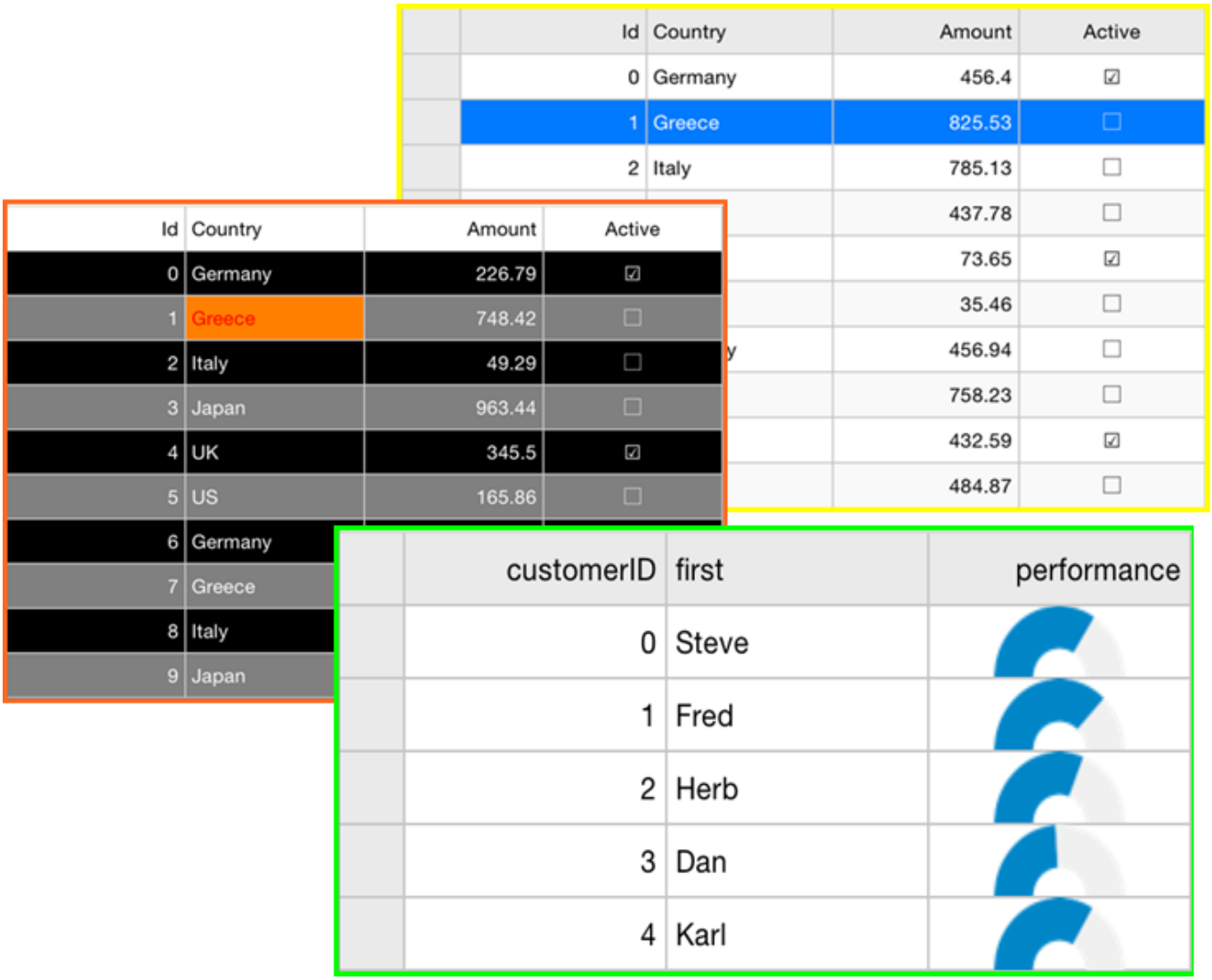
```
public partial class SurveyZones : ContentPage
    {
        public SurveyZones()
        {
            InitializeComponent();

            //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.Color = Color.Green;
            seriesZone1.Binding = "Y";
```

```
seriesZone1.BindingX = "X";
seriesZone1.ChartType = ChartType.Area;
seriesZone1.SeriesName = "Promoter";

//Zone 2 for passives
var seriesZone2 = new ChartSeries();
seriesZone2.ItemsSource = zone2;
seriesZone2.Color = Color.Yellow;
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.Color = Color.Red;
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.Color = Color.Olive;
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.Color = Color.Maroon;
seriesThreshold2.ChartType = ChartType.Line;
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";
```

```
            satisfactionSeries.Color = Color.Black;

            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. The control also allows you to reorder rows and columns without using any code.



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.

> Please note that from 2018v3 onwards, the default appearance of FlexGrid has changed to the material design pattern. For information on how to switch back to the classic view, see Material Theme.

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	Country ID	Active
	0	Steve Frommer	Congo	1	☑
	1	Oprah Orsted	Egypt	2	☑
	2	Charlie Orsted	Congo	1	☑
	3	Gil Krause	Egypt	2	☑
	4	Gil Neiman	Japan	4	☑
	5	Oprah Krause	Japan	4	☑
	6	Herb Orsted	Egypt	2	☑
	7	Gil Neiman	Egypt	2	☐
	8	Herb Neiman	Egypt	2	☐
	9	Xavier Frommer	Egypt	2	☑

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 C1.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 Content 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="App1.QuickStart"
   xmlns:c1="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">
         <c1:FlexGrid 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, specify the ItemsSource property for the FlexGrid.

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

   C#

   ```
   public QuickStart()
   {
       InitializeComponent();
       grid.ItemsSource = Customer.GetCustomerList(10);
   }
   ```

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.
   - **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 Content Page:** In the class constructor App( ), set the Content 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. (Optional) In case you compile your UWP application in **Release** mode, you need to explicitly add the following code to the **OnLaunched** method in your **App.xaml.cs** to include the correct assemblies with your application.

        C#
        ```
        var assembliesToInclude = new List<Assembly>();
        assembliesToInclude.Add(typeof(C1.Xamarin.Forms.Grid.Platform.UWP.FlexGridRenderer)
        .GetTypeInfo().Assembly);
        assembliesToInclude.Add(typeof(C1.UWP.Grid.FlexGrid).GetTypeInfo().Assembly);
        Xamarin.Forms.Forms.Init(e, assembliesToInclude);
        ```
4. Press **F5** to run the project.

Back to Top

Key Features

FlexGrid provides many different features that enable the developers to build intuitive and professional-looking

applications. The main features for FlexGrid are as follows:

- **Customizable appearance**

  The FlexGrid control is enriched with built-in properties to customize the grid's appearance. A user can set attributes such as alternating row color, background color, text color, header color, font etc. to customize the overall look of the control. It confers granular styling capabilities that are perfectly suited for Material themes.

- **Inline editing**

  FlexGrid enables a user to perform in-line editing using the default device keyboard. The quick edit mode within the cell ensures a flawless editing experience.

- **Filtering**

  FlexGrid supports filtering that allows a user to display subsets of data out of large data sets based on the criteria defined. A user can perform custom filtering by adding a text box to enter a value and display a particular set of data.

- **Frozen columns/rows**

  FlexGrid lets you freeze columns and rows so that they are always visible upon scrolling through a grid with a big number of columns/rows.

- **Material theme**

  FlexGrid has a default appearance of material design pattern. A user can programmatically switch to the classic style.

- **Pull-to-refresh**

  FlexGrid comes with pull-to-refresh feature. This lets a user refresh the contents of the screen of a hand-held device by dragging the screen downward with the finger.

- **Sorting**

  FlexGrid enables users to sort the grid data efficiently and quickly. You can easily tap a column's header to sort the grid by that column during runtime.

- **Animation**

  By default, FlexGrid confers fast, animated transitions for enhanced user experience. This can be observed in operations like dragging/dropping and expanding/collapsing groups.

Features

Reordering Rows and Columns

Reordering of rows and columns is a very common scenario when handling and analyzing the data in a grid. FlexGrid provides reordering by dragging the header cells at run-time. The feature is available by default and user can drag and reorder the rows and columns both with a very smooth transition which is important to enhance the overall user experience.

	Id	First Name	Last Name	Address
*	*Click here to add a new row*			
	0	Ulrich	Frommer	411 Panoramic E
	1	Ted	Evers	812 Fake AVE
	2	Paul	Bishop	378 Main ST S
	3	Mark	Bishop	753 Main BLVD
	4	Jack	Orsted	283 Main AVE SI
	5	Jack	Griswold	232 Panoramic A
	6	Ben	Trask	373 Panoramic A
	7	Fred	Heath	61 Park BLVD
	8	Vic	Bishop	4 Green BLVD
	9	Oprah	Frommer	863 Main BLVD
	10	Andy	Myers	857 Golden ST

However, you can change this behavior to limit dragging only to rows, or columns or to disable it completely. This can be achieved by using the AllowDragging property of the FlexGrid class which accepts the values from GridAllowDragging enumeration. You can also disable reordering of a particular row or column by setting the AllowDragging property of the GridRow or GridColumn class respectively.

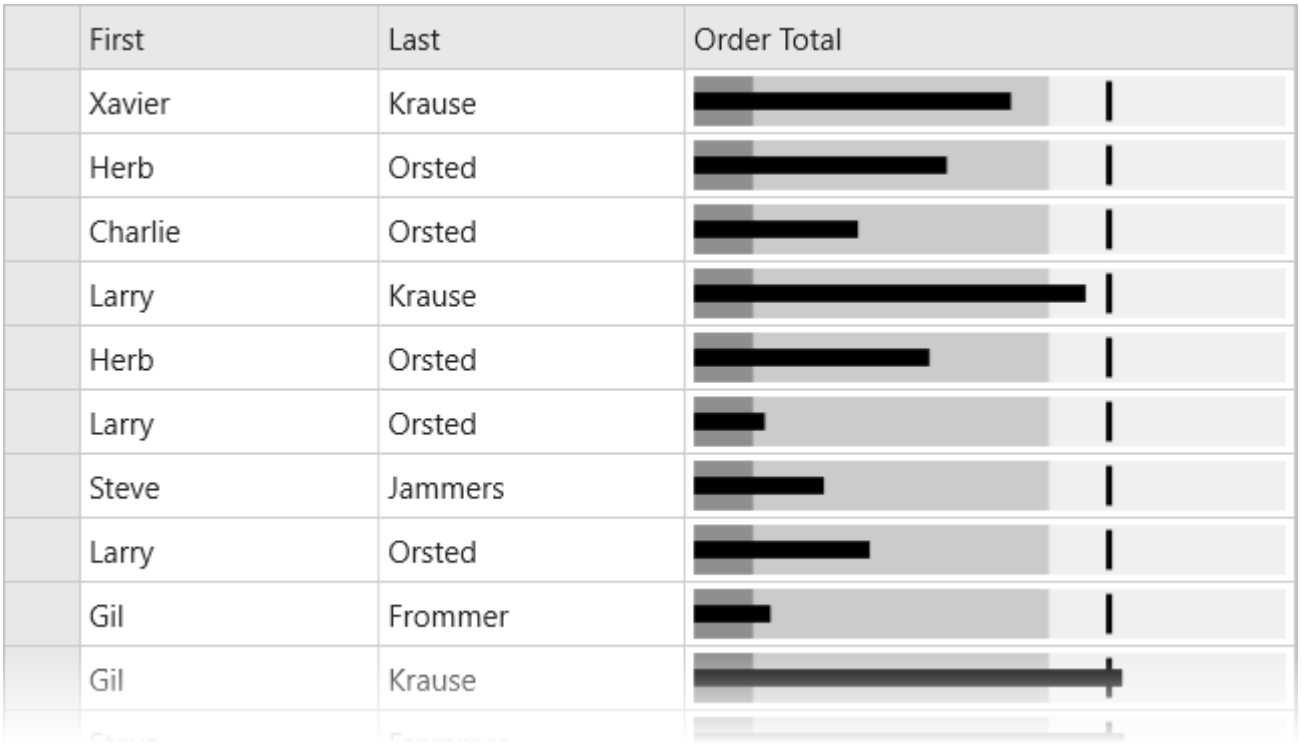
The following code snippet shows how you can disable column reordering in the FlexGrid control.

C#

```
// Disable column reordering
grid.AllowDragging = GridAllowDragging.None;
```

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.



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 **BulletGraph**, using the **CellTemplate** property on a specific column.

XAML

```
  <c1:FlexGrid AutoGenerateColumns="False" IsReadOnly="True">
         <c1:FlexGrid.Columns>
             <c1:GridColumn Binding="First" Width="*"/>
             <c1:GridColumn Binding="Last" Width="*"/>
             <c1:GridColumn Binding="OrderTotal" Header="Order Total" Width="*">
                 <c1:GridColumn.CellTemplate>
                     <DataTemplate>
                         <gauge:C1BulletGraph IsAnimated="False" IsReadOnly="True"
                                Value="{Binding OrderTotal}" Min="0" Max="10000"
Bad="1000"
                                Good="6000" Target="7000"/>
                     </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 &lt; 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

```
<c1:FlexGrid AutoGenerateColumns="False">
   <c1:FlexGrid.Columns>
      <c1:GridColumn Binding="Hired">
        <c1:GridColumn.CellEditingTemplate>
    <DataTemplate>
     <DatePicker Date="{Binding Hired, Mode=TwoWay}"/>
    </DataTemplate>
   </c1:GridColumn.CellEditingTemplate>
  </c1:GridColumn>
 </c1:FlexGrid.Columns>
</c1:FlexGrid>
```

Custom Icon

FlexGrid displays various icons during its operations such as sorting, filtering etc. These icons can be changed using various icon templates provided in the FlexGrid control. These icon templates can be accessed through following properties.

Properties	Description
SortAscendingIconTemplate	Allows you to set the template of sort icon for sorting values in ascending order.
SortDescendingIconTemplate	Allows you to set the template of sort icon for sorting values in descending order.
GroupExpandedIconTemplate	Allows you to set the template which is used to create the icon displayed when the group is expanded.
GroupCollapsedIconTemplate	Allows you to set the template which is used to create the icon displayed when the group is collapsed.

EditIconTemplate	Allows you to set the template which is used to create the icon displayed in the header when a row is being edited.
NewRowIconTemplate	Allows you to set the template which is used to create the icon displayed in the header of a new row.
DetailCollapsedIconTemplate	Allows you to set the template which is used to create the icon displayed when the detail is collapsed.
DetailExpandedIconTemplate	Allows you to set the template which is used to create the icon displayed when the detail is expanded.

You can change the icons set by these templates either to the built-in icons provided by the FlexGrid or to your own custom image, geometric figures, font etc as an icon.

FlexGrid also allows you to change the appearance of the different icons used in the control using the C1Icon class. The C1Icon class is an abstract class that provides a series of different objects that can be used for displaying monochromatic icons which can easily be tinted and resized. You can also change the position of these icons by setting the **SortIconPosition** property.

C#

```
grid.SortIconPosition = GridSortIconPosition.Left;
```

Using built-in Icons

To set the built-in icons for the abovementioned templates, you can set the following properties of the C1IconTemplate class.

Icon	Image
Edit	
Asterisk	
ArrowUp	
ArrowDown	
ChevronUp	
ChevronDown	
ChevronLeft	
ChevronRight	
TriangleNorth	
TriangleSouth	
TriangleEast	
TriangleWest	
TriangleSouthEast	
Star5	

For instance, to change the default sort ascending icon to a built-in icon, for example, TriangleNorth, use the following

code:

```
C#
grid.SortAscendingIconTemplate = C1IconTemplate.TriangleNorth;
```

Using Custom Icons

FlexGrid also allows you to set your own custom image, font, or path as an icon through the respective classes.

Icon Type	Icon Class Name
Bitmap/Image	C1BitmapIcon class
Font character	C1FontIcon class
Path	C1PathIcon class (child class of C1VectorIcon class)

For instance, to change the default sort descending icon to a custom image, use the following code:

In C#

```
C#
grid.SortDescendingIconTemplate =new C1IconTemplate(() => { return
new C1BitMapIcon{ Source = "arrow_down.png" }; });
```

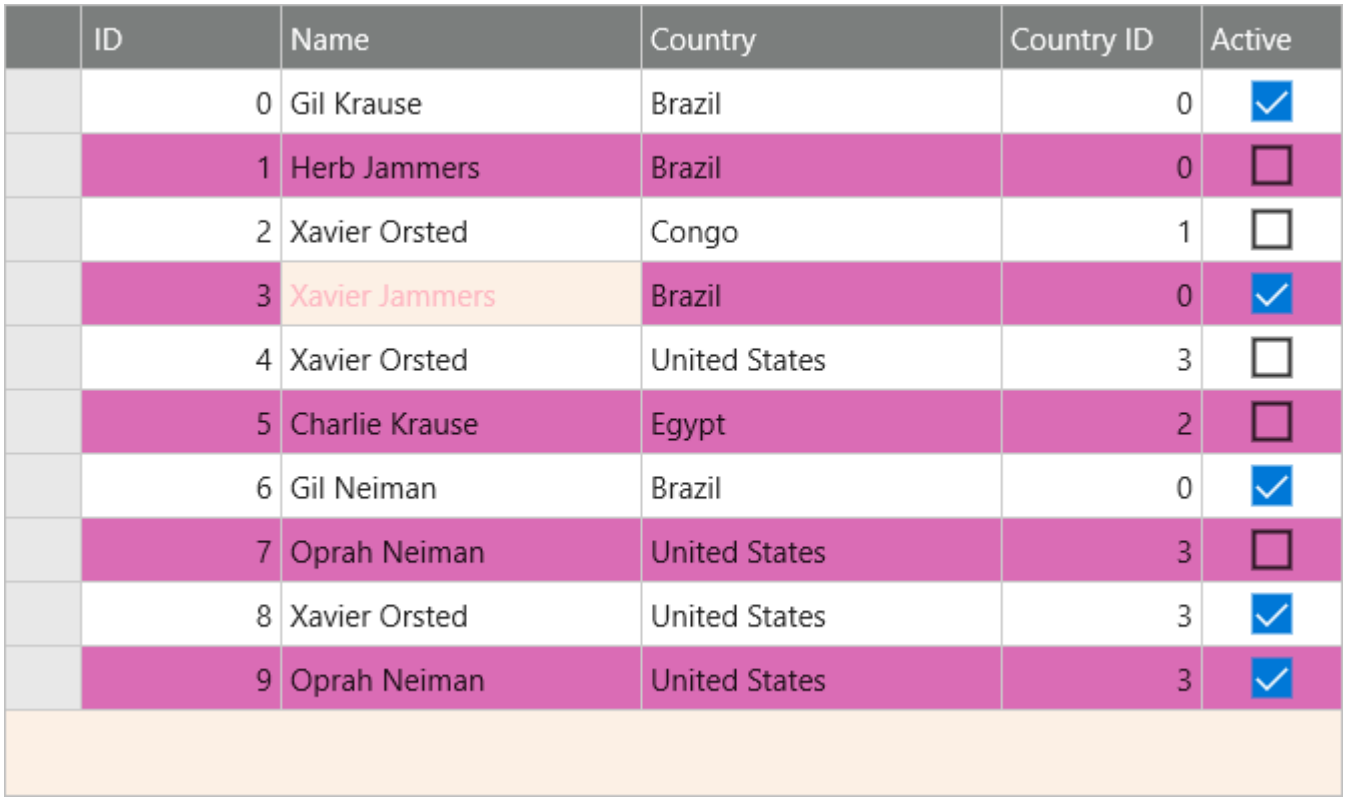
In XAML

```
XAML
<c1:FlexGrid x:Name="grid">
        <c1:FlexGrid.SortAscendingIconTemplate>
                <core:C1IconTemplate>
                        <DataTemplate>
                                <core:C1BitmapIcon Source="arrow_up.png"/>
                        </DataTemplate>
                </core:C1IconTemplate>
        </c1:FlexGrid.SortAscendingIconTemplate>
</c1:FlexGrid>
```

Customize Appearance

FlexGrid has various built-in properties to customize grid's appearance. A user can set attributes such as background color, alternating row color, text color, header color, font, selection mode, selected cell color, etc to customize the overall appearance of the FlexGrid control. Moreover, it also allows you to set individual properties for RowHeaderGridLinesVisibility, ColumnHeaderGridLinesVisibility, and TopLeftHeaderGridLinesVisibility that provides more granular styling capabilities that are more suited for Material themes.

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



	ID	Name	Country	Country ID	Active
	0	Gil Krause	Brazil	0	☑
	1	Herb Jammers	Brazil	0	☐
	2	Xavier Orsted	Congo	1	☐
	3	Xavier Jammers	Brazil	0	☑
	4	Xavier Orsted	United States	3	☐
	5	Charlie Krause	Egypt	2	☐
	6	Gil Neiman	Brazil	0	☑
	7	Oprah Neiman	United States	3	☐
	8	Xavier Orsted	United States	3	☑
	9	Oprah Neiman	United States	3	☑

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("#FFB6C1");
grid.ColumnHeaderBackgroundColor = Color.FromHex("#7B7E7D");
grid.ColumnHeaderTextColor= Color.FromHex("#FFFFFF");
grid.SelectionBackgroundColor = Color.FromHex("#FCF0E5");
grid.RowHeaderGridLinesVisibility = GridLinesVisibility.Vertical;
```

In XAML

XAML

```
<c1:FlexGrid SelectionMode="Cell"
BackgroundColor="White" AlternatingRowBackgroundColor="Purple" SelectionMode=Cell
ColumnHeaderBackgroundColor="Silver" ColumnHeaderTextColor="White"
SelectionBackgroundColor="Navy" SelectionTextColor="LightPink" x:Name="grid"
AutoGenerateColumns="False" RowHeaderGridLinesVisibility="Vertical">
```

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.

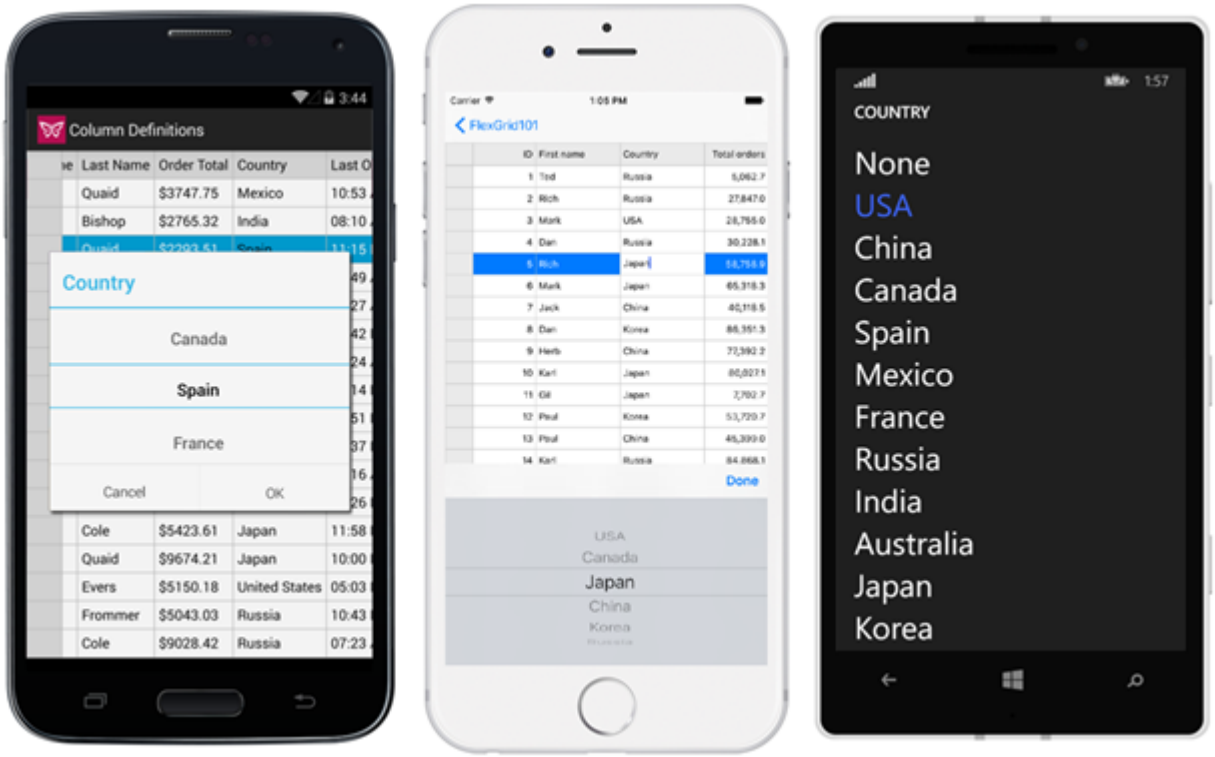
The supported keystrokes and their purpose are listed alphabetically as follows:

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.
Tab	Navigates the cell selection to the next column and then wraps to the next row. To enable the Tab key action, you need to set the **KeyActionTab** property to **Cycle**. This property accepts value from **GridTabAction** enumeration.
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

```
<c1:GridColumn Binding="CountryId" Header="Country">
  <c1:GridColumn.DataMap>
    <c1:GridDataMap ItemsSource="{Binding Countries, Source={StaticResource
viewModel}}"
                    SelectedValuePath="ID"
                    DisplayMemberPath="Name" />
  </c1:GridColumn.DataMap>
```

```
</c1: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	Order Total
	0	Gil	Orsted	1,970.74
	1	Gil	Krause	4,592.31
	2	Oprah	Frommer	2,713.43
	3	Herb	Neiman	3,226.71
	4	Xavier	Krause	1,184.79
	5	Herb	Orsted	4,510.17
	6	Steve	Neiman	2,837.17
	7	Oprah	Neiman	3,357.98
	8	Oprah	Jammers	3,586.06
	9	Oprah	Orsted	9,383.65
	10	Steve	Neiman	1,099.43
	11	Charlie	Orsted	1,538.42

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.

1. Add the following code to the MainPage.xaml page for defining columns in the FlexGrid.

   XAML

   ```
   <Grid VerticalOptions="FillAndExpand">
   ```

```
    <c1:FlexGrid x:Name="grid" AutoGenerateColumns="False">
       <c1:FlexGrid.Columns>
         <c1:GridColumn Binding="ID" Width="100"/>
         <c1:GridColumn Binding="First" Width="Auto"/>
         <c1:GridColumn Binding="Last"/>
         <c1:GridColumn Binding="OrderTotal" Format="N2"/>
       </c1:FlexGrid.Columns>
     </c1:FlexGrid>
</Grid>
```

2. Open MainPage.xaml.cs and add the following code to display data in the FlexGrid control.

C#

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

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.

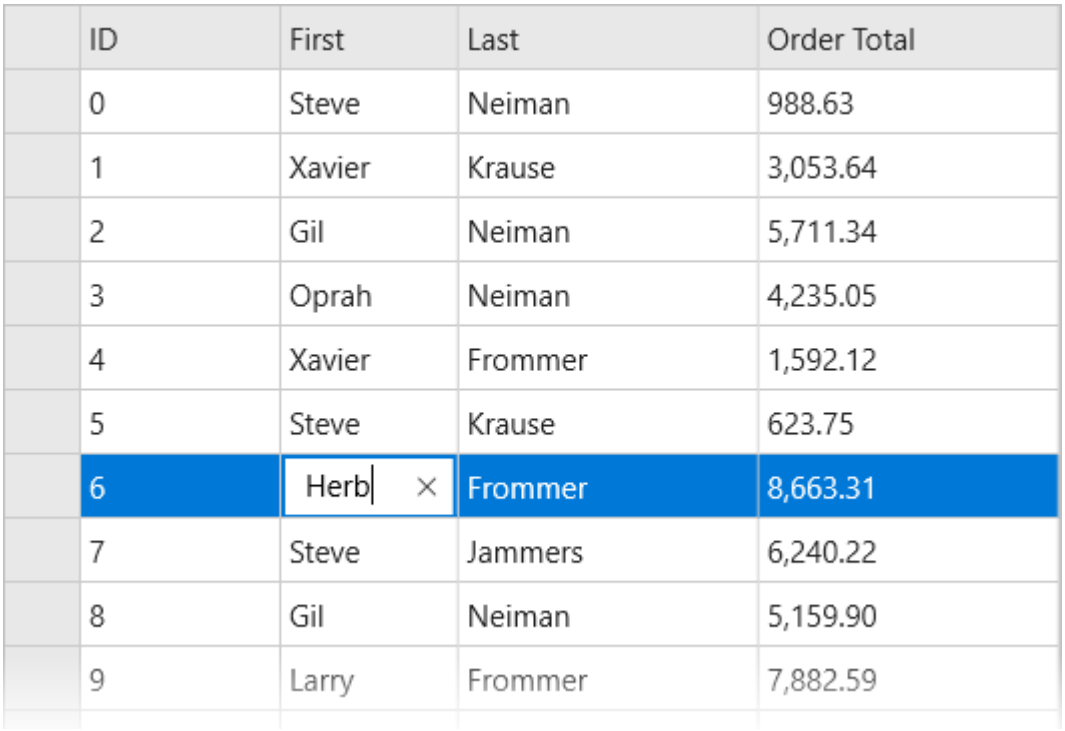
FlexGrid provides support for various types of Editing, including inline, form-based and custom cell editing.

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 restricting editing in the rows and columns.

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



	ID	First	Last	Order Total
	0	Steve	Neiman	988.63
	1	Xavier	Krause	3,053.64
	2	Gil	Neiman	5,711.34
	3	Oprah	Neiman	4,235.05
	4	Xavier	Frommer	1,592.12
	5	Steve	Krause	623.75
	6	Herb	Frommer	8,663.31
	7	Steve	Jammers	6,240.22
	8	Gil	Neiman	5,159.90
	9	Larry	Frommer	7,882.59

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

```
<c1:FlexGrid IsReadOnly="False" x:Name="grid"/>
```

Add New Row

FlexGrid allows you to show a new row template at the top or bottom of the grid using NewRowPosition property, which takes values from the GridNewRowPosition enumeration. You can control the text to be displayed to a user by setting the NewRowPlaceHolder property. Users may use the new row template to add items to the grid's itemsSource collection.

The image below shows how the FlexGrid appears after adding the new row.

	Id	First Name	Last Name	Address	City	Co
*	*Tap to begin entering a new row*					
	0	Charlie	Paulson	916 Green	Saint Petersburg	Ru
	1	Andy	Bishop	188 Panoramic	Brasilia	Br
	2	Ted	Heath	925 Green	Recife	Br
	3	Rich	Quaid	562 Grand ST	Jakarta	Ind
	4	Karl	Paulson	469 Green AVE	Moscow	Ru
	5	Vic	Stevens	446 Panoramic	Chelyabinsk	Ru
	6	Dan	Myers	739 Golden	Islamabad	Pa
	7	Fred	Jammers	555 Grand	Beijing	Ch
	8	Andy	Griswold	405 Park ST	Kawasaki	Ja

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.NewRowPosition = GridNewRowPosition.Top;
grid.NewRowPlaceholder = "Tap to begin entering a new row";
```

In XAML

XAML

```
<c1:FlexGrid x:Name="grid" NewRowPosition="Top" NewRowPlaceholder="Tap to begin
entering a new row"/>
```

Export

FlexGrid allows you to export a file and save it to a device or a stream. You export files to text, CSV and HTML formats and save them to a file system, a Stream or a StreamWriter using Save method of the FlexGrid class. The **Save** method can save the exported file with different options of encoding and presentation of the FlexGrid data. The method has following seven overloads:

Overload	Description
Save(StreamWriter, GridFileFormat, GridSaveOptions)	Saves the contents of the grid to a System.IO.StreamWriter.
Save(Stream, GridFileFormat, Encoding, GridSaveOptions)	Saves the contents of the grid to a stream with specified memory location, format, encoding, and options.
Save(Stream, GridFormat, GridSaveOptions)	Saves the contents of the grid to a UTF8 encoded stream with specified memory location, format and options.
Save(Stream, GridFileFormat)	Saves the contents of the grid to a UTF8 encoded stream with specified memory location and format.
Save(String, GridFileFormat, Encoding,	Saves the contents of the grid to a file a with specified name, format,

GridSaveOptions)	encoding, and options.
Save(String, GridFileFormat, GridSaveOptions)	Saves the contents of the grid to a UTF8 encoded file with specified name, format and options.
Save(String, GridFileFormat)	Saves the content of the grid to a UTF8 encoded file with specified name and format.

The following code example demonstrates the implementation of the Save method to save the exported file. The example uses the sample created in the Quick Start section.

C# copyCode

```
//Export FlexGrid
string PathAndName = Path.Combine(
    Environment.GetFolderPath(
        Environment.SpecialFolder.LocalApplicationData),
    "ExportedGrid") + "." + "csv";

grid.Save(PathAndName, GridFileFormat.Csv,
    System.Text.Encoding.UTF8, GridSaveOptions.SaveColumnHeaders);
```

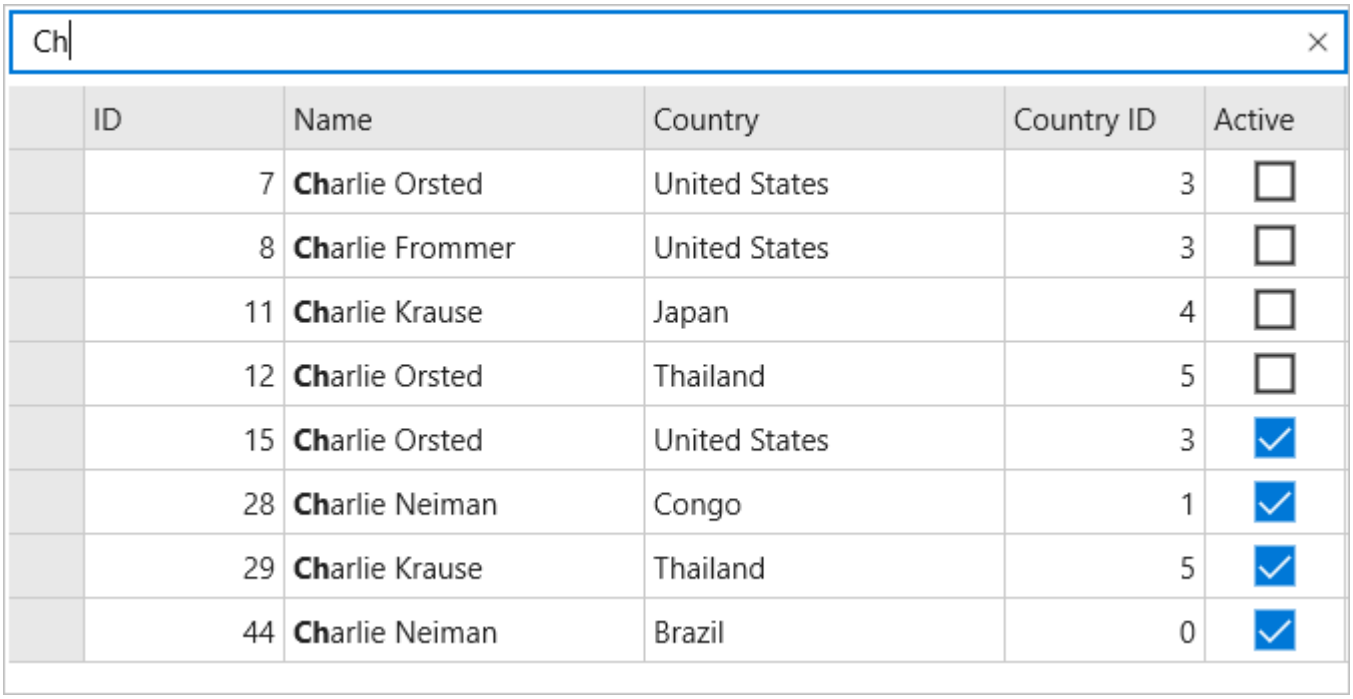
Filtering

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 text box that lets you type the value you want to search in the grid. For example, when you type **Ch** in the Filter text box, the collection view interface filters the grid data to display all the values containing **Ch**.



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 Content Page, Filter.xaml, to your project.
2. To initialize the FlexGrid control and perform filtering using XAML code, modify the markup between the <ContentPage> </ContentPage>tags as shown below.

In XAML

XAML

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:c1="clr-
namespace:C1.Xamarin.Forms.Grid;assembly=C1.Xamarin.Forms.Grid"
             x:Class="FlexGridFeatures.Filter">
    <ContentPage.Content>
        <StackLayout Orientation="Vertical" Margin="20,40,0,10">
            <Entry x:Name="filterEntry" Placeholder="Filter Text"></Entry>
            <Grid VerticalOptions="FillAndExpand">
                <c1:FlexGrid x:Name="grid" AutoGenerateColumns="True">
                    <c1:FlexGrid.Behaviors>
                        <c1:FullTextFilterBehavior FilterEntry="{x:Reference
Name=filterEntry}"
                            Mode="WhileTyping" MatchNumbers="True"
TreatSpacesAsAndOperator="True" />
                    </c1:FlexGrid.Behaviors>
                </c1:FlexGrid>
            </Grid>
        </StackLayout>
    </ContentPage.Content>
</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 below the InitializeComponent() method to display data in the FlexGrid control:

In Code

C#

```
grid.ItemsSource = 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.

The following code example demonstrates how to specify a column's format in C# and XAML. The example uses the sample created in the Quick start section.

In Code

C#

```
grid.Columns["OrderTotal"].Format = "N2";
```

In XAML

XAML

```
<c1:FlexGrid.Columns>
    <c1:GridColumn Binding="OrderTotal" Format="N2"/>
</c1:FlexGrid.Columns>
```

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 the previous format (M/dd/yyyy) to "yyyy-MM" format.

In Code

C#

```
grid.Columns["Hired"].Format = "yyyy-MM";
```

In XAML

XAML

```
<c1:FlexGrid.Columns>
    <c1:GridColumn Binding="Hired" Format="yyyy-MM"/>
</c1:FlexGrid.Columns>
```

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

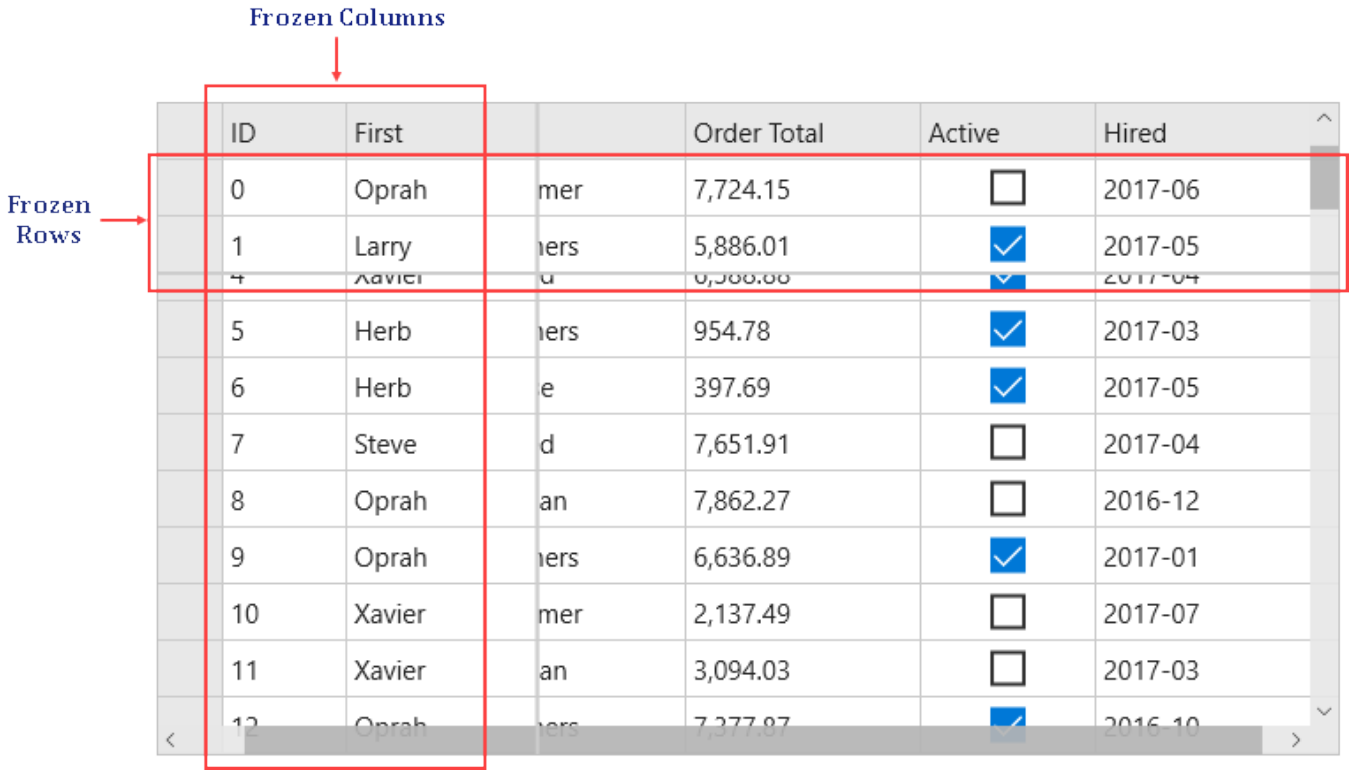
If you need to format a column that is not numerical or date/time, refer to Custom Cells and Data Mapping topics.

Frozen Rows and Columns

FlexGrid allows you to freeze rows and columns so that they remain visible as the user scrolls through the grid. 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. Frozen cells can be edited and selected as regular cells, exactly as in Excel. A user can freeze rows and columns separately as well as simultaneously.

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:



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

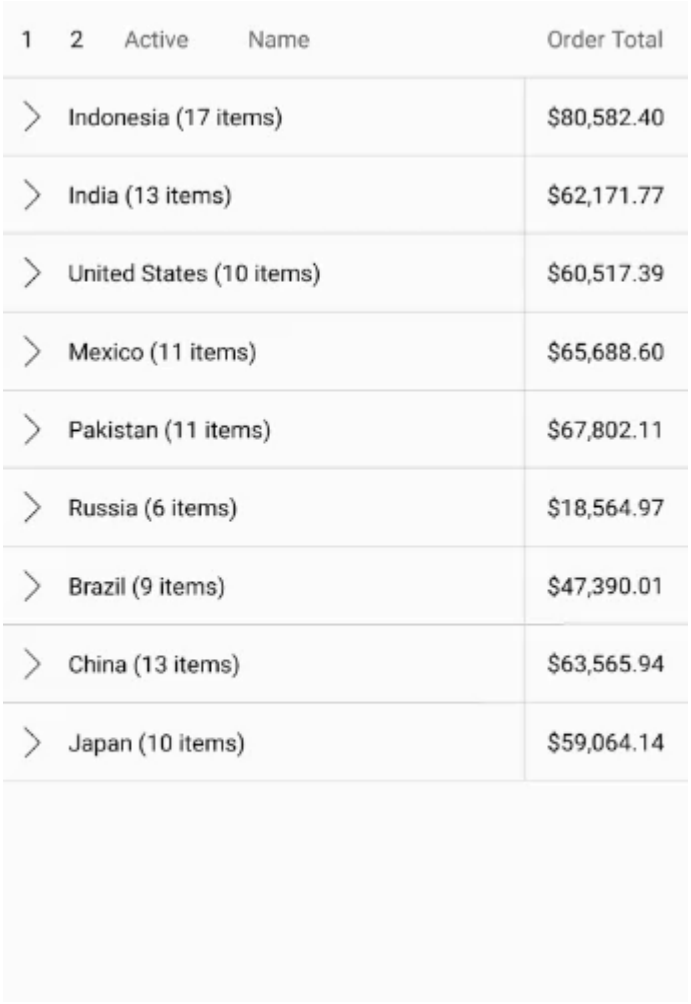
XAML

```
<Grid>
  <c1: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.



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 Content 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"
             xmlns:c1="clr-
namespace:C1.Xamarin.Forms.Grid;assembly=C1.Xamarin.Forms.Grid"
             x:Class="FlexGridFeatures.Grouping">

    <StackLayout Orientation="Vertical">
```

```
        <Grid VerticalOptions="FillAndExpand">
            <c1:FlexGrid x:Name="grid"  AutoGenerateColumns="False"
ShowOutlineBar="True" GridLinesVisibility="Vertical" IsReadOnly="True">
                <c1:FlexGrid.Columns>
                    <c1:GridColumn Binding="Active" Width="Auto"/>
                    <c1:GridColumn Binding="Name" Width="*"/>
                    <c1:GridColumn Binding="OrderTotal" Width="Auto" Format="C"
Aggregate="Sum" HorizontalAlignment="End"/>
                </c1:FlexGrid.Columns>
                <c1:FlexGrid.Behaviors>
                    <c1:EmptyGridBehavior EmptyView="{x:Reference
Name=emptyListLabel}" />
                </c1:FlexGrid.Behaviors>
            </c1:FlexGrid>
            <Label x:Name="emptyListLabel"
            Text="There are no items to show."
            FontSize="Large"
            HorizontalOptions="Center"/>
        </Grid>
    </StackLayout>
</ContentPage>
```

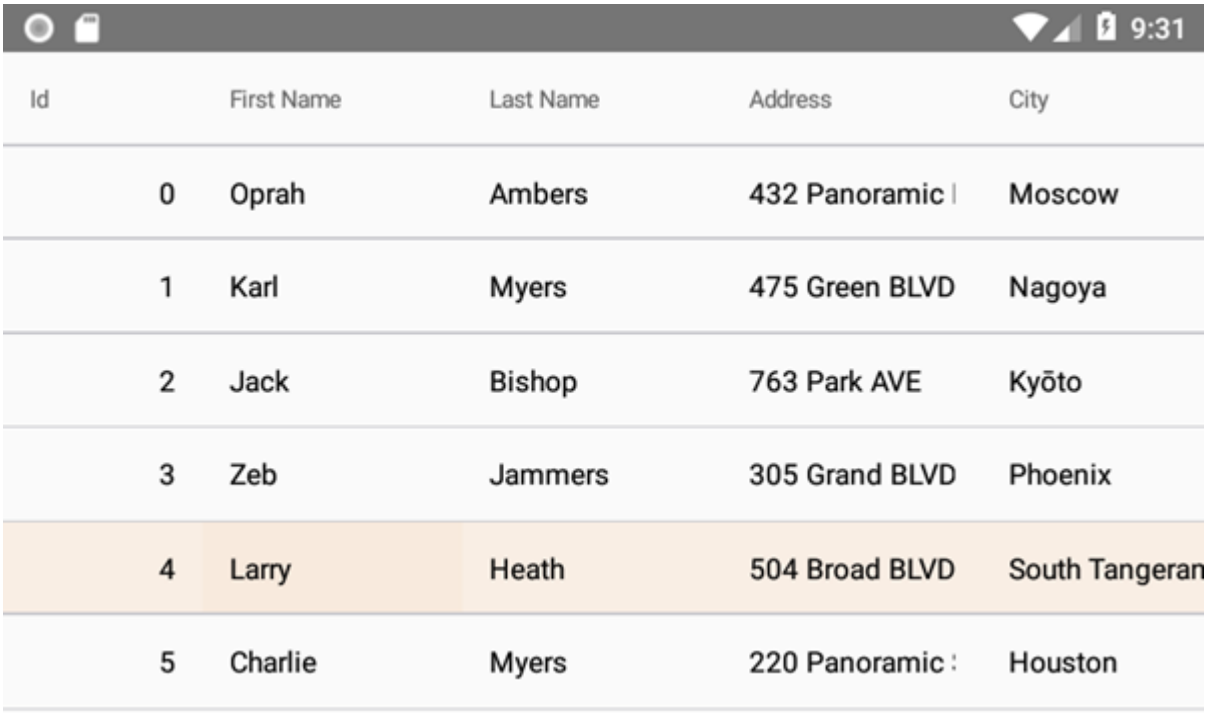
3. In the Solution Explorer, expand the Grouping.xaml node and open the Grouping.xaml.cs to view the C# code.
4. Add the following code to apply grouping to the column Country in the FlexGrid control.

C#

```
public partial class Grouping : ContentPage
{
    C1CollectionView<Customer> _collectionView;
    public Grouping()
    {
        InitializeComponent();
        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;
    }
}
```

Material Theme

From 2018v3 onwards, the default appearance of FlexGrid has changed to the material design pattern. So, for users using 2018v3 or beyond, FlexGrid appears as follows:



In order to revert to the classic style which was there until 2018v2, use the following code:

XAML

```
XML
Style="{x:Static c1:FlexGrid.ClassicStyle}"
```

C#

```
C#
grid.Style = FlexGrid.ClassicStyle;
```

We have also added following features to support the material design in the FlexGrid control:

- **Checklist behavior**
  To further support the material design, you can now enable the checklist like selection. This kind of selection lets you select the non-consecutive rows of data. You can enable it by using the following code:

  XAML

  ```
  XML
  <c1:FlexGrid x:Name="grid">
          <c1:FlexGrid.Behaviors>
              <c1:CheckListBehavior SelectionBinding="Selected"/>
          </c1:FlexGrid.Behaviors>
  </c1:FlexGrid>
  ```

  C#

  ```
  C#
  var details = new CheckListBehavior();
  ```

```
details.SelectionBinding = "Selected";
details.Attach(grid);
```

- **Header grid lines**
  Also, to provide more garnular styling capabilities suited for Material design, FlexGrid supports RowHeaderGridLinesVisibility, ColumnHeaderGridLinesVisibility, and TopLeftHeaderGridLinesVisibility properties to display or hide the individual header grid lines.

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.

	ID	Name	Country	Country ID	Active
	0	Larry Krause	United States	3	☐
	1	Charlie Frommer	Brazil	0	☐
	2	Steve Neiman		0	☑
	3	Oprah Krause		0	☐
	4	Steve Frommer	United States	3	☑
	5	Xavier Orsted		3	☑
	6	Steve Jammers	Congo	1	☐
	7	Gil Krause	Brazil	0	☑
	8	Gil Orsted	Egypt	2	☐
	9	Steve Krause	Brazil	0	☑
	10	Larry Neiman	Congo	1	☐
	11	Herb Orsted	Brazil	0	☑

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 Content 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:c1="clr-
namespace:C1.Xamarin.Forms.Grid;assembly=C1.Xamarin.Forms.Grid"
     x:Class="CellMerging.Merging" x:Name="page">
  <Grid RowSpacing="0">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition />
    </Grid.RowDefinitions>
    <c1: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#

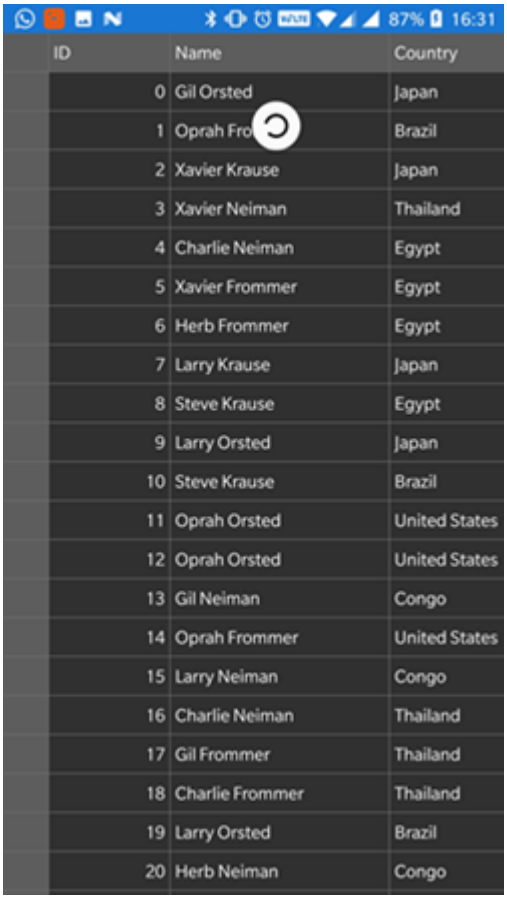
```
public partial class Merging : ContentPage
    {
        public Merging()
        {
            InitializeComponent();
            var data = Customer.GetCustomerList(100);
            grid.ItemsSource = data;
            grid.Columns["Country"].AllowMerging = true;
        }
    }
```

Pull To Refresh

Pull-to-refresh is a touch screen gesture that consists of touching the screen of a handheld device with a finger, dragging the screen downward with the finger, and then releasing it, to refresh the contents of the screen. This feature is useful when the grid contains items, which may change after the initial load.

FlexGrid allows you to enable the PullToRefresh feature by setting the **AllowRefreshing** to true and creating a new class that implements the **C1CursorCollectionView** to simulate the dynamic data. In this sample, we have set the **ItemSource** property of the grid to **MyCollectionView()** method that defines **PageSize** and no of items to be displayed in the grid.

The below image shows how you can use pull down gesture to refresh the data on the page.



The following code examples demonstrate how to set this property in C#. This example uses **Customer.cs** data model created in the Quick Start section.

In Code

Add the following code in the **QuickStart.cs** file to display line marker for FlexChart control.

C#

```
class QuickStart
    {
        public static FlexGrid GetGridControl()
        {
            //Create an instance of the Control and set its properties
            FlexGrid grid = new FlexGrid();
            grid.ItemsSource = new MyCollectionView();
            grid.AllowRefreshing = true;
            return grid;
        }
    }
```

MyCollectionView.cs

C#

```
public class MyCollectionView : C1CursorCollectionView<Customer>
    {
```

```
        ObservableCollection<Customer> items;
        public MyCollectionView()
        {
            PageSize = 50;
            items = Customer.GetCustomerList(300);
        }

        public int PageSize { get; set; }

        protected override async Task<Tuple<string, IReadOnlyList<Customer>>>
GetPageAsync(int startingIndex, string pageToken, int? count = null,
CancellationToken cancellationToken = default(CancellationToken))
        {
            return await Task.Run(() =>
            {
                var moreItems = new ObservableCollection<Customer>
(items.Skip(startingIndex).Take(PageSize));
                return new Tuple<string, IReadOnlyList<Customer>>("Token not used",
moreItems);
            });
        }
    }
```

Resizing Columns/Rows

FlexGrid's AllowResizing property allows a user to resize columns/rows by simply touching and dragging the handle between two columns/rows. This property accepts the values from the GridAllowResizing enum that specifies a value to define which elements can be resized.

You can set the **AllowResizing** property of FlexGrid to **None** in case you want to restrict resizing at runtime.

In Code

C#

```
grid.AllowResizing = GridAllowResizing.None;
```

In XAML

C#

```
<c1:FlexGrid x:Name="grid"  AutoGenerateColumns="False" AllowResizing="None">
```

The resizing functionality for columns/rows is useful when a user wants to add data in FlexGrid. The user can simply resize the column/row 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.

In Code

C#

```
grid.AllowResizing = GridAllowResizing.Columns;
```

In XAML

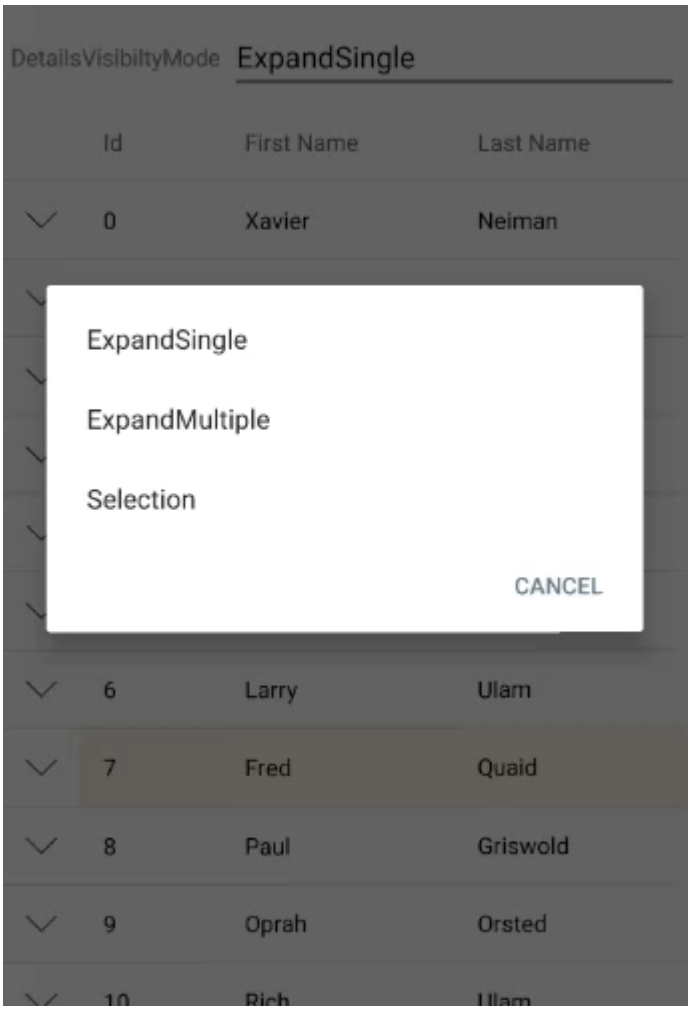
C#

```
<c1:FlexGrid x:Name="grid"  AutoGenerateColumns="False" AllowResizing="Columns">
```

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. Moreover, you can set the details visibility mode to expand single, expand multiple or selection, with the help of DetailVisibiltyMode property provided by the FlexGrid class.

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



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,Ōsaka,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 Content 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:c1="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>
    <c1:FlexGrid x:Name="grid" Grid.Row="3" AutoGenerateColumns="False">
      <c1:FlexGrid.Columns>
        <c1:GridColumn Binding="Id" Width="Auto"/>
        <c1:GridColumn Binding="FirstName" Width="*"/>
        <c1:GridColumn Binding="LastName" Width="*"/>
      </c1:FlexGrid.Columns>
      <c1:FlexGrid.Behaviors>
        <c1: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>
        </c1:FlexGridDetailProvider>
      </c1:FlexGrid.Behaviors>
    </c1: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 replacing their icons with images using 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

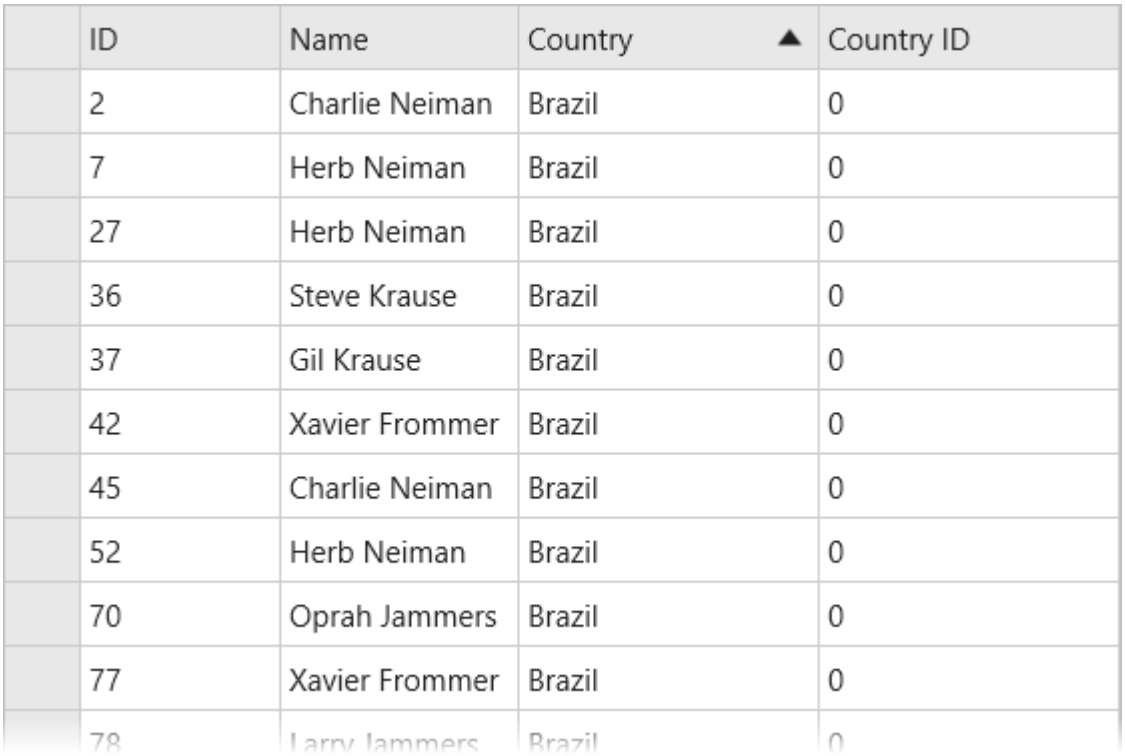
FlexGrid allows you to sort grid's data using the following methods:

- **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 ▲	Country ID
	2	Charlie Neiman	Brazil	0
	7	Herb Neiman	Brazil	0
	27	Herb Neiman	Brazil	0
	36	Steve Krause	Brazil	0
	37	Gil Krause	Brazil	0
	42	Xavier Frommer	Brazil	0
	45	Charlie Neiman	Brazil	0
	52	Herb Neiman	Brazil	0
	70	Oprah Jammers	Brazil	0
	77	Xavier Frommer	Brazil	0
	78	Larry Jammers	Brazil	0

C#

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

In XAML

XAML

```
<c1: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	Country ID
	9	Charlie Neiman	Congo	1
	1	Herb Jammers	Egypt	2
	8	Larry Krause	Congo	1
	2	Oprah Orsted	Egypt	2
	5	Oprah Orsted	Egypt	2
	4	Steve Frommer	Egypt	2
	6	Steve Orsted	Congo	1
	7	Xavier Krause	Thailand	5
	0	Xavier Neiman	Congo	1
	3	Xavier Neiman	United States	3

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

In Code

C#

```
public static FlexGrid GetGrid()
       {
         var dataCollection = Customer.GetCustomerList(10);
         C1CollectionView<Customer> cv = new C1CollectionView<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".

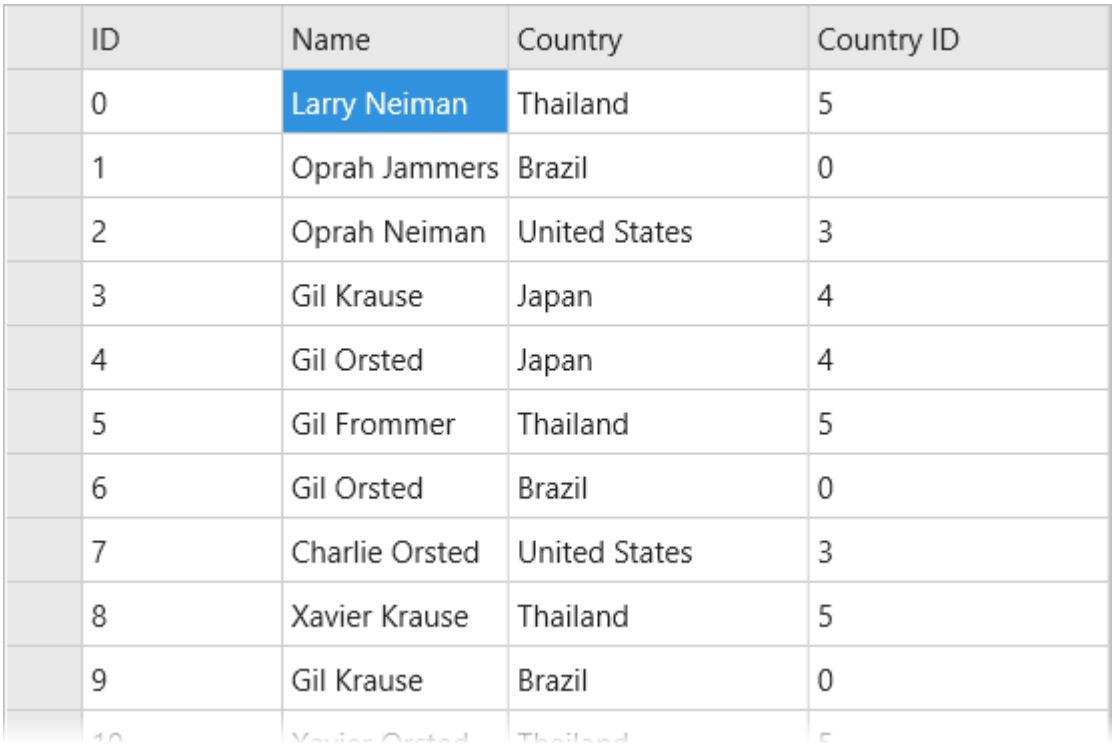
C#

```
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, RowRange, or None**. This property accepts these values from the GridSelectionMode enum.

The image below shows how the FlexGrid appears after the SelectionMode property is set to **Cell**.



	ID	Name	Country	Country ID
	0	Larry Neiman	Thailand	5
	1	Oprah Jammers	Brazil	0
	2	Oprah Neiman	United States	3
	3	Gil Krause	Japan	4
	4	Gil Orsted	Japan	4
	5	Gil Frommer	Thailand	5
	6	Gil Orsted	Brazil	0
	7	Charlie Orsted	United States	3
	8	Xavier Krause	Thailand	5
	9	Gil Krause	Brazil	0

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.Cell;
```

In XAML

XAML

```
<Grid VerticalOptions="FillAndExpand">
```

```
  <c1:FlexGrid SelectionMode="Cell" x:Name="grid" />
</Grid>
```

Selection Menu

The selection menu contains common actions such as, editing, selecting, and deleting text. In FlexGrid, selection menu is enabled by default. However, you can disable it by setting the **ShowSelectionMenu** property to false. In desktop applications, you can activate the selection menu with a right click on a cell. In mobile applications, you can activate selection menu with a long press on a cell or by tapping the row header. You can also add custom actions to the selection menu by setting a handler for **CreateSelectionMenu** event.

	ID	Name	Country	Country ID	Active	First	Last
	0	Charlie Jammers	Japan	4	☑	Charlie	Jammers
	1	Xavier Orsted	Congo	1	☑	Xavier	Orsted
	2	Gil Jammers	Congo	1	☑	Gil	Jammers
		Gil Orsted	Thailand	5	☐	Gil	Orsted
		Oprah Frommer	Japan	4	☐	Oprah	Frommer
		Steve Jammers	United States	3	☑	Steve	Jammers
		Xavier Orsted	Brazil	0	☑	Xavier	Orsted
	7	Herb Jammers	Egypt	2	☐	Herb	Jammers
	8	Herb Krause	Brazil	0	☑	Herb	Krause
	9	Herb Krause	Thailand	5	☑	Herb	Krause

Cut
Copy
Paste
Delete

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	Name	Country	Country ID
	0	Larry Orsted	Brazil	0
	1	Steve Neiman	Japan	4
	2	Oprah Jammers	Japan	4
	3	Steve Orsted	Congo	1
	4	Xavier Orsted	Japan	4
	5	Gil Neiman	Japan	4
	6	Herb Neiman	Japan	4
	7	Steve Frommer	Brazil	0
	8	Charlie Jammers	Congo	1
	9	Gil Frommer	United States	3

The following code example demonstrates star sizing in FlexGrid in XAML. The example uses the sample created in the Quick Start section.

In XAML

XAML

```
<Grid>
    <c1:FlexGrid x:Name="grid" AutoGenerateColumns="False">
      <c1:FlexGrid.Columns>
        <c1:GridColumn Binding="ID" Width="*"/>
        <c1:GridColumn Binding="Name" Width="Auto"/>
        <c1:GridColumn Binding="Country" Width="*"/>
        <c1:GridColumn Binding="CountryID" Width="Auto"/>
      </c1:FlexGrid.Columns>
    </c1: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.

In Code

C#

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

In XAML

C#

```
<c1:GridColumn Binding="Name" Width="Auto" WordWrap="True"/>
```

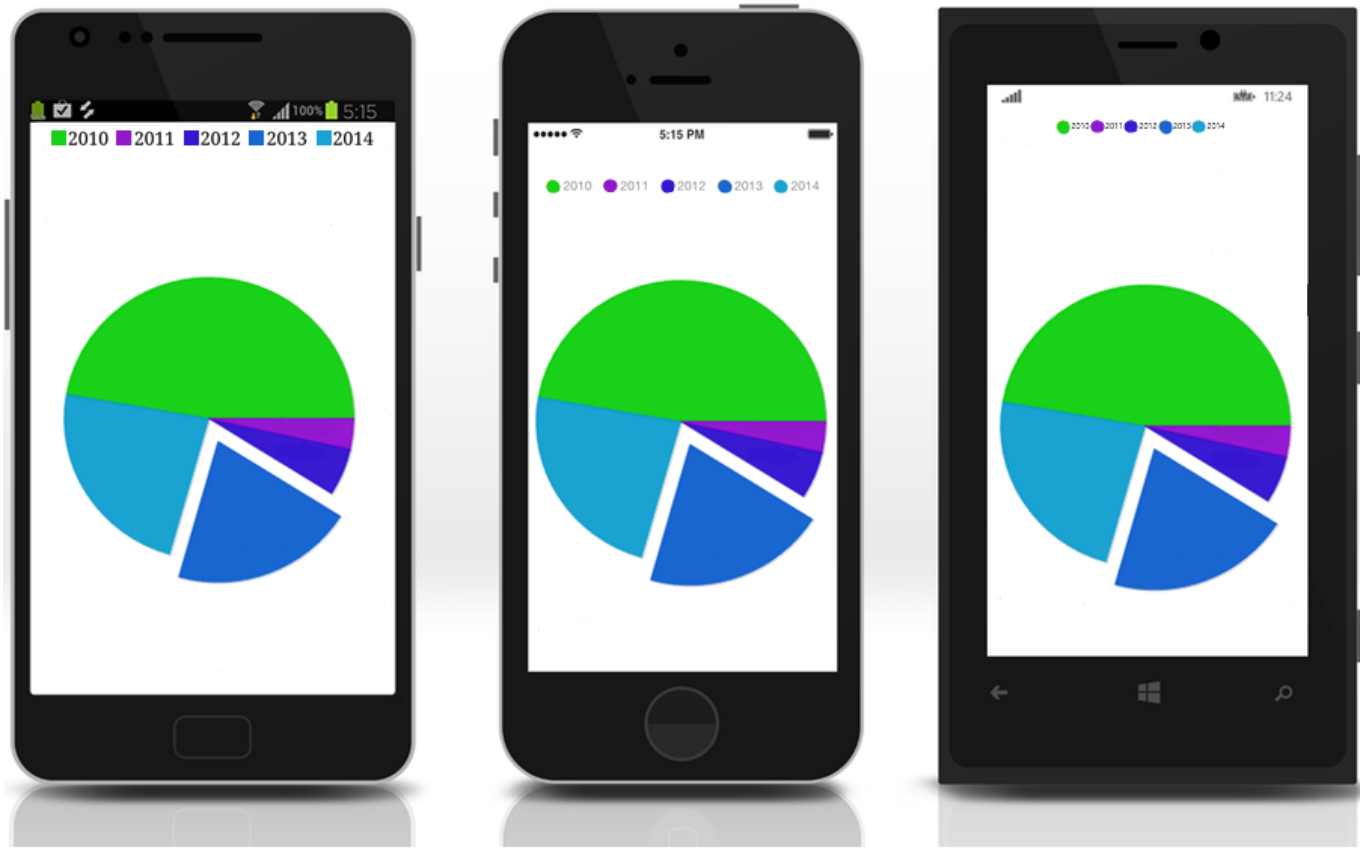
To resize the row heights automatically, you can then call the AutoSizeRow and AutoSizeRows methods as illustrated in the code below.

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

- **Animation**: Use Animation to add effects to your FlexPie.
- **Scrolling & Zooming**: Enable scrolling and touch based zooming for your pie charts, for a better view of bigger charts on a smaller screen.
- **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.

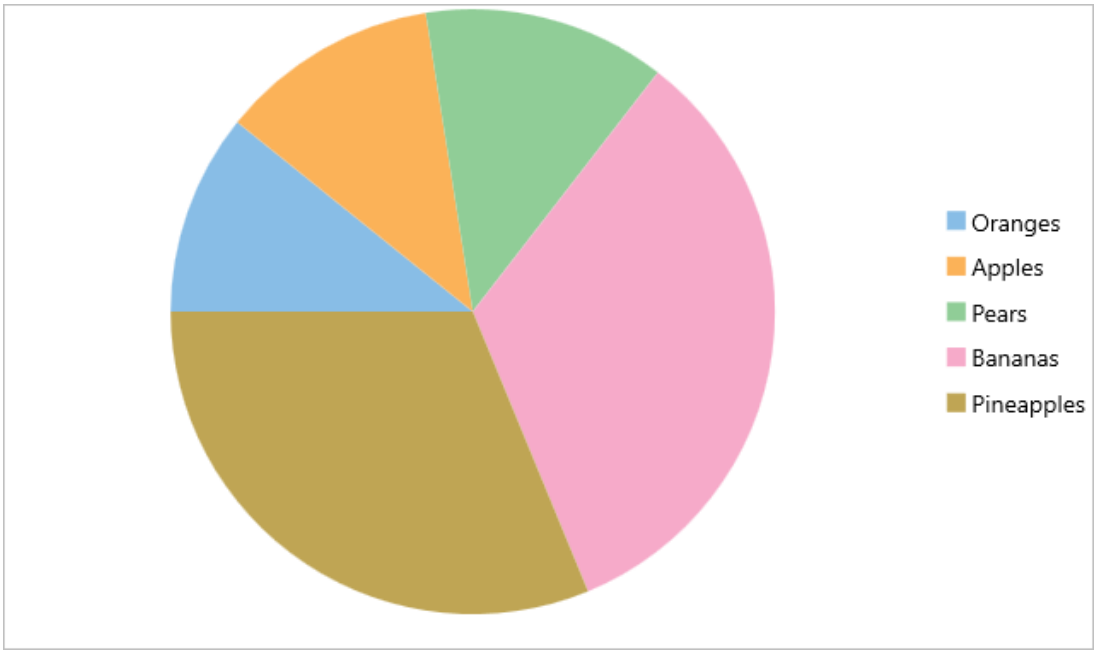
Quick Start: Add data to FlexPie

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 C1.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 Content 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:c1="clr-namespace:C1.Xamarin.Forms.Chart;assembly=C1.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>
       <c1:FlexPie x:Name="chart" ItemsSource="{Binding Data}" BindingName="Name"
                   Binding="Value" Grid.Row="1" Grid.ColumnSpan="2"
   ```

```
VerticalOptions="FillAndExpand">
  </c1: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 Content 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#

```
C1.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#

```
C1.Xamarin.Forms.Chart.Platform.UWP.FlexPieRenderer.Init();
```

4. (Optional) In case you compile your UWP application in **Release** mode, you need to explicitly add the following code to the **OnLaunched** method in your **App.xaml.cs** to include the correct assemblies within your application.

```
C#
var assembliesToInclude = new List<Assembly>();
assembliesToInclude.Add(typeof(C1.Xamarin.Forms.Chart.Platform.UWP.FlexChartRenderer)
.GetTypeInfo().Assembly);
assembliesToInclude.Add(typeof(C1.UWP.Chart.FlexChart).GetTypeInfo().Assembly);
Xamarin.Forms.Forms.Init(e, assembliesToInclude);
```

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

Back to Top

Features

Animation

FlexPie allows you to enable animation effects using one of the two ways, either on loading when the chart is drawn or on updating when the chart is redrawn after modifications. It supports animation in charts through C1Animation class available in the C1.Xamarin.Forms.Core namespace.

The following GIF shows animation in FlexPie.

You can also set the duration of animation in chart using Duration property and interpolate the values of animation using Easing property of the C1Animation class, which accepts values from the C1Easing class. This class supports a collection of standard easing functions such as CircleIn, CircleOut, and Linear.

- **CircleIn:** Easing function that starts slow and speeds up in the form of a circle.
- **CircleOut:** Easing function that starts fast and slows down in the form of a circle.

- **Linear:** Easing function with constant speed.

```
C#
C1Animation animate = new C1Animation();
// set update animation duration
animate.Duration = new TimeSpan(1500 * 10000);
// interpolate the values of animation
animate.Easing = C1Easing.Linear;
```

In addition to easing functions of the C1Easing class, FlexPie supports built in easing functions of Xamarin.Forms.Easing class. For more information, refer Xamarin Easing Class.

You can show animation while loading or updating a chart. To show animation while loading the chart, use LoadAnimation property of the ChartBase class, which gets the load animation from the object of C1Animation class. Similarly, to animate the chart when underlying data collection changes on adding, removing, or modifying a value, you can use UpdateAnimation property of the ChartBase class.

```
C#
// set the loading animation easing
chart.LoadAnimation = animate;
```

You can apply animation effect by setting the AnimationMode property which accepts values from the AnimationMode enumeration. This enumeration supports four different animation modes: All, None, Series, and Point.

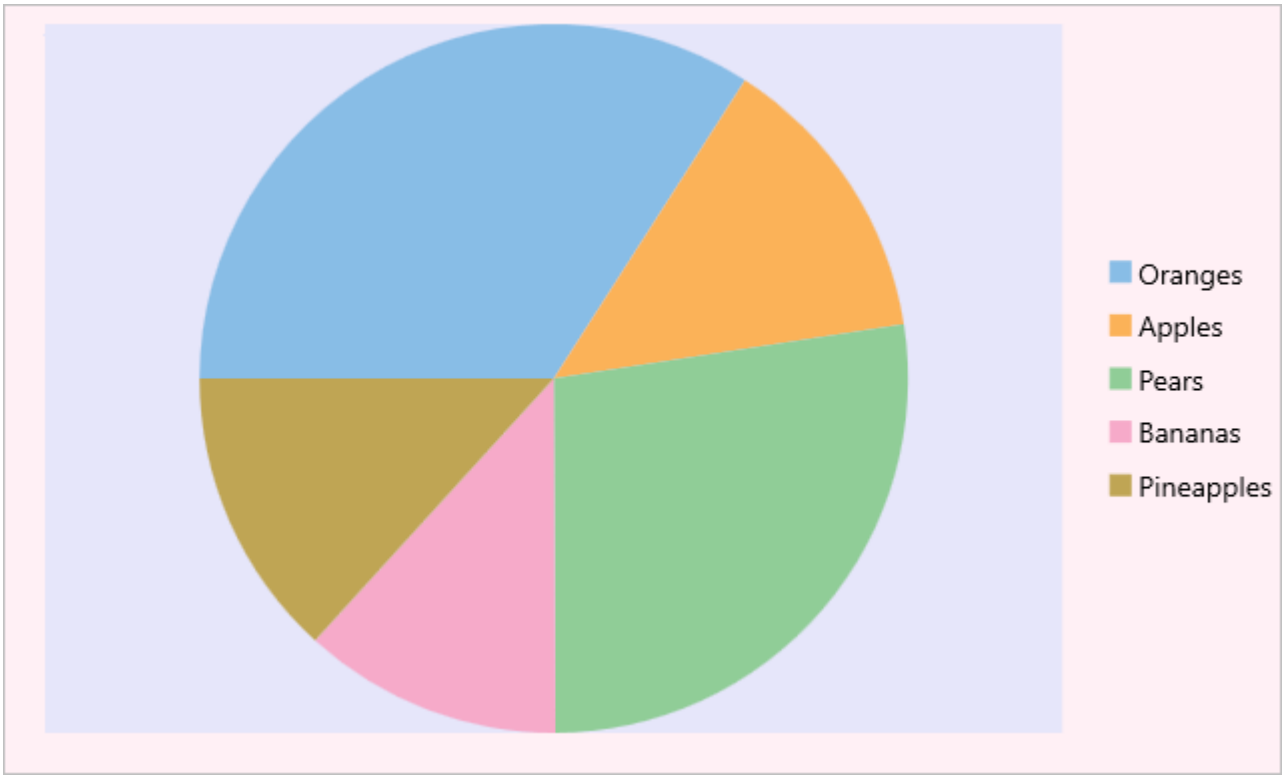
- **All**: The chart expands outward from the center point.
- **None**: Does not display any animation.
- **Series**: The chart expands outward from the center point.
- **Point**: The chart radially animates clockwise around the center point.

```
C#
// set the animation mode
chart.AnimationMode = AnimationMode.Point;
```

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# and XAML. This example uses the sample created in the Quick Start section.

C#

```
//Customize chart plot area
ChartStyle s = new ChartStyle();
s.Fill = Color.Lavender;
s.StrokeThickness = 0;
chart.PlotStyle = s;

//Set background color
chart.BackgroundColor = Color.LavenderBlush;
```

In XAML

XAML

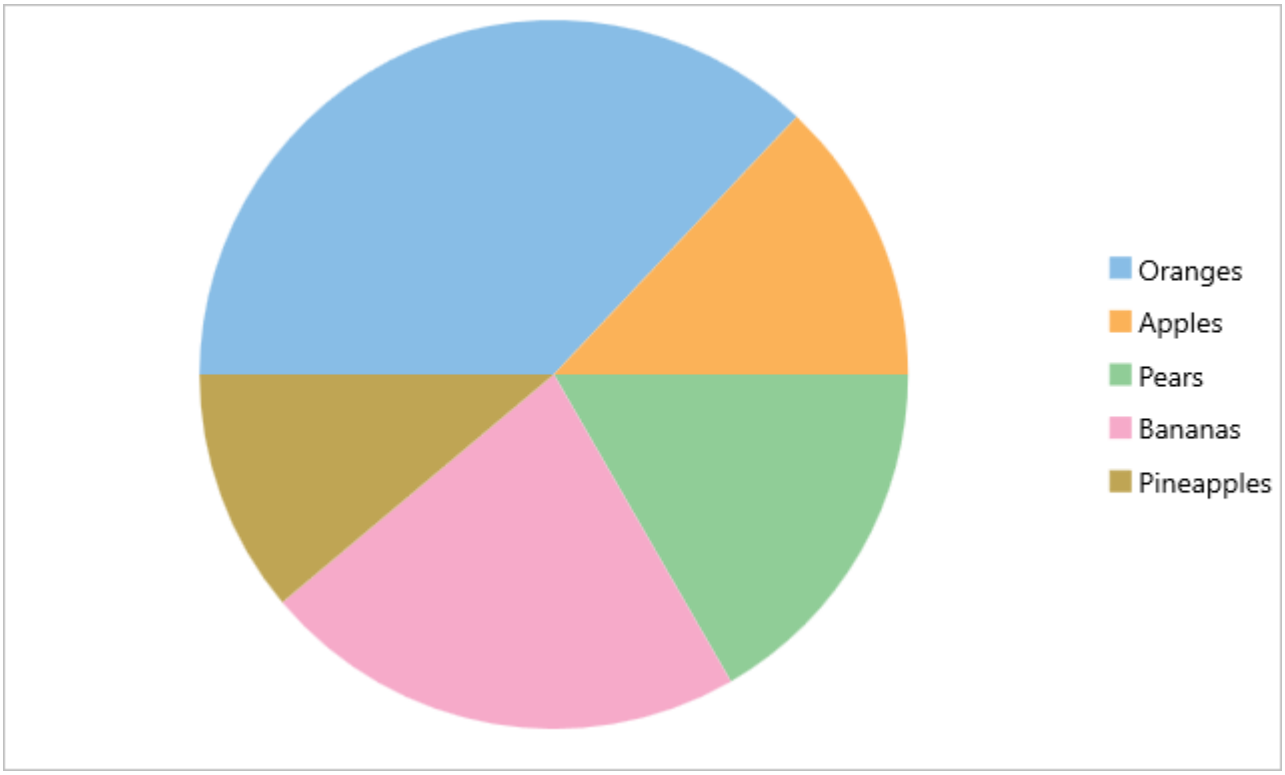
```
<c1:FlexPie x:Name="chart" ItemsSource="{Binding Data}" BindingName="Name"
    Binding="Value" BackgroundColor="LavenderBlush">
    <c1:FlexPie.PlotStyle>
        <c1:ChartStyle Fill="Lavender" StrokeThickness="0"/>
    </c1:FlexPie.PlotStyle>
</c1:FlexPie>
```

Data Binding

You can bind the FlexPie control to data by using the following set of properties for data binding:

Property	Description
Binding	Property for binding values
BindingName	Property for binding with items appearing in the legend
ItemsSource	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.

C#

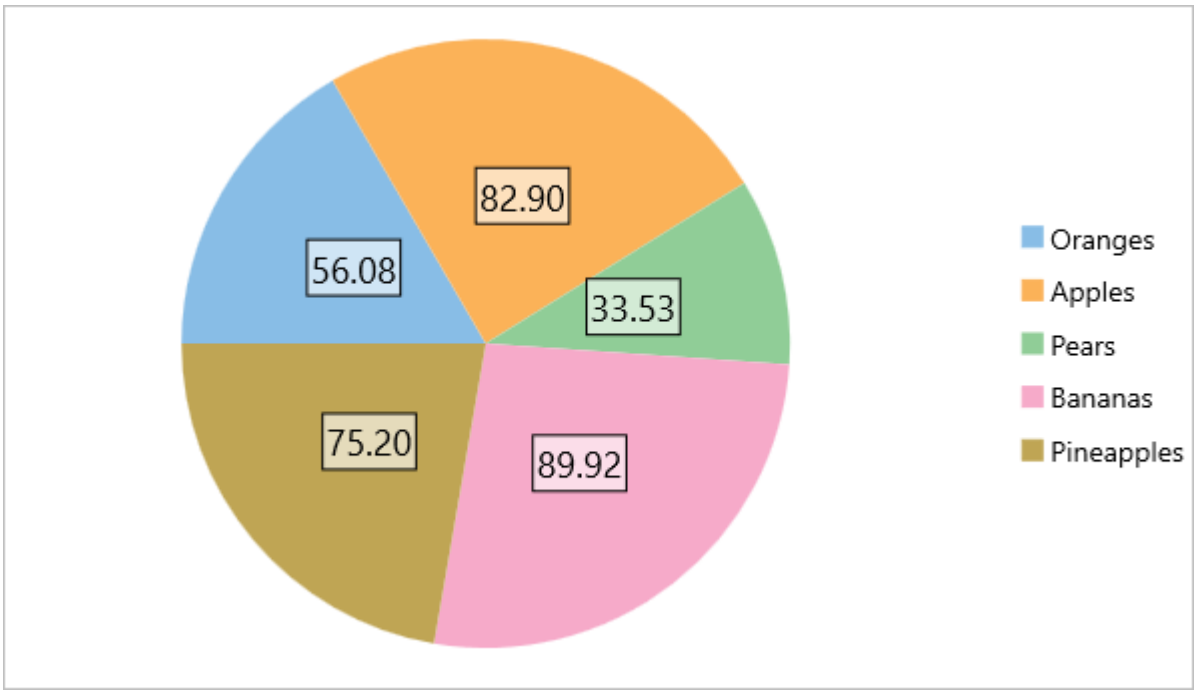
```
chart.BindingName = "Name";
chart.Binding = "Value";
chart.ItemsSource = new Object[]
    {
        new {Value=100, Name="Oranges"},
        new {Value=35, Name="Apples"},
        new {Value=45, Name="Pears"},
        new {Value=60, Name="Bananas"},
        new {Value=30, Name="Pineapples"}
    };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

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:

- **None** - No data labels.
- **Inside**- Displays data labels within the pie.
- **Center**- Displays data labels right in the center of the pie.
- **Outside**- Displays data labels outside the pie.
- **Radial**- Displays data labels inside the pie slice and depends on its angle.
- **Circular** - Displays data labels inside the pie slice in circular direction.

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 Code

To add data label and set its position in FlexPie chart, use the following code in C#.

C#

```
//Adding data labels and setting position
chart.DataLabel.Position = PieLabelPosition.Center;
chart.DataLabel.Border = true;
chart.DataLabel.Content = "{}{value:F2}";
chart.DataLabel.Style.Fill = Color.FromHex("#99FFFFFF");
chart.DataLabel.Style.FontSize = 18;
```

In XAML

To add data label and set its position in FlexPie chart, add the following markup between the <c1:FlexPie> </c1:FlexPie> tags in XAML code.

XAML

```
<c1:FlexPie.DataLabel>
    <c1:PieDataLabel Position="Center" Border="True" Content="{}{value:F2}">
        <c1:PieDataLabel.Style >
            <c1:ChartStyle Fill="#99FFFFFF" FontSize="18" ></c1:ChartStyle>
        </c1:PieDataLabel.Style>
    </c1:PieDataLabel>
</c1:FlexPie.DataLabel>
```

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.

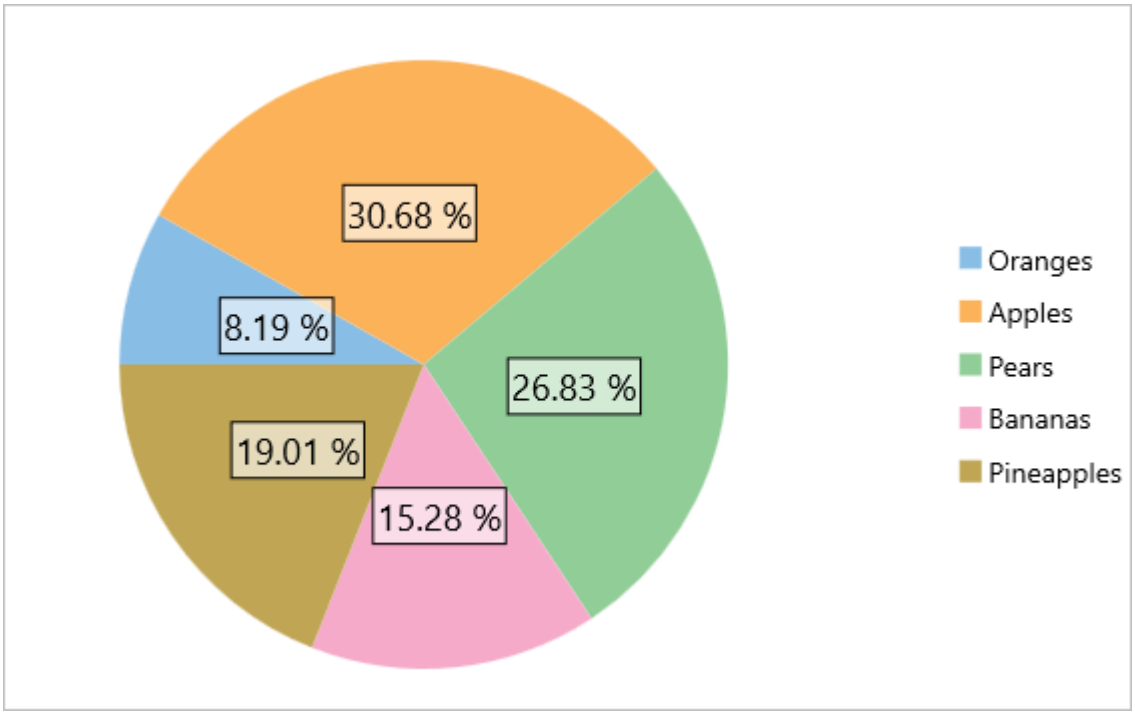
To display percentage values as the data labels, you would bind a Xamarin.Forms Label to 'Percentage' as shown below.

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.



Manage Overlapped Data Labels

A common issue pertaining to charts is overlapping of data labels that represent data points. In most cases, overlapping occurs due to long text in data labels or large numbers of data points.

To manage overlapped data labels in FlexPie chart, you can make use of **Overlapping** property provided by **PieDataLabel** class. The Overlapping property accepts the following values from the **PieLabelOverlapping** enumeration.

Enumeration	Description
PieLabelOverlapping.Default	Show all labels including the overlapping ones.
PieLabelOverlapping.Hide	Hides the overlapping labels, if its content is larger than the corresponding pie segment.
PieLabelOverlapping.Trim	Trim overlapping data labels, if its width is larger than the corresponding pie segment.

Use the following code to manage overlapping data labels.

In Code

C#

```
//Set Overlapping property
flexPie1.DataLabel.Overlapping = PieLabelOverlapping.Trim;
```

In XAML

XAML

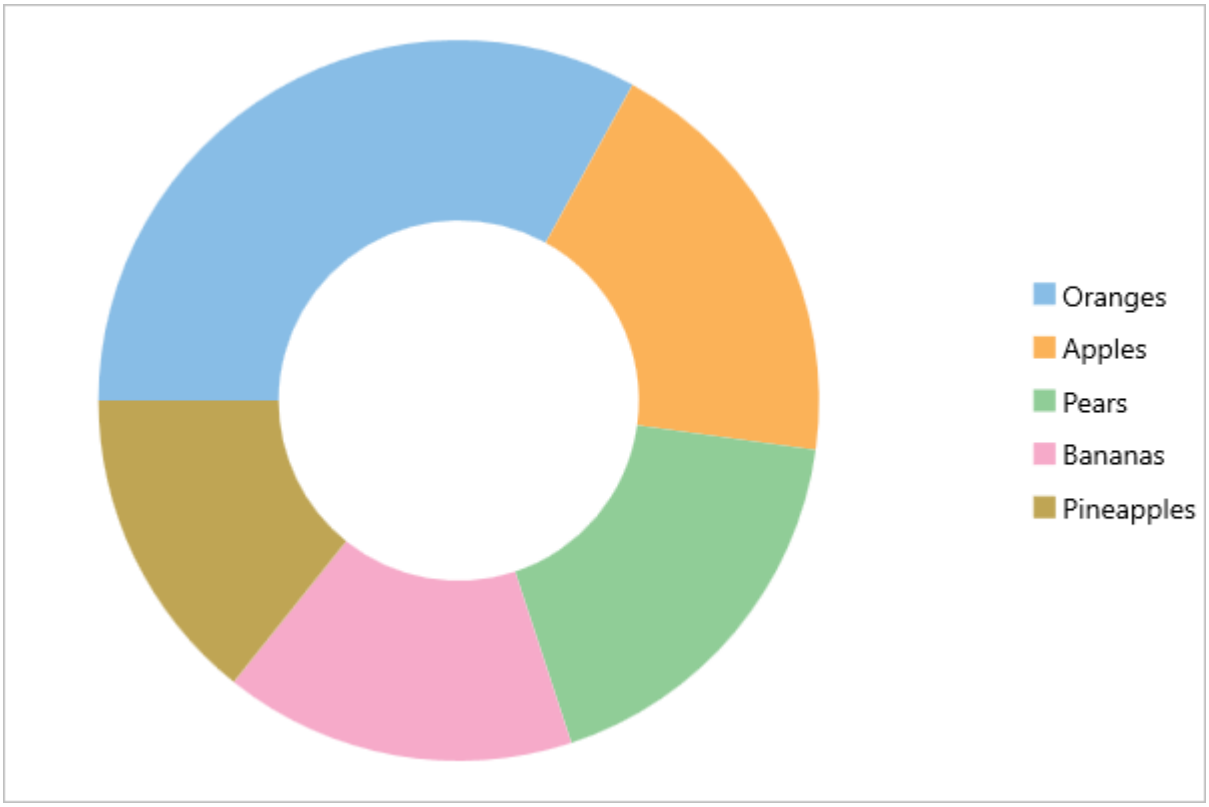
```
<c1:PieDataLabel Position="Center" Content="{}{value:F2}" Overlapping="Trim" >
</c1:PieDataLabel>
```

The following image shows FlexPie appears after setting the **Overlapping** property.

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;
```

In XAML

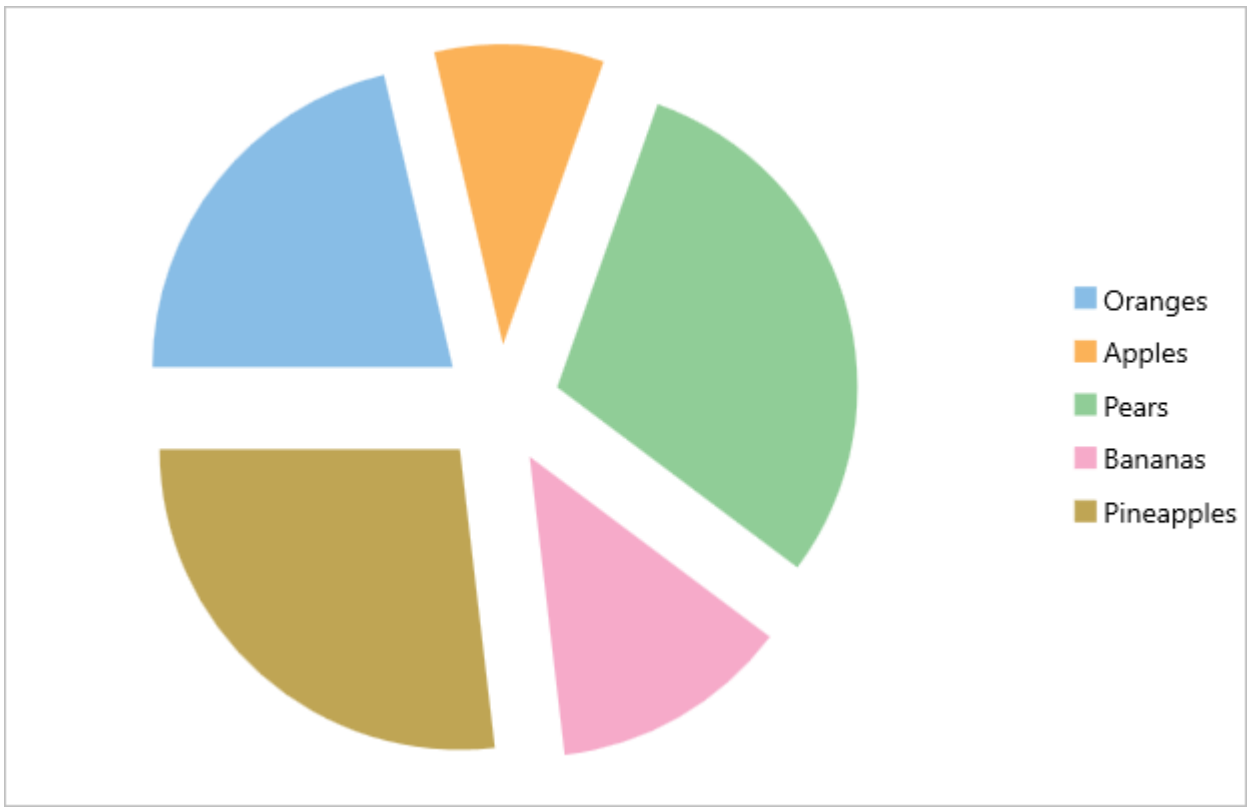
XAML

```
<c1:FlexPie x:Name="chart" ItemsSource="{Binding Data}" BindingName="Name"
    Binding="Value" 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# copyCode

```
chart.Offset = 0.2;
```

In XAML

XAML

```
<c1:FlexPie x:Name="chart" ItemsSource="{Binding Data}" BindingName="Name"
    Binding="Value" Offset="0.2" />
```

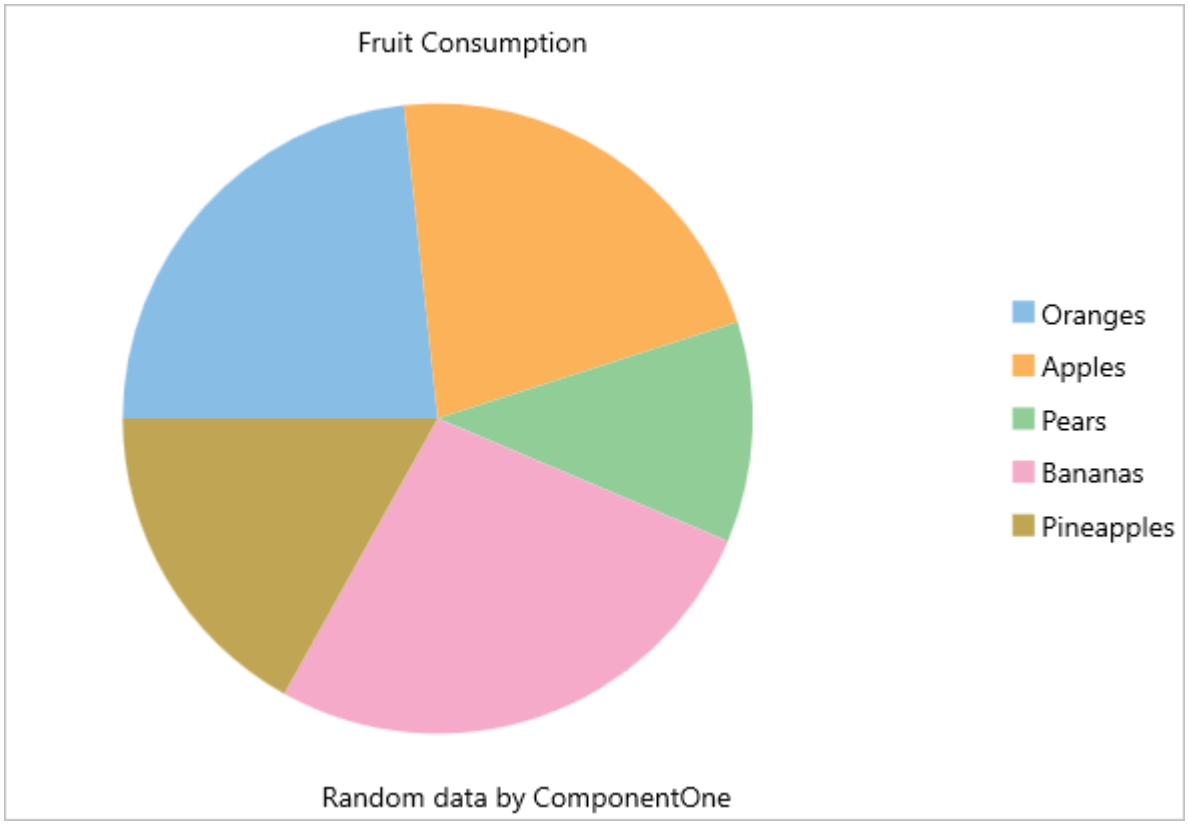
Header and Footer

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.Center;

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

In XAML

The following code example demonstrates how to set these properties in XAML. This example uses the sample created in the Quick Start section.

XAML

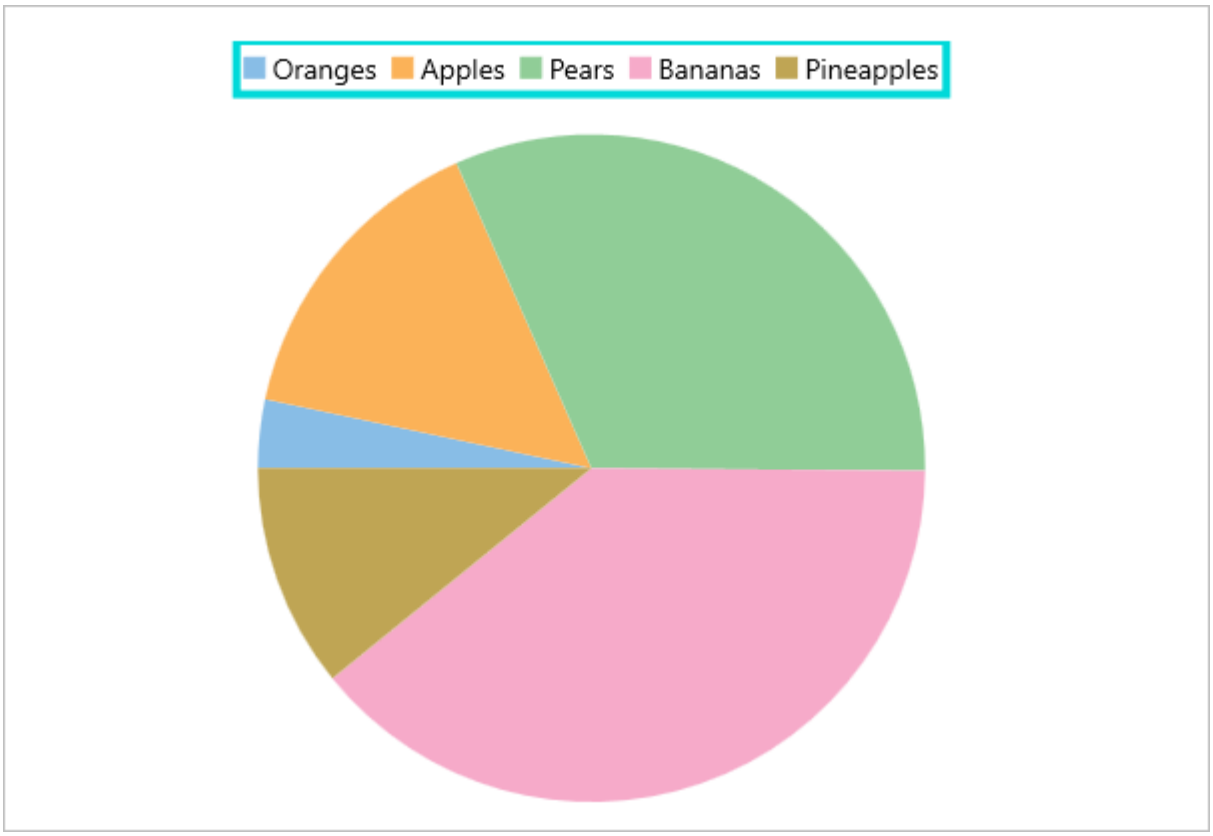
```
<c1:FlexPie x:Name="chart" ItemsSource="{Binding Data}" BindingName="Name"
    Binding="Value" Header="Fruit Consumption" HeaderAlignment="Center"
    Footer="Random data by ComponentOne" FooterAlignment="Center" />
```

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.



The following code example demonstrates how to set these properties in C# and XAML. This examples uses the sample created in the Quick Start section.

In Code

C#

```
chart.LegendPosition = ChartPositionType.Top;
chart.LegendOrientation = Orientation.Horizontal;
chart.LegendStyle.Stroke = Color.FromHex("#00D9D9");
chart.LegendStyle.StrokeThickness = 4;
```

In XAML

XAML

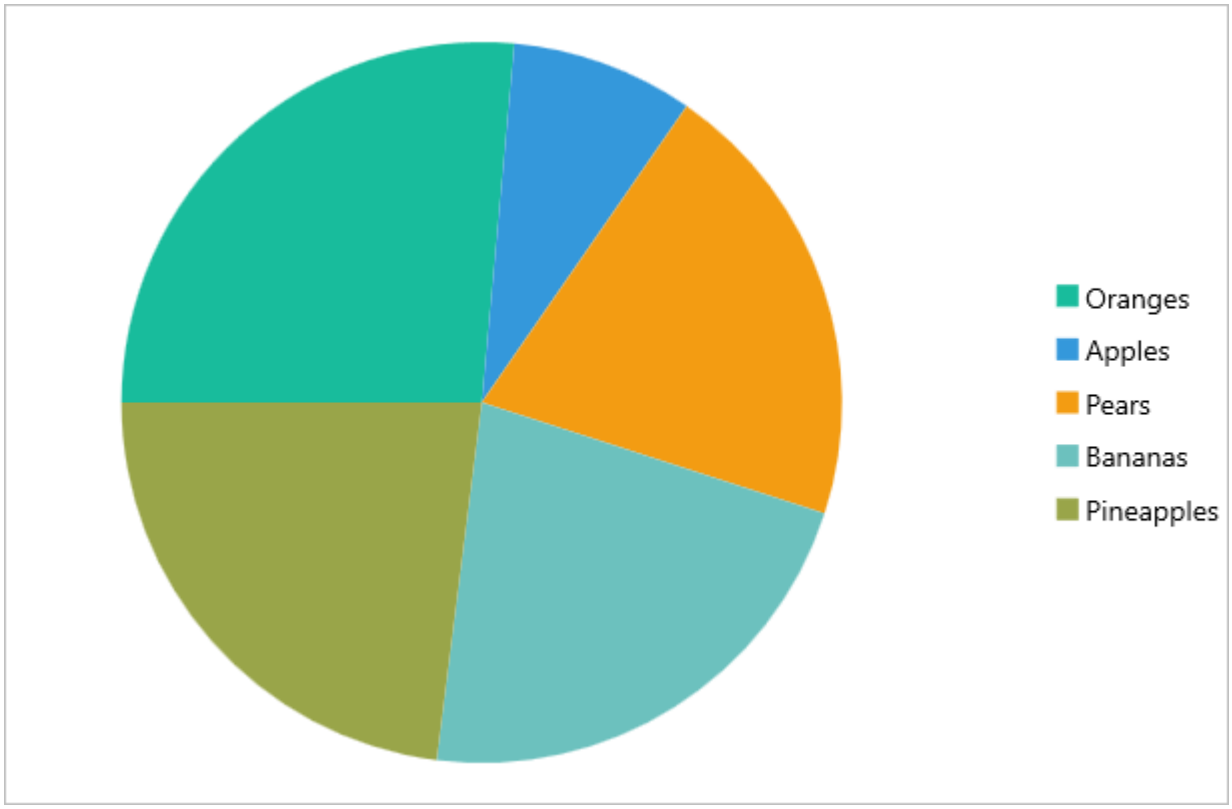
```
<c1:FlexPie x:Name="chart" ItemsSource="{Binding Data}" BindingName="Name"
    Binding="Value" LegendPosition="Top" LegendOrientation="Horizontal">
    <c1:FlexPie.LegendStyle>
        <c1:ChartStyle Stroke="#00D9D9" StrokeThickness="4"></c1:ChartStyle>
    </c1:FlexPie.LegendStyle>
</c1:FlexPie>
```

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# and XAML. This example uses the sample created in the Quick Start section.

In Code

C#

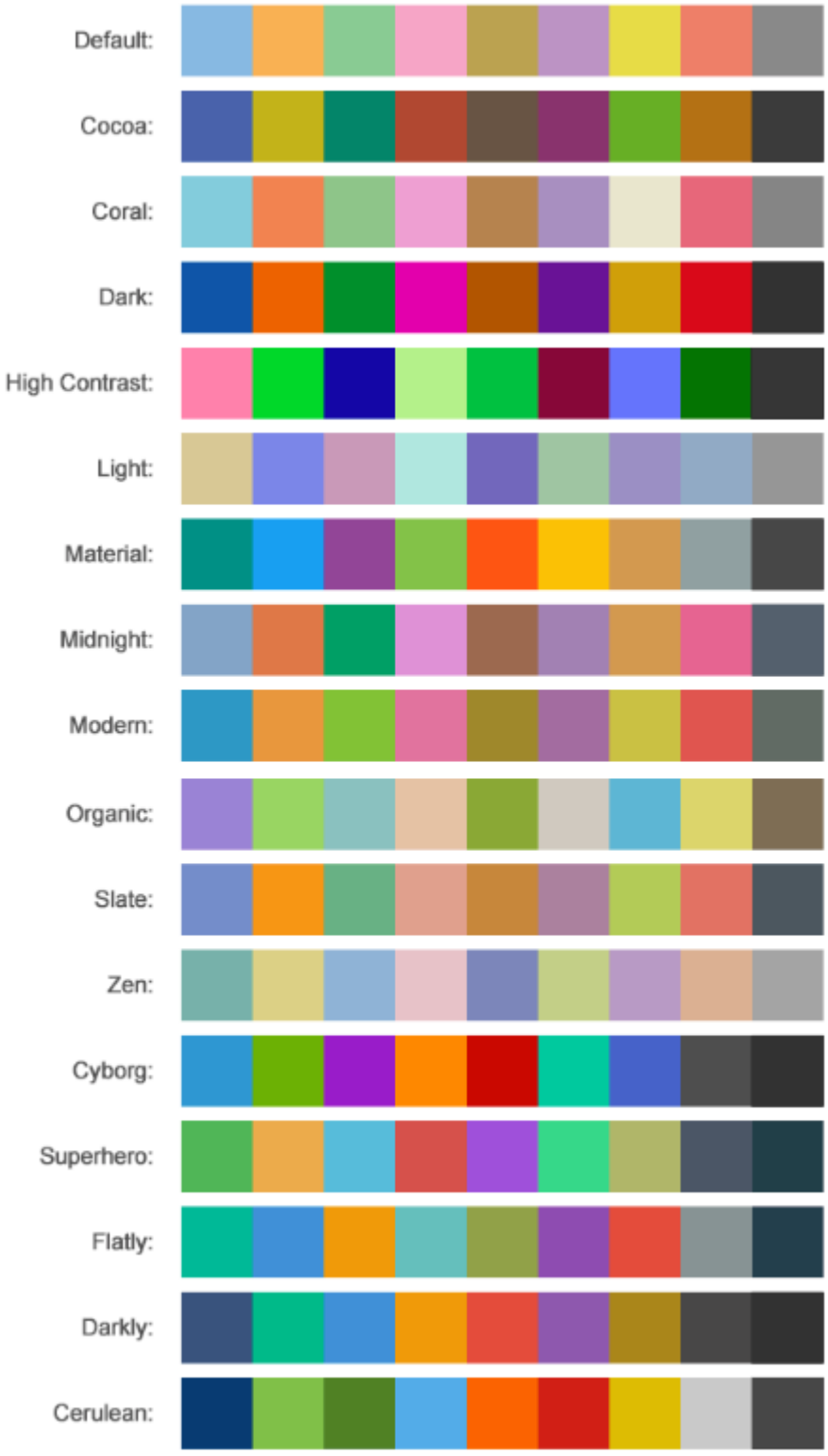
```
//setting themes
 chart.Palette = Palette.Flatly;
```

In XAML

XAML

```
<c1:FlexPie x:Name="chart" ItemsSource="{Binding Data}" BindingName="Name"
    Binding="Value" Palette="Flatly">
```

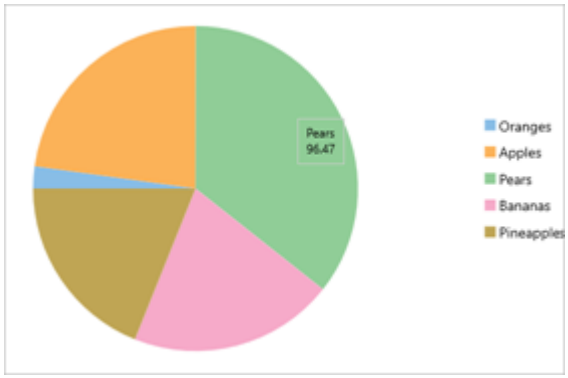
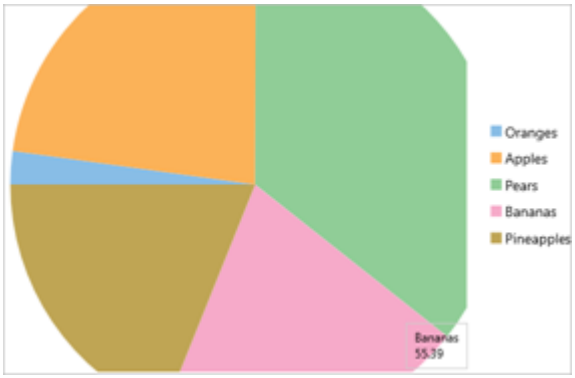
FlexPie comes with pre-defined templates that can be applied for quick customization. Here are the pre-defined templates available in the Palette enumeration.



Zooming and Panning

Zooming can be performed in FlexPie using ZoomBehavior class. To implement zooming, you need to create an object of the ZoomBehavior class available in the C1.Xamarin.Forms.Chart.Interaction namespace and pass it as a parameter to the Add method. This method adds zoom behavior to the behavior collection by accessing it through Behaviors property of the ChartBase class.

The image below shows how FlexPie appears on zooming.

Before Zooming	After Zooming
	

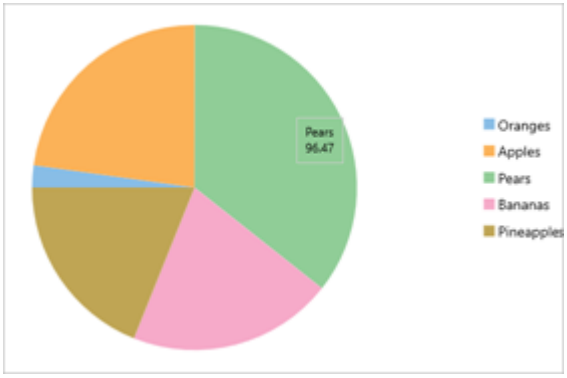
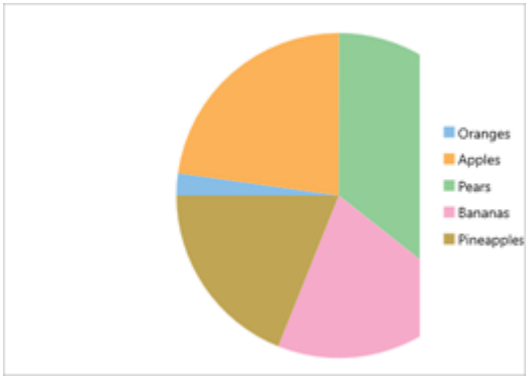
The following code examples demonstrate how to implement zooming in C#. These examples use the sample created in the Quick Start section.

C#

```
ZoomBehavior z = new ZoomBehavior();
chart.Behaviors.Add(z);
```

Similarly, panning can be implemented in the FlexPie chart by creating an object of TranslateBehavior class available in the C1.Xamarin.Forms.Chart.Interaction namespace and passing it as a parameter to the Add method. This method adds the translation behavior to the behavior collection by accessing it through Behaviors property of the ChartBase class. In addition, you can use TranslationX and TranslationY property of the FlexPie class to set the translation x and translation y delta for the chart.

The image below shows how FlexPie appears on panning.

Before Panning	After Panning
	

The following code examples demonstrate how to implement panning in C#. These examples use the sample created in the Quick Start section.

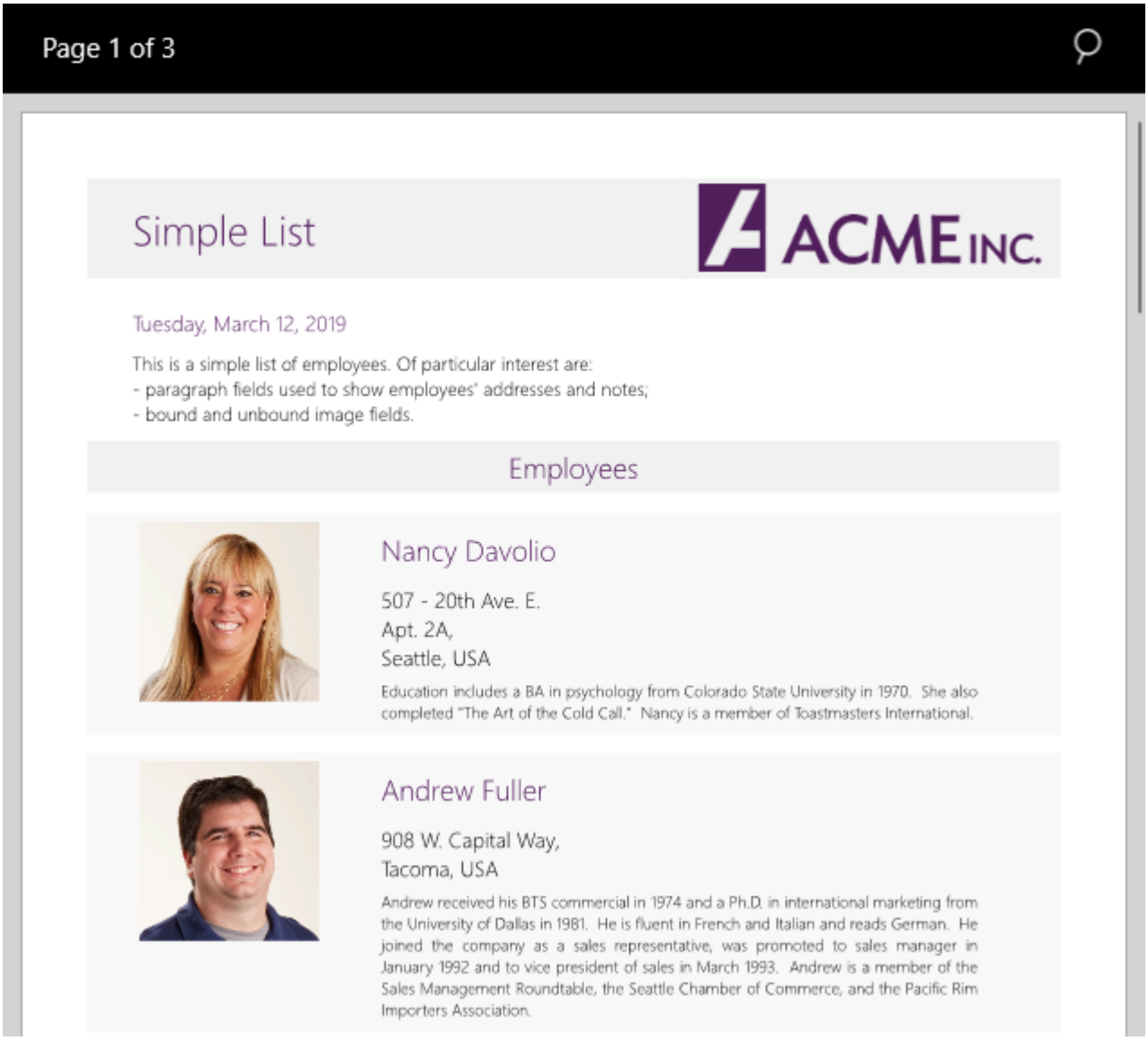
C#

```
TranslateBehavior t = new TranslateBehavior();
chart.Behaviors.Add(t);

chart.TranslationX = 10;
chart.TranslationY = 10;
```

Moreover, you can allow users to translate their custom views when pie is panning or zooming through TranslateCustomViews event handler available in the FlexPie class.

FlexViewer

FlexViewer (Beta), as the name suggests, is a flexible, fast and powerful previewing control which comes with a modern, interactive and user-friendly UI. Currently, it uses GcPdf as its document source. FlexViewer allows you to view PDF documents, navigate through the PDF pages using page navigation option that let's you jump to a specific page by just typing specified page number in the Page textbox. In addition, it allows you to search text in the PDF document and zoom the PDF page from the UI itself, eliminating the need of writing any code.



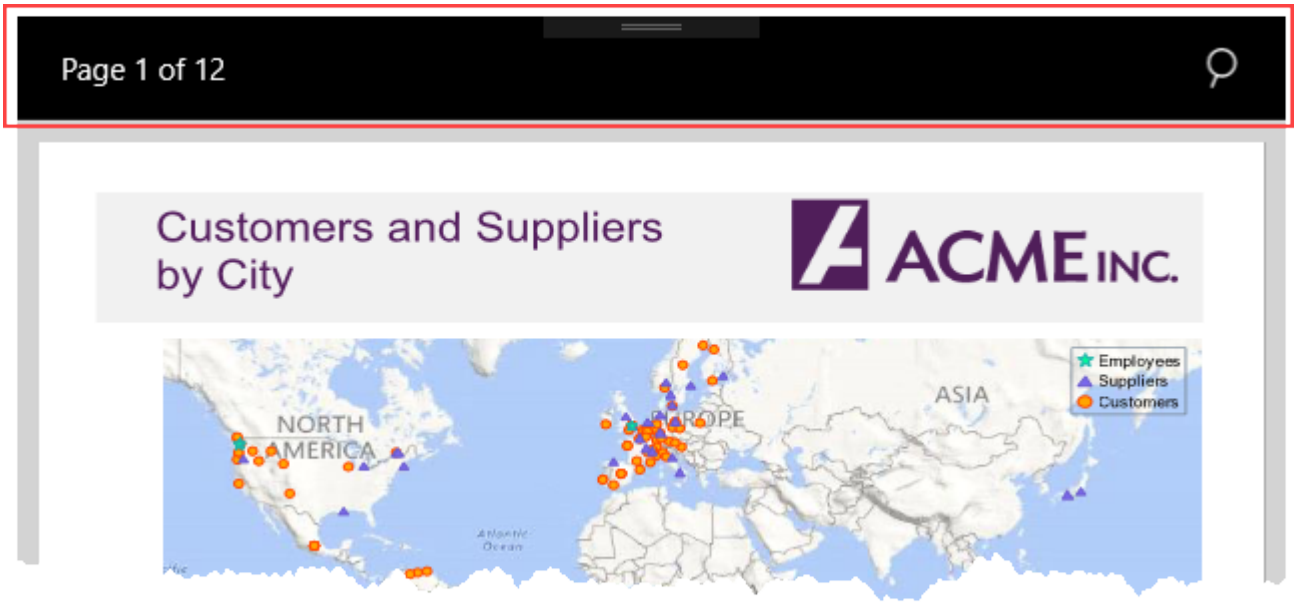
Key Features

The key features of FlexViewer are as follows:

- **Modern user-friendly UI:** Interactive and user friendly UI that helps preview PDF documents with ease.
- **View PDF documents:** Load and display PDF document content, including images and shapes, in the FlexViewer control.
- **Page navigation:** Navigate to a specific page using the page label from the FlexViewer toolbar.
- **Search text:** Find text in document using Search toolbar and get highlighted search results.
- **Export PDF:** Save PDF documents to a file, stream or as images in different formats, such as BMP, JPEG, PNG, GIF, and TIFF.
- **Virtualize UI:** Open large documents without going out of memory.

FlexViewer Toolbar

FlexViewer toolbar appears at the top of the FlexViewer control with two options, a Page label on the left side and search icon on the right. Clicking the Page label transitions it to a textbox allowing you to navigate through the document pages and clicking the search icon opens a search toolbar replacing the main toolbar.



The toolbar consists of the following command buttons:

Command Button	Command Button Name	Description
Page 1 of 12	Page label	Displays the current page number and the total number of pages in the PDF document.
	Search	Allows you to search text in the PDF document.

Quick Start

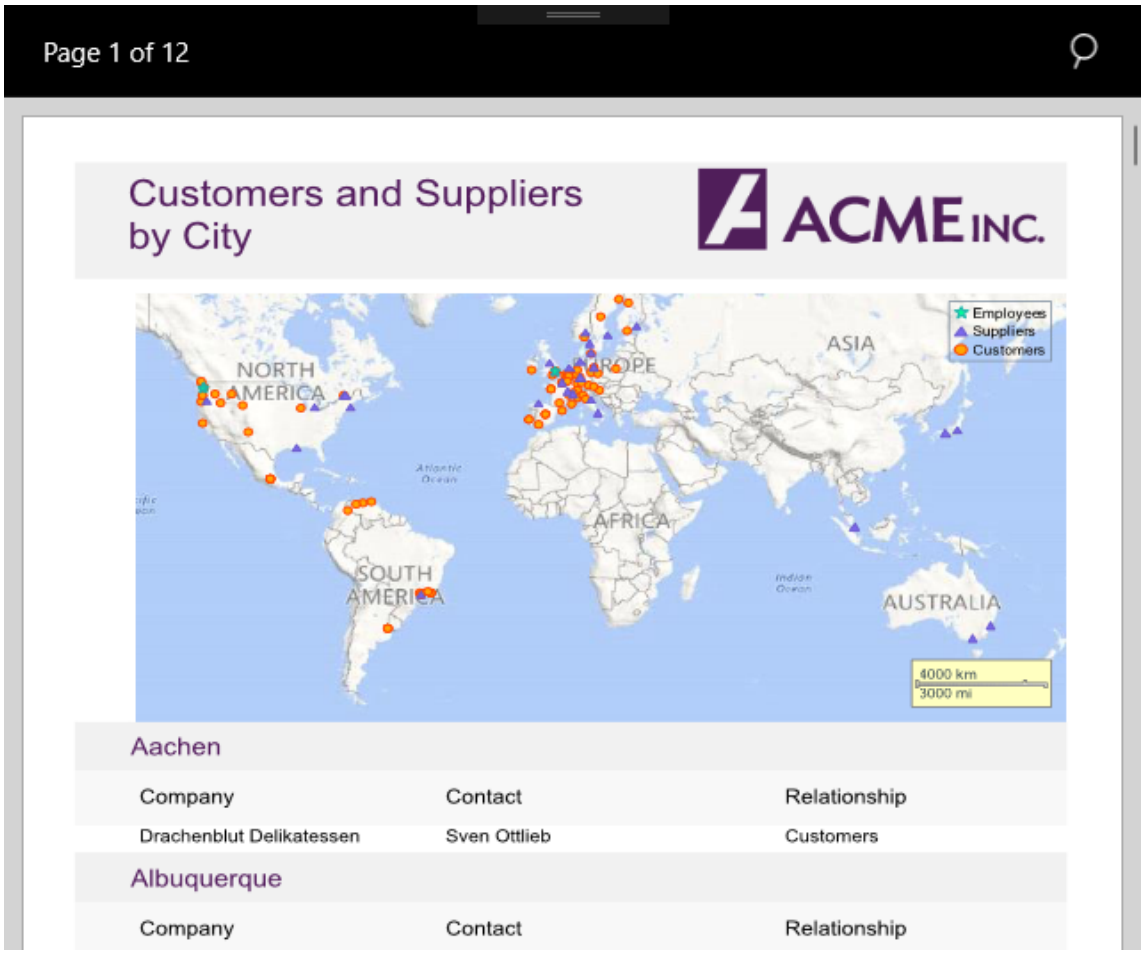
This quick start will guide you through the steps of adding FlexViewer control to the application, binding it with a document source, i.e., GcPdf, and loading the PDF in the FlexViewer control.

Note: To use GcPdf as the document source for your application, you need to install GcPdf NuGet package to add the GcPdf references to your application.

To achieve it, follow these steps:

1. **Add FlexViewer control**
2. **Bind FlexViewer with document source**
3. **Load the PDF document**
4. **Run the Project**

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



Step 1: Add FlexViewer control

1. Add a new content page (Name: `QuickStart.xaml`) to your Mobile App (Xamerin.Forms) and include references as shown below.

XAML

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
     xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
     xmlns:local="clr-namespace:FlexViewerXamarin"
     xmlns:c1="http://schemas.grapecity.com/xf/2019/xaml"
     x:Class="FlexViewerXamarin.QuickStart">
</ContentPage>
```

2. Initialize the FlexViewer control by adding the following markup for the control between the <ContentPage> </ContentPage> tags.

XAML

```
<c1:FlexViewer x:Name="flexViewer"/>
```

Back to Top

Step 2: Bind FlexViewer with document source

1. Add the following references in the **QuickStart.xaml.cs** file.

```
C#
using Xamarin.Forms;
using Xamarin.Forms.Viewer;
```

2. Add the following code to bind the FlexViewer control with the document source. In this sample, we have used GcPdf as the document source.

```
C#                                                                    copyCode
Assembly assembly;

//Bind FlexViewer control to the PDF
assembly = IntrospectionExtensions.GetTypeInfo(typeof(MainPage)).Assembly;
Stream stream = assembly.GetManifestResourceStream("App4.Resources.DefaultDocument.pdf");
```

Back to Top

Step 3: Load the PDF document

Load the PDF document in the FlexViewer control using the following code. In this example, we have added a PDF document to the Resources folder to load it in the viewer.

```
C#                                                                    copyCode
//Load PDF document to the viewer
flexViewer.LoadDocument(stream);
```

Back to Top

Step 4: Run the Project

1. In the **Solution Explorer**, double click App.xaml.cs to open it.
2. Complete the following steps to display the FlexGrid control.
   - **To return a Content 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.Viewer.Platform.iOS.FlexViewerRenderer.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.Viewer.Platform.UWP.FlexViewerRenderer.Init();
```

4. (Optional) In case you compile your UWP application in **Release** mode, you need to explicitly add the following code to the **OnLaunched** method in your **App.xaml.cs** to include the correct assemblies within your application.

```
C#
var assembliesToInclude = new List<Assembly>();
assembliesToInclude.Add(typeof(

C1.Xamarin.Forms.Viewer.Platform.UWP.FlexViewerRenderer).GetTypeInfo().Assembly);
assembliesToInclude.Add(typeof(C1.UWP.Viewer.FlexViewer).GetTypeInfo().Assembly);
Xamarin.Forms.Forms.Init(e, assembliesToInclude);
```

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

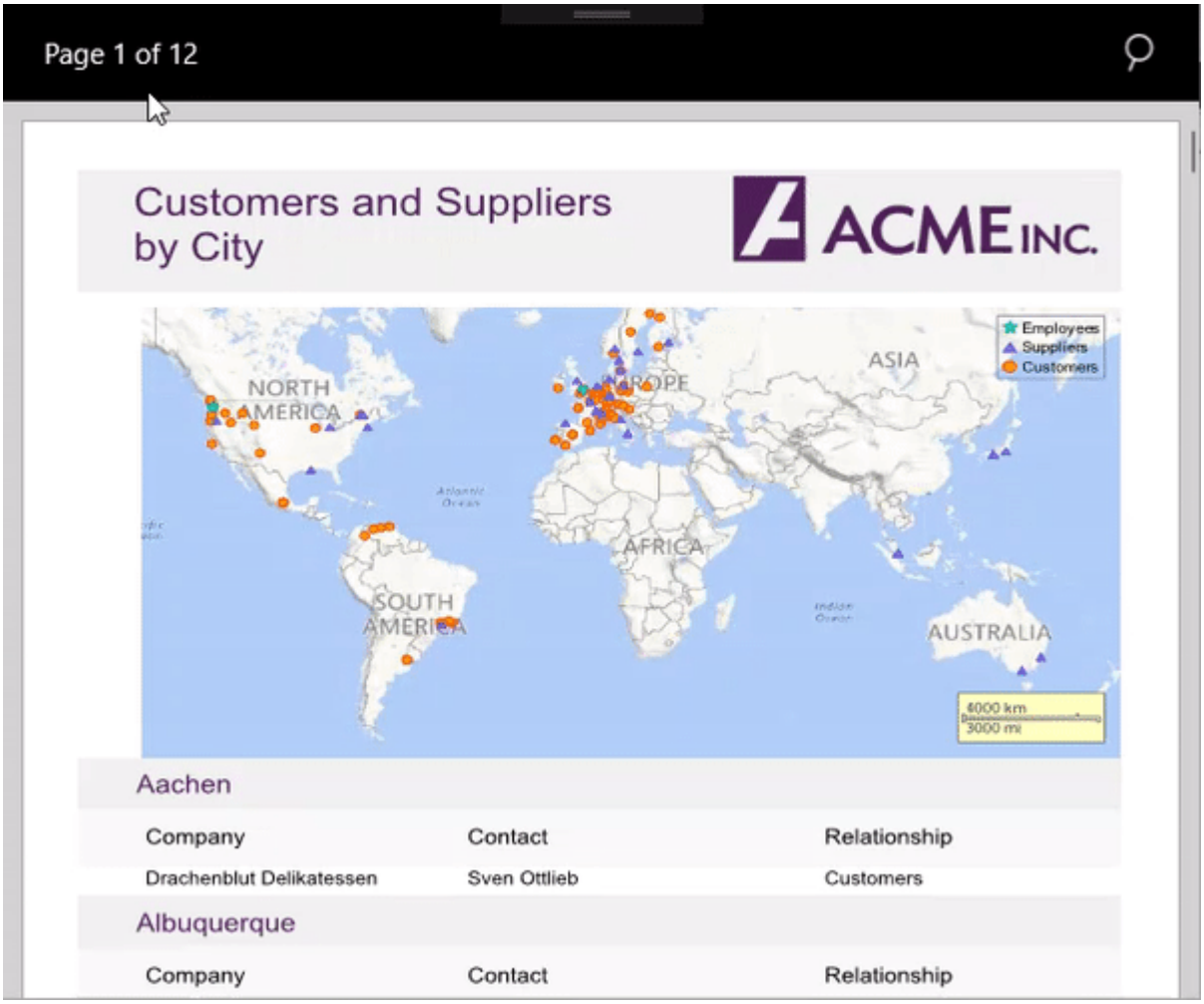
Back to Top

Features

Navigation

At times, depending on the document size, you might need to navigate through multiple pages in a document to view different sections in it. FlexViewer provides you with the interactive Page label that lets you move forward through the pages. On clicking the label, a text box appears allowing you to type a specific page number to jump to that particular page.

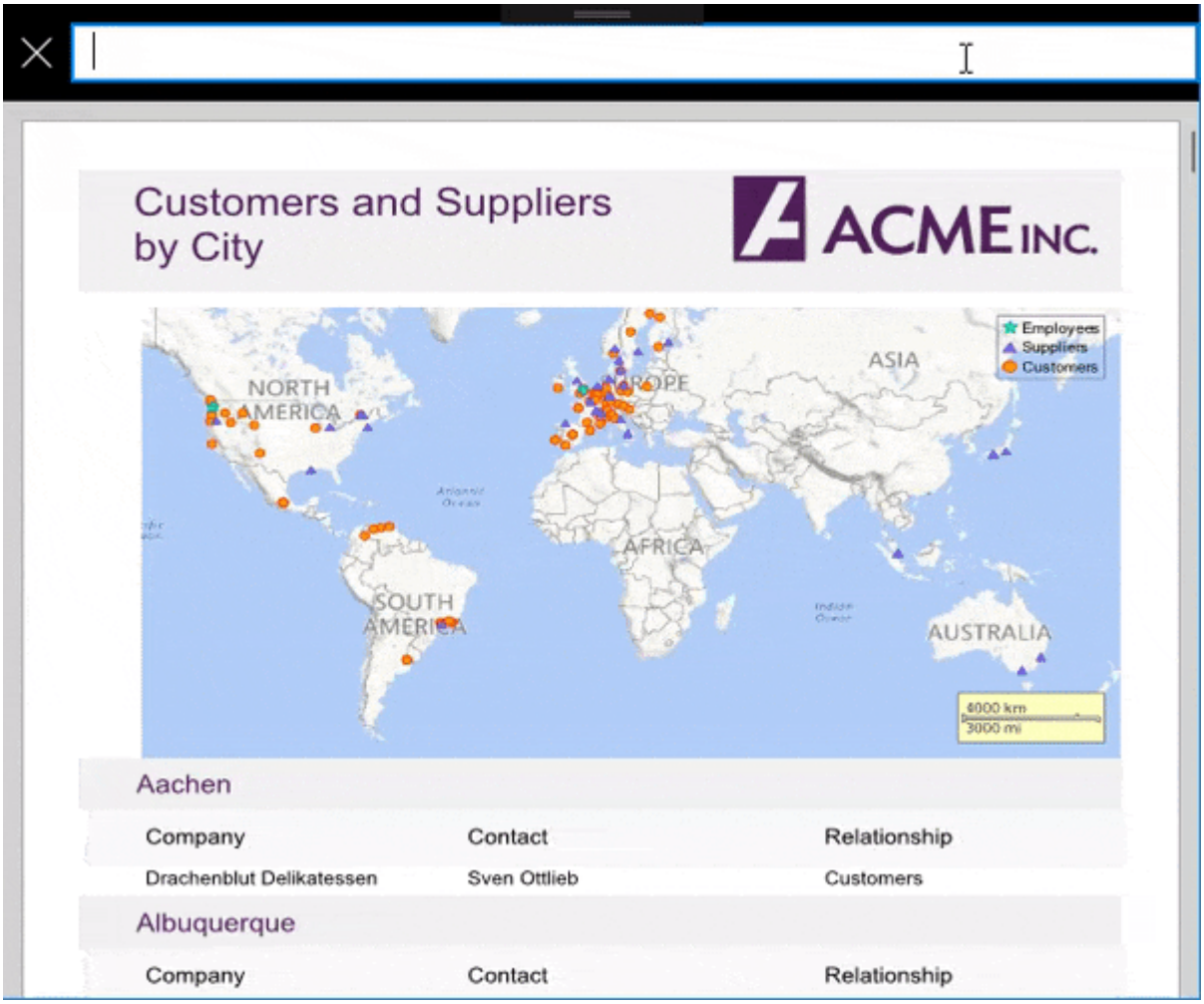
The following GIF depicts how you can navigate to a particular page using page navigation in FlexViewer.



Text Search

FlexViewer provides the flexibility to find text in a document. It provides a search icon at the top-right corner of the toolbar. On clicking the search icon, the search toolbar appears replacing the existing toolbar. In the search toolbar, you can simply type in the text and press Enter to search for it in the document. The FlexViewer control highlights all the matching instances making it easy for you to find all the occurrences of searched text in the document.

The following GIF depicts how text search can be performed in FlexViewer.

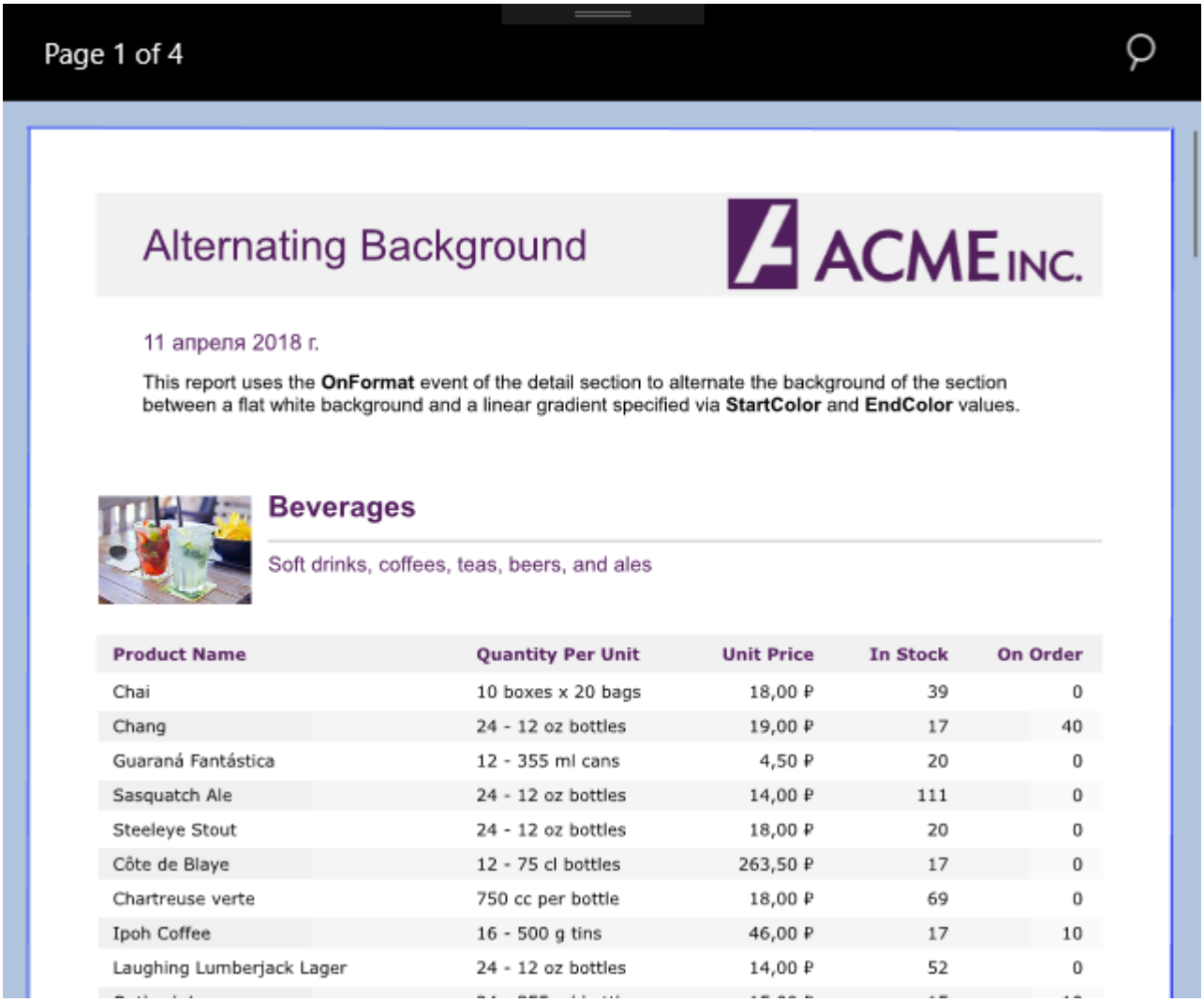


Appearance

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 specific to the FlexViewer control which can be used to customize the appearance of the loaded document and the area around it.
The FlexViewer class provides the following set of properties that can be used to change the overall look and feel of the document:

- **PageBackgroundColor**: Allows you to change the color for filling the page content.
- **PageBorderColor**: Provides a brush used for drawing page borders.
- **Padding**: Sets the amount of padding between pages and the preview window edges.
- **PageSpacing**: Sets the amount of padding between pages in the preview.

The following image shows FlexViewer with the customized appearance.



To change the appearance of the loaded PDF document, use the following code. This example uses the sample code created in Quick Start section.

C# copyCode

```
//Set background color for area of the page
flexViewer.BackgroundColor= Color.LightSteelBlue;

//Fills the page content with color
flexViewer.PageBackgroundColor = Color.White;

//Color the page border
flexViewer.PageBorderColor = Color.Blue;

//Set padding between the pages and preview window edges
flexViewer.Padding = new Thickness(20, 20, 20, 20);

//Set padding between the pages in preview
flexViewer.PageSpacing = 5;
```

Export

FlexViewer allows you to export a PDF document and save it to a file, stream or as an image(s). You can use Save method of the FlexViewer class to save a PDF document to a file or a stream. In addition, you can use various methods provided by the FlexViewer class to save a PDF document as an image. These methods are listed in the following table:

Methods	Description
SaveAsBmp	Saves the document pages as images in BMP format, i.e., one image for each page.
SaveAsGif	Saves the document pages as images in GIF format, i.e., one image for each page.
SaveAsJpeg	Saves the document pages as images in JPEG format, i.e., one image for each page.
SaveAsPng	Saves the document pages as images in PNG format, i.e., one image for each page.
SaveAsTiff	Saves the document pages as images in TIFF format, i.e., one image for each frame.

The following code illustrates how to export PDF pages to images. This example uses the sample code created in Quick Start section.

C# copyCode

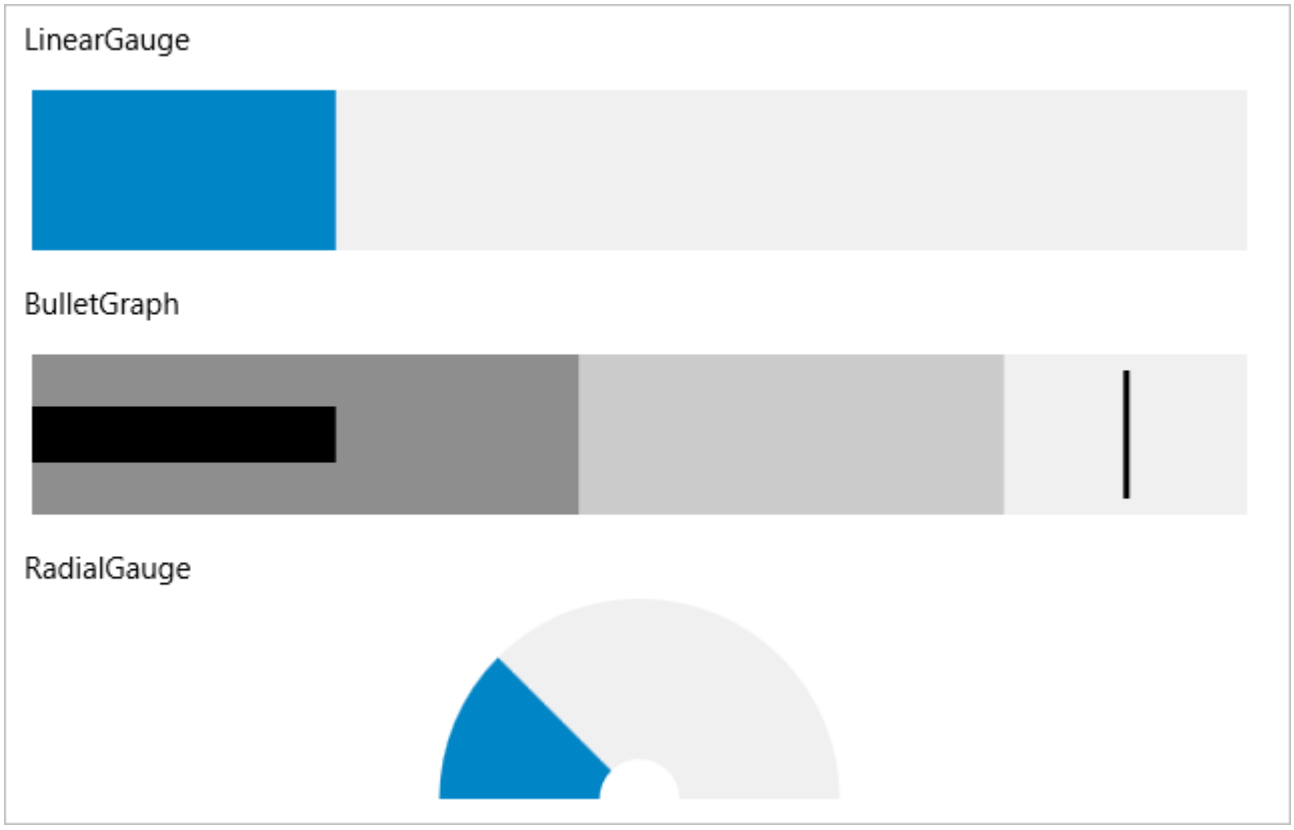
```
//Output range object which defines which pages of the document should be saved
OutputRange pageRange = new OutputRange(1, 2);

//Specify the options that should be used while saving the document's pages to image
SaveAsImageOptions saveOptions = new SaveAsImageOptions()
{
    BackColor = Color.LightCyan,
    Resolution = 100
};

//Save image as png to a local system folder
flexViewer.SaveAsPng(Path.Combine(Environment.GetFolderPath(
    Environment.SpecialFolder.LocalApplicationData), "New{0}.png"), pageRange, saveOptions);
```

UI Virtualization

Virtualization is an optimization technique that renders page visuals only as they come into view. FlexViewer provides the ability to virtualize UI using low memory foot print to open large PDF files without going out of memory. This reduces the PDF loading time and improves performance, thus enhancing UI performance.

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.
- **Drag Support:** Dragging the gauge with finger touch changes the value accordingly.

Gauge Types

Xamarin Edition comprises of three kinds of gauges: LinearGauge, RadialGauge and BulletGraph.

Type	Image	Usage
Linear Gauge: 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. It also provides IsReversed property to reorient the radial gauge so that it is drawn reversed (counter-clockwise).		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 C1.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 Content 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:c1="clr-namespace:C1.Xamarin.Forms.Gauge;assembly=C1.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 Content Page:** In the class constructor App( ), set the Content 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. (Optional) In case you compile your UWP application in **Release** mode, you need to explicitly add the following code to the **OnLaunched** method in your **App.xaml.cs** to include the correct assemblies with your application.

        C#
        ```
        var assembliesToInclude = new List<Assembly>();
        assembliesToInclude.Add(typeof(C1.Xamarin.Forms.Gauge.Platform.UWP.C1GaugeRenderer)
        .GetTypeInfo().Assembly);
        assembliesToInclude.Add(typeof(C1.UWP.Gauge.C1GaugeRenderer).GetTypeInfo().Assembly);
        Xamarin.Forms.Forms.Init(e, assembliesToInclude);
        ```
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;
    gauge.ShowRanges = true;

    //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 Content 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:c1="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" ShowRanges="True">
            <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

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 Content Page:** In the class constructor App( ), set the Content 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. (Optional) In case you compile your UWP application in **Release** mode, you need to explicitly add the following code to the **OnLaunched** method in your **App.xaml.cs** to include the correct assemblies with your application.

```
C#
var assembliesToInclude = new List<Assembly>();
assembliesToInclude.Add(typeof(C1.Xamarin.Forms.Gauge.Platform.UWP.C1GaugeRenderer)
.GetTypeInfo().Assembly);
assembliesToInclude.Add(typeof(C1.UWP.Gauge.C1GaugeRenderer).GetTypeInfo().Assembly);
Xamarin.Forms.Forms.Init(e, assembliesToInclude);
```

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

Back to Top

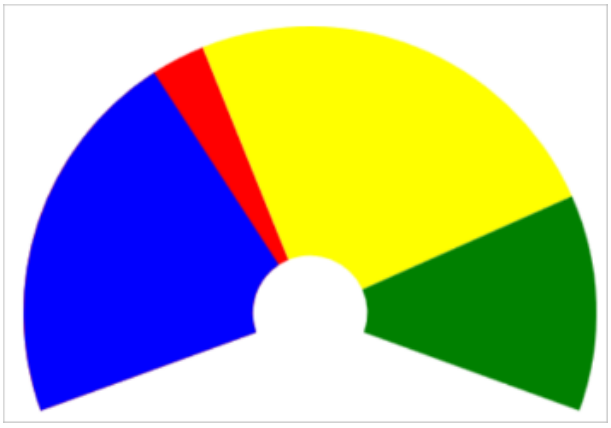
RadialGauge Quick Start

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
    C1RadialGauge gauge = new C1RadialGauge();
    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;
    gauge.ShowRanges = true;

    //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 Content 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:c1="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>
<c1:C1RadialGauge Value="35" Min="0" Max="100"  ShowText="None" StartAngle = "-20"
        SweepAngle = "220" AutoScale = "true" PointerColor="Blue" ShowRanges="True">
            <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 Content Page:** In the class constructor App( ), set the Content 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. (Optional) In case you compile your UWP application in **Release** mode, you need to explicitly add the following code to the **OnLaunched** method in your **App.xaml.cs** to include the correct assemblies with your application.

```
C#
var assembliesToInclude = new List<Assembly>();
assembliesToInclude.Add(typeof(C1.Xamarin.Forms.Gauge.Platform.UWP.C1GaugeRenderer)
.GetTypeInfo().Assembly);
assembliesToInclude.Add(typeof(C1.UWP.Gauge.C1GaugeRenderer).GetTypeInfo().Assembly);
Xamarin.Forms.Forms.Init(e, assembliesToInclude);
```

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.

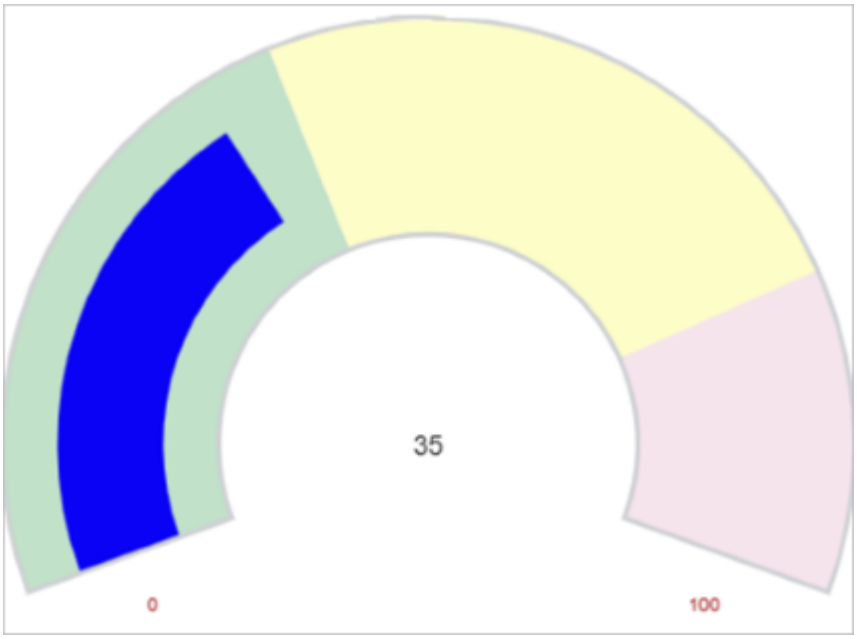
The following code example demonstrates how to set this property in C#. These example uses the sample created in the Quick start.

```
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.

In Code

C#

```
//Customize Gauge
gauge.ShowText = GaugeShowText.All;
gauge.Thickness = 0.5;
gauge.FaceBorderColor = Color.FromHex("#cfd1d4");
gauge.FaceBorderWidth = 5;

gauge.MinTextColor = Color.Brown;
gauge.MaxTextColor = Color.Brown;
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;
```

In XAML

XAML

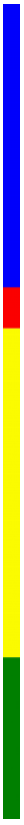
```
<c1:C1RadialGauge Min="0" Max="100" Value="35" StartAngle = "-20" SweepAngle = "220"
AutoScale = "true"
    FaceBorderColor="#cfd1d4" FaceBorderWidth="5" MinTextColor="Brown"
MaxTextColor="Brown" >
```

```
    <c1:C1RadialGauge.Ranges>
        <c1:GaugeRange  Min="0" Max="40" Color="#C1E1C9"/>
        <c1:GaugeRange Min="40" Max="80" Color="#fdfdc8"/>
        <c1:GaugeRange Min="80" Max="100" Color="#f6e4ed"/>
    </c1:C1RadialGauge.Ranges>
    <c1:C1RadialGauge.Pointer>
        <c1:GaugeRange Thickness="0.5"></c1:GaugeRange>
    </c1:C1RadialGauge.Pointer>
</c1:C1RadialGauge>
```

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#

```
<c1:C1LinearGauge 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#

```
<c1:C1LinearGauge Value="35" Min="0" Max="100" Thickness="0.1">
      <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>
```

User Scenario

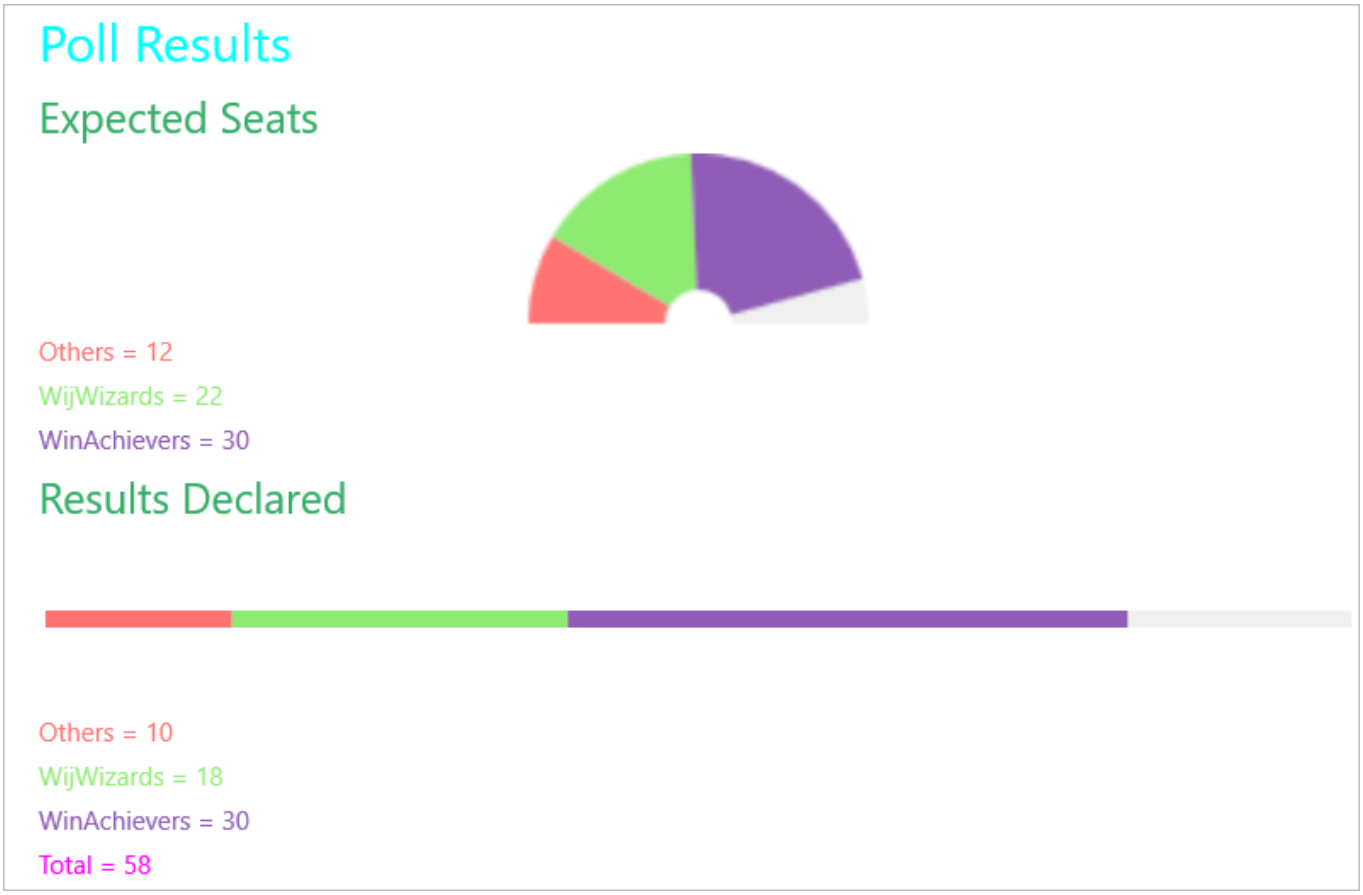
Poll Results

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#

```
class ElectionData
{
    double others, wijWizards, winAchievers;

    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; }
    }
```

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 Content 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:c1="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="Aqua"  Font="30" />
    <Label Text="Expected Seats" TextColor="#36ae69" Font="25"/>
    <!--Radial Gauge-->
    <c1:C1RadialGauge x:Name="gauge"  Min="0" Max="70" ShowText="None" AutoScale
= "true" ShowRanges="True">

        <!--Colored ranges to represent parties-->
        <c1:C1RadialGauge.Ranges>
            <c1:GaugeRange x:Name="Range1" Color="#FF7373"/>
            <c1:GaugeRange x:Name="Range2"  Color="#8DEB71"/>
            <c1:GaugeRange x:Name="Range3"  Color="#8F5DB7"/>
        </c1:C1RadialGauge.Ranges>
    </c1:C1RadialGauge>
```

```
    <!--Labels to display party names and seat count-->
    <Label x:Name="lbl" TextColor="#FF7373"/>
    <Label x:Name="lbl1" TextColor="#8DEB71"/>
    <Label x:Name="lbl2" TextColor="#8F5DB7"/>
    <Label Text="Results Declared" TextColor="#36ae69" Font="25"/>

    <!--Linear Gauge-->
    <c1:C1LinearGauge x:Name="lGauge"  Min="0" Max="70" Thickness="0.1"
Direction="Right" ShowRanges="True">

        <!--Colored ranges to represent parties-->
        <c1:C1LinearGauge.Ranges>
            <c1:GaugeRange x:Name="lRange1"  Color="#FF7373"/>
            <c1:GaugeRange x:Name="lRange2"  Color="#8DEB71"/>
            <c1:GaugeRange x:Name="lRange3"  Color="#8F5DB7"/>
        </c1:C1LinearGauge.Ranges>
    </c1:C1LinearGauge>

    <!--Labels to display party names and seat count-->
    <Label x:Name="llbl" TextColor="#FF7373"/>
    <Label x:Name="llbl1" TextColor="#8DEB71"/>
    <Label x:Name="llbl2" TextColor="#8F5DB7"/>
    <Label x:Name="llbl4" TextColor="Fuchsia"/>
<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#

```
// Results Declared
ElectionData bds = new ElectionData() { Others = 10, WijWizards = 18,
WinAchievers = 30 };

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;

// Add data to labels
llbl.Text = "Others = " + bds.Others;
llbl1.Text = "WijWizards = " + bds.WijWizards;
llbl2.Text = "WinAchievers = " + bds.WinAchievers;
llbl4.Text = "Total = " + (bds.Others + bds.WijWizards +
bds.WinAchievers).ToString();

// Expected Seats
ElectionData ds = new ElectionData() { Others = 12, WijWizards = 22,
WinAchievers = 30 };

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;

// Add data to labels
lbl.Text = "Others = " + ds.Others;
lbl1.Text = "WijWizards = " + ds.WijWizards;
lbl2.Text = "WinAchievers = " + ds.WinAchievers;
```

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 Content Page ElectionResult 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 ElectionResult();
   }
   ```

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.

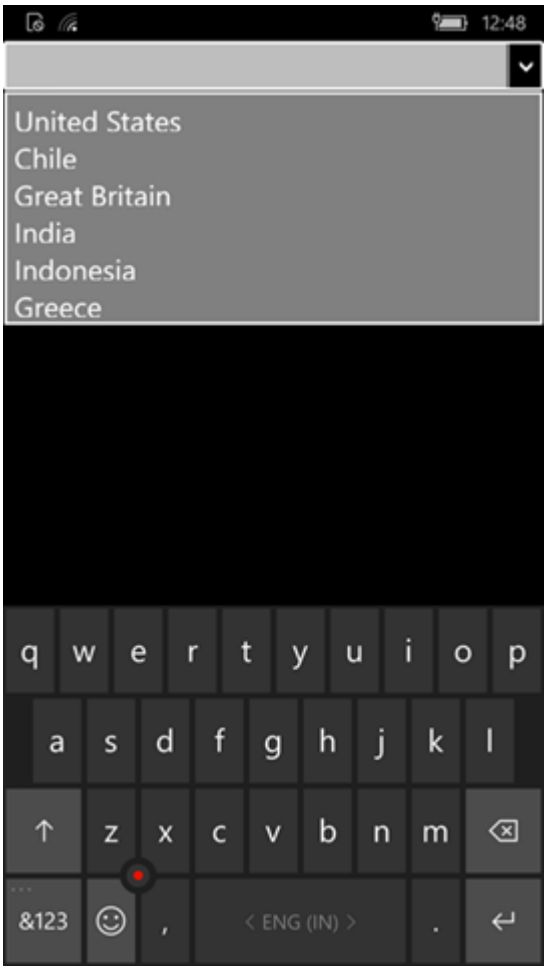
Back to Top

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.
- **Highlight Matches** - Highlight the input text with matching string in the suggestions.

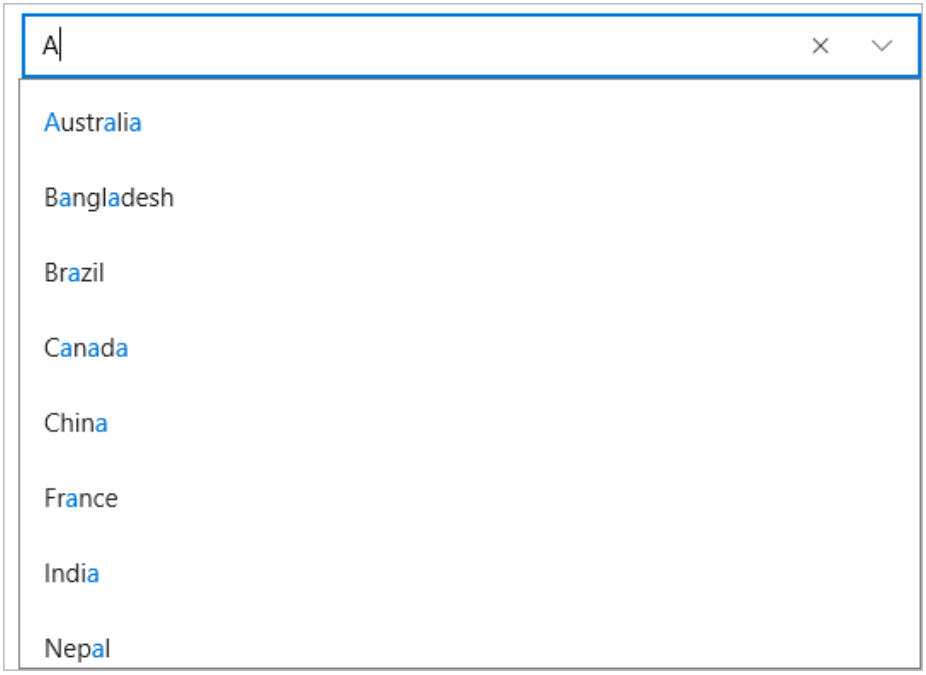
Quick Start: Populating AutoComplete with Data

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 override string ToString()
        {
            return Name;
        }

        public static readonly string[] Countries = { "Germany", "Japan", "Australia",
            "Bangladesh", "Brazil", "Canada", "China", "France", "India", "Nepal",
            "Pakistan", "Srilanka" };
        public string Name { get; set; }
        public static ObservableCollection<Country> GetCountries()
        {
```

```
            ObservableCollection<Country> listContries = new ObservableCollection<Country>();

            foreach (var item in Countries)
            {
                listContries.Add(new Country() { Name = item });
            }
            return listContries;
        }
    }
```

Step 2: Add AutoComplete control and populate it with suggestions

1. Add a new Content 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 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">
    <c1:C1AutoComplete x:Name="autoComplete" ItemsSource="{Binding ItemsSource}"
DisplayMemberPath="Name"
        HorizontalOptions="FillAndExpand" VerticalOptions="Start"></c1: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#

```
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 Content 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();
        ```

        3. (Optional) In case you compile your UWP application in **Release** mode, you need to explicitly add the following code to the **OnLaunched** method in your **App.xaml.cs** to include the correct assemblies with your application.

        C#

        ```
        var assembliesToInclude = new List<Assembly>();
        ```

```
assembliesToInclude.Add(typeof(C1.Xamarin.Forms.Input.Platform.UWP.C1InputRenderer)
.GetTypeInfo().Assembly);
assembliesToInclude.Add(typeof(C1.UWP.Input.C1InputRenderer).GetTypeInfo().Assembly);
Xamarin.Forms.Forms.Init(e, assembliesToInclude);
```

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#

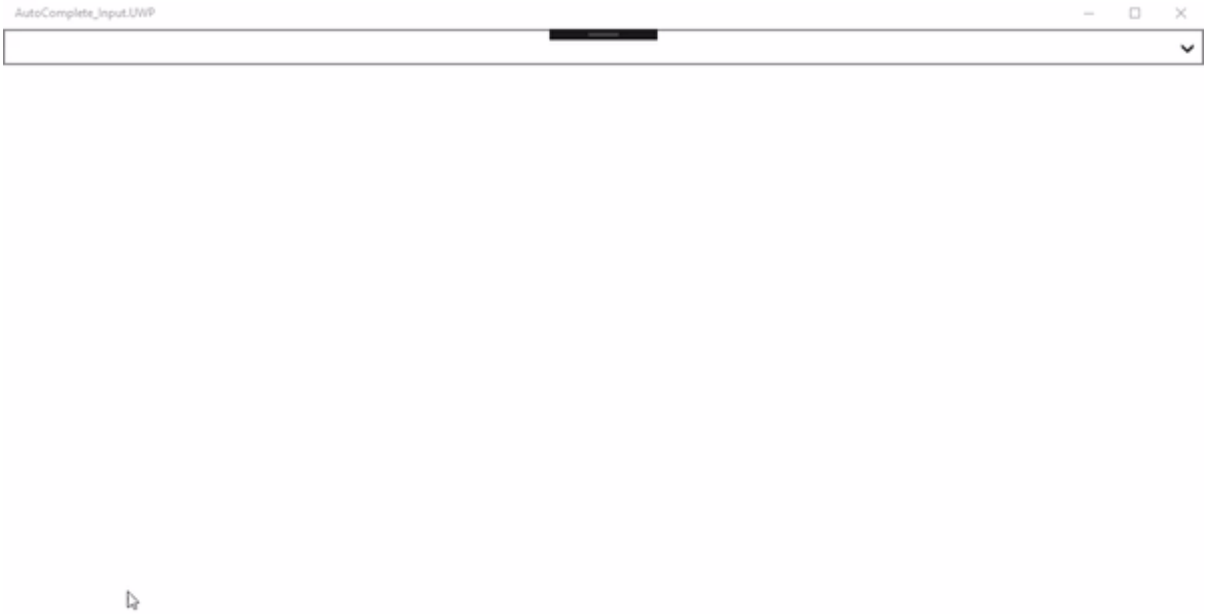
```
autoComplete.ItemsSource = Country.GetCountries();
autoComplete.DisplayMemberPath = "Name";
```

AutoComplete Mode

AutoComplete supports multiple filtering criteria that can be used to filter the items list based on the user input. These filtering criteria are defined in the AutoCompleteMode enumeration and can be set using the **AutoCompleteMode** property. The AutoCompleteMode property works as a series of flags, therefore, you can also set multiple filtering criteria. **AutoCompleteMode** enumeration specifies the mode for automatic completion in the control through following values:

1. StartsWith
2. Contains
3. EndsWith
4. MatchCase
5. MatchWholeWord

The following GIF image shows how the AutoComplete control appears after setting the **AutoCompleteMode** property.



The following code example shows setting the auto complete mode feature in the AutoComplete control.

In Code

C#

```
autoComplete.AutoCompleteMode = AutoCompleteMode.Contains |
AutoCompleteMode.StartsWith;
```

In XAML

XAML

```
<c1:C1AutoComplete x:Name="autoComplete" ItemsSource="{Binding ItemsSource}"
DisplayMemberPath="Name"
HorizontalOptions="FillAndExpand" AutoCompleteMode="Contains"></c1:C1AutoComplete>
```

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.

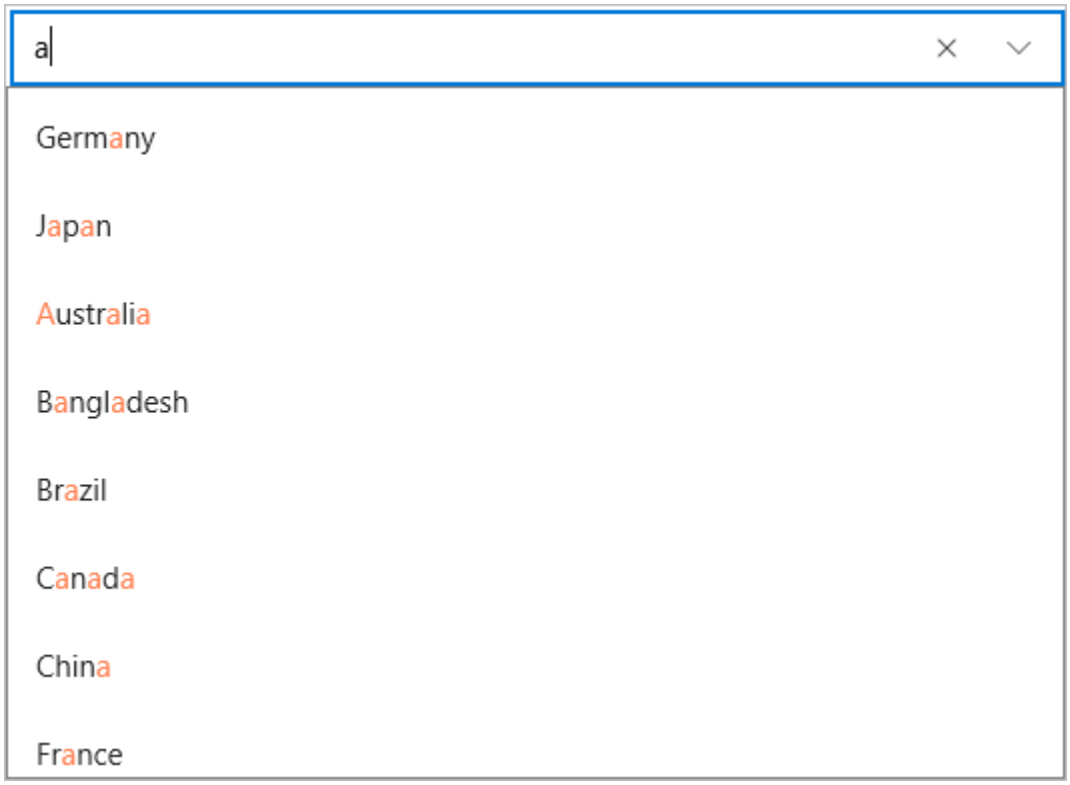
The following code example shows setting time delay in the AutoComplete control.

C#

```
//Setting the time delay
autoComplete.Delay = TimeSpan.FromMilliseconds(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.

In Code

C#

```
autoComplete.HighlightedColor = Xamarin.Forms.Color.Coral;
```

In XAML

XAML

```
<c1:C1AutoComplete x:Name="autoComplete" ItemsSource="{Binding ItemsSource}"
DisplayMemberPath="Name"
    HorizontalOptions="FillAndExpand" HighlightedColor="Coral">
```

Keyboard Navigation

AutoComplete provides keyboard support that can be used to navigate through the drop down list. The keyboard navigation keys can replace the use of mouse by allowing you to edit and navigate through the drop down list. The keys and their corresponding actions are listed below:

Keys	Actions
Arrow Up and Down	Navigates through the list of items in the drop down. If drop down isn't open, it remains hidden but the next item is selected.
Enter	Begins/Ends editing, selects the highlighted item, and hides the drop down
Esc	Hides the drop down
Tab	Jumps between different controls on the platforms that support tab stops (UWP)

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.

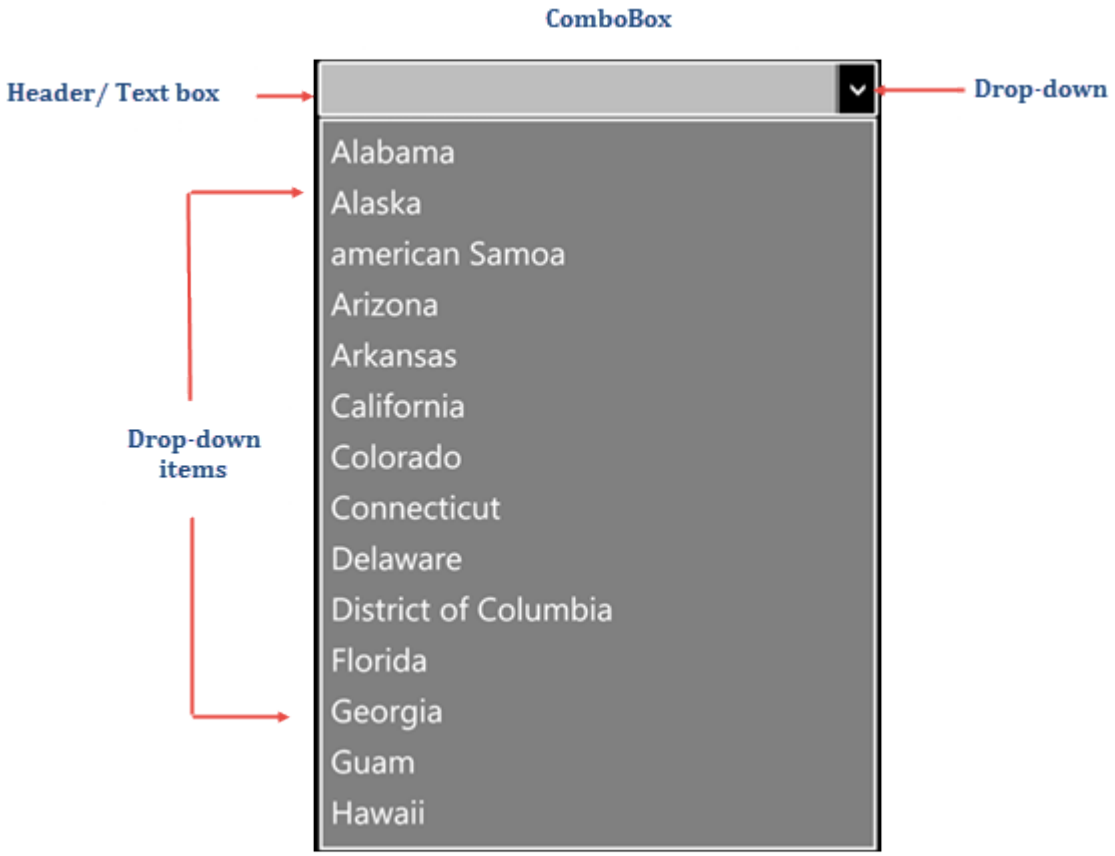
You can set the value of the checkbox by setting the **IsChecked** property. You can customize the style by setting the **Color** property. The following image shows a CheckBox displayed in a FlexGrid template.

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
☐	Mark Heath	$6,333.38
☑	Herb Krause	$448.51
☐	Ulrich Lehman	$7,100.58
☑	Jack Ulam	$8,765.26
◢ Indonesia (6 items)		$22,132.60
☑	Ulrich Quaid	$265.49
☐	Ted Richards	$1,581.26
☐	Xavier Neiman	$4,353.91
☐	Charlie Evers	$7,787.81
☑	Vic Lehman	$1,488.28
☐	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.

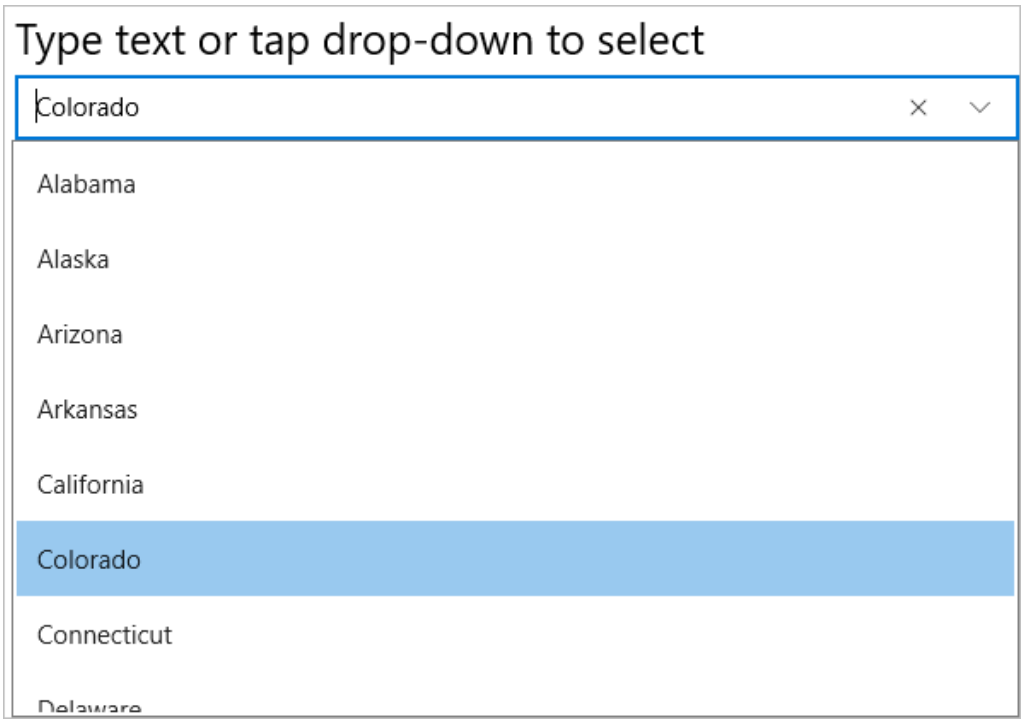
Quick Start: Display a ComboBox Control

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 Content 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"
            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();
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 Content 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();
        ```

        3. (Optional) In case you compile your UWP application in **Release** mode, you need to explicitly add the following code to the **OnLaunched** method in your **App.xaml.cs** to include the correct assemblies with your application.

        C#

        ```
        var assembliesToInclude = new List<Assembly>();
        assembliesToInclude.Add(typeof(C1.Xamarin.Forms.Input.Platform.UWP.C1InputRenderer)
        .GetTypeInfo().Assembly);
        assembliesToInclude.Add(typeof(C1.UWP.Input.C1InputRenderer).GetTypeInfo().Assembly);
        Xamarin.Forms.Forms.Init(e, assembliesToInclude);
        ```

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

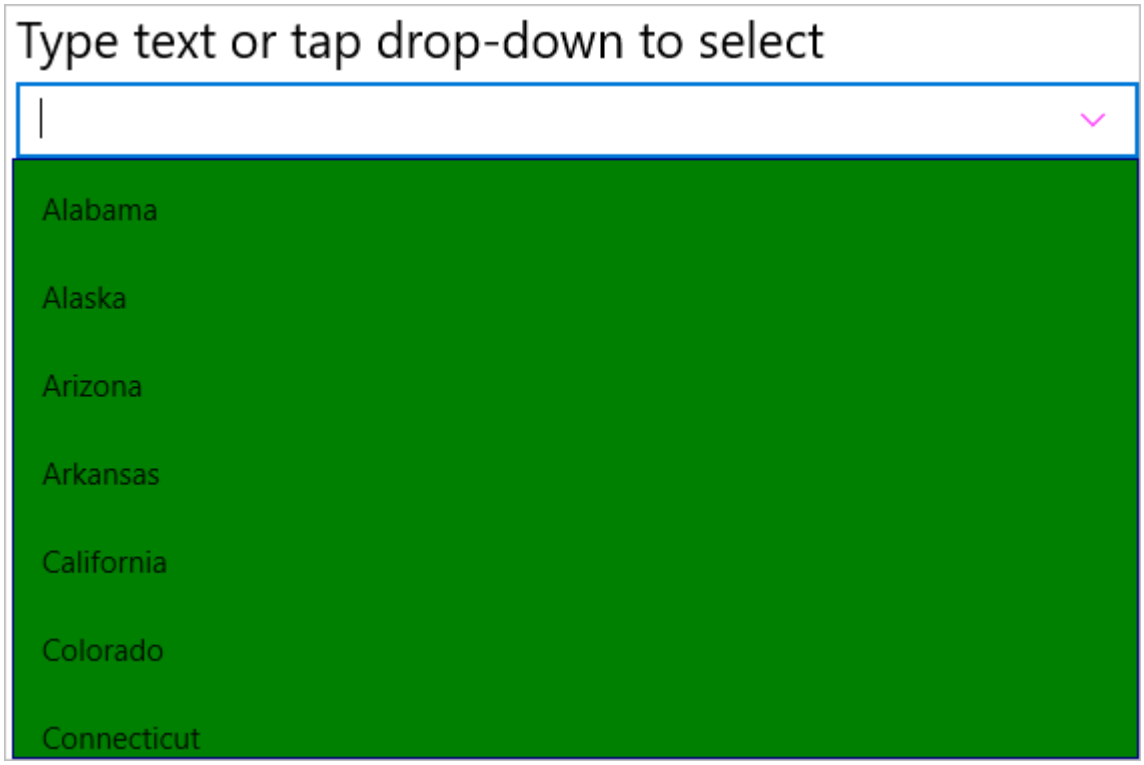
Features

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
- DropDownMode - sets the display mode in which the drop-down opens, accepting values from the DropDownMode 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.

In Code

C#

```
cbxEdit.ButtonColor = Xamarin.Forms.Color.Magenta;
cbxEdit.DropDownBackgroundColor = Xamarin.Forms.Color.Green;
cbxEdit.DropDownBorderColor = Xamarin.Forms.Color.Navy;
cbxEdit.DropDownMode = C1.Xamarin.Forms.Input.DropDownMode.ForceBelow;
```

In XAML

XAML

```
<c1:C1ComboBox x:Name="cbxEdit" HorizontalOptions="FillAndExpand"
DisplayMemberPath="Name"
    ButtonColor="Magenta" DropDownBackgroundColor="Green" DropDownBorderColor="Navy"
    DropDownMode="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#

```
var array = States.GetStates();
cbxEdit.ItemsSource = array;
cbxEdit.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. By default, the ComboBox control is editable, allowing users to provide input by typing. However, you can disable editing by setting the value of IsEditable property to False.

To change the default behavior, set the IsEditable property using the following code.

In Code

C#

```
this.cbxEdit.IsEditable = false;
```

In XAML

XAML

```
<c1:C1ComboBox x:Name="cbxEdit" IsEditable="False" HorizontalOptions="FillAndExpand"
    DisplayMemberPath="Name" />
```

Keyboard Navigation

ComboBox provides keyboard support that can be used to navigate through the drop down list. The keyboard navigation keys can replace the use of mouse by allowing you to edit and navigate through the drop down list. The keys and their corresponding actions are listed below:

Keys	Actions
Arrow Up and Down	Navigates through the list of items in the drop down. If drop down isn't open, it remains hidden but the next item is selected.

Enter	Begins/Ends editing, selects the highlighted item, and hides the drop down
Esc	Hides the drop down
Tab	Jumps between different controls on the platforms that support tab stops (UWP)

DropDown

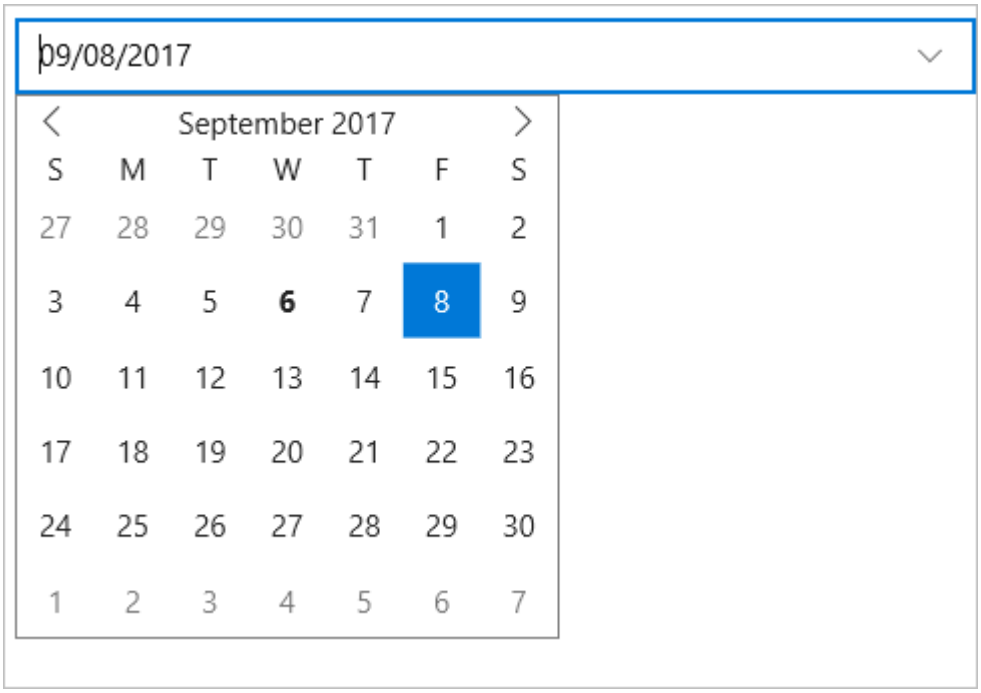
DropDown is a basic drop-down control that can be used as a base to create custom drop-down controls such as date picker, auto complete menus, etc. The control comprises three major elements, including a Header view, a button, and a DropDown view. The header includes the entire width of the control, 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 MaskedEntry 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 Content 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:calendar="clr-
namespace:C1.Xamarin.Forms.Calendar;assembly=C1.Xamarin.Forms.Calendar"
```

4. Initialize a DropDown control, a MaskedEntry control, and a Calendar control by adding the given markup between the <ContentPage></ContentPage> tags.

XAML

```
<StackLayout>
    <Grid VerticalOptions="FillAndExpand" >
        <c1:C1DropDown x:Name="dropdown" HorizontalOptions="FillAndExpand"
VerticalOptions="Start" >
            <c1:C1DropDown.Header>
                <c1:C1MaskedEntry x:Name="mask" Mask="00/00/0000" />
            </c1:C1DropDown.Header>
            <c1:C1DropDown.DropDown>
                <calendar:C1Calendar x:Name="calendar"
HorizontalOptions="FillAndExpand"
                    DayOfWeekFormat="d"  BackgroundColor="Transparent"
WidthRequest="270"
                    HeightRequest="270">
                </calendar:C1Calendar>
            </c1:C1DropDown.DropDown>
        </c1:C1DropDown>
    </Grid>
</StackLayout>
```

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 MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
        this.calendar.SelectionChanged += CalendarSelectionChanged;
    }

    private void CalendarSelectionChanged(object sender,
CalendarSelectionChangedEventArgs e)
    {
        C1Calendar calendar = (C1Calendar)sender;
        this.mask.Value =
((DateTime)calendar.SelectedDate).ToString("MM/dd/yyyy");
        this.dropdown.IsDropDownOpen = false;
    }

    }
```

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

Keyboard Navigation

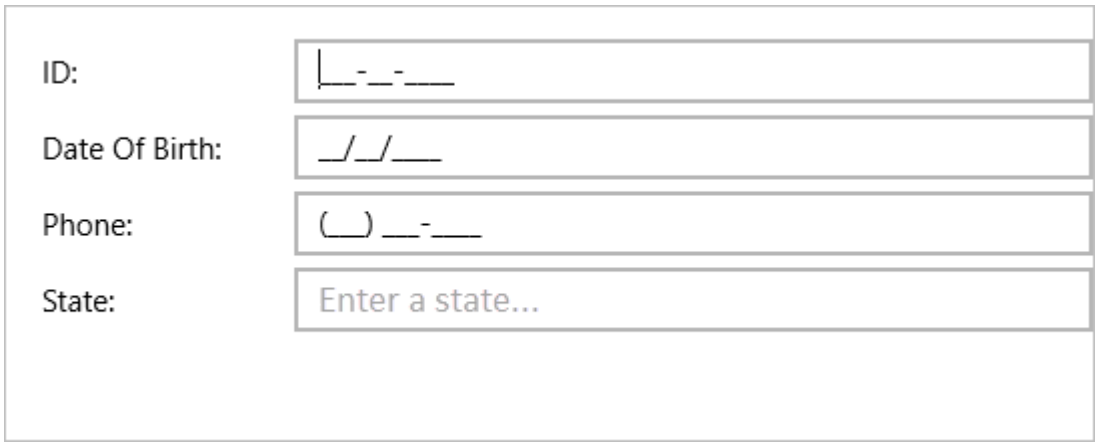
DropDown provides keyboard support that can be used to navigate through the drop down list. The keyboard navigation keys can replace the use of mouse by allowing you to edit and navigate through the drop down list. The keys and their corresponding actions are listed below:

Keys	Actions
Arrow Up and Down	Navigates through the list of items in the drop down. If drop down isn't open, it remains hidden but the next item is selected.
Enter	Begins/Ends editing, selects the highlighted item, and hides the drop down
Esc	Hides the drop down
Tab	Jumps between different controls on the platforms that support tab stops (UWP)

MaskedEntry

MaskedEntry 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 MaskedEntry 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.



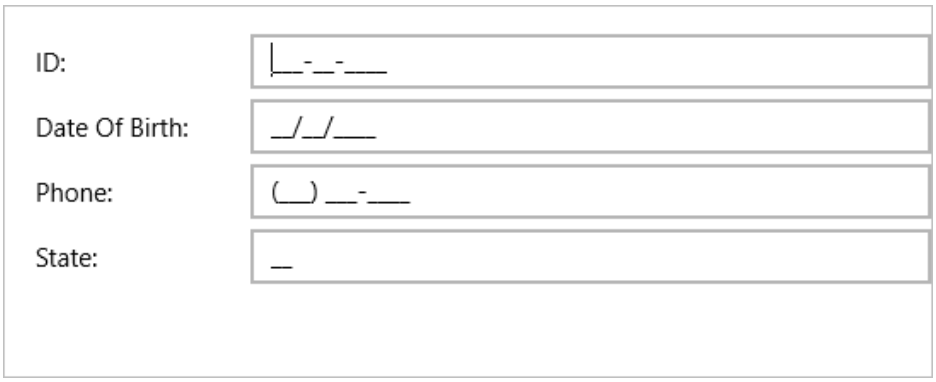
Quick Start: Display a MaskedEntry Control

This section describes adding a MaskedEntry 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 MaskedEntry control.

- **Step 1: Add MaskedEntry controls to display four input fields**
- **Step 2: Run the Project**

The following image shows the input fields configured after completing the above steps.



Step 1: Add MaskedEntry controls to display four input fields

Complete the following steps to initialize a MaskedEntry 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 Content 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 MaskedEntry 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 VerticalOptions="Center">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="120"/>
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Label Text="ID:" VerticalOptions="Center" />
        <c1:C1MaskedEntry x:Name="c1MaskedTextBox1" Mask="000-00-0000" Grid.Column="1"  />
        <Label Text="Date Of Birth:"  VerticalOptions="Center"  Grid.Row="1" />
        <c1:C1MaskedEntry x:Name="c1MaskedTextBox2" Keyboard="Numeric" Mask="90/90/0000"
            Grid.Row="1" Grid.Column="1" />
        <Label Text="Phone:"  VerticalOptions="Center"  Grid.Row="2"  />
        <c1:C1MaskedEntry x:Name="c1MaskedTextBox3" Mask="(999) 000-0000" Keyboard="Telephone"
            Grid.Row="2" Grid.Column="1"   />
        <Label Text="State:" VerticalOptions="Center" Grid.Row="3" />
        <c1:C1MaskedEntry 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 Content 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();
```

     3. (Optional) In case you compile your UWP application in **Release** mode, you need to explicitly add the following code to the **OnLaunched** method in your **App.xaml.cs** to include the correct assemblies with your application.

```
C#
var assembliesToInclude = new List<Assembly>();
assembliesToInclude.Add(typeof(C1.Xamarin.Forms.Input.Platform.UWP.C1InputRenderer)
.GetTypeInfo().Assembly);
assembliesToInclude.Add(typeof(C1.UWP.Input.C1InputRenderer).GetTypeInfo().Assembly);
Xamarin.Forms.Forms.Init(e, assembliesToInclude);
```

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

Mask Symbols

The MaskedEntry 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 MaskedEntry 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 MaskedEntry control.

Mask Symbol	Description
0	Digit
9	Digit or space
#	Digit, sign, or space
L	Letter
?	Letter, optional
C	Character, optional
&	Character, required
I	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.

Toggle Button

C1ToggleButton provides a cross-platform implementation of the ToggleButton control. The control represents a two state button that a user can select (check) or clear (uncheck). It can be used to visualize Boolean values much like a CheckBox control, and allows users to make changes in the state of the control by tapping it. Moreover, the **C1ToggleButton** control offers more style options than a Checkbox control. It provides you the ability to customize the color, text, or even a custom image or view for each state. You can change the state of the control by setting the IsChecked property. Text can be controlled with the CheckedText and UncheckedText properties, images can be controlled with the CheckedImageSource and UncheckedImageSource properties, and views can be controlled with CheckedContent and UncheckedContent properties.

Quick Start: Change State and Customize the Control

This section describes adding the C1ToggleButton control to your portable or shared application and changing color of the control on the basis of change in state of the control.

Complete the following steps to change color of the control on changing its state.

- **Step 1: Create an event to change the state of the control**
- **Step 2: Run the Project**

The following image shows the change in color of the control on changing its state.

Step 1: Create an event to change the state of the control

1. Create a new portable or shared Xamarin.Forms application (Refer Creating a New Xamarin.Forms App for detailed instructions).
2. Edit the <ContentPage> tag to include the following reference.

   XAML
   ```
   xmlns:c1tog="clr-namespace:C1.Xamarin.Forms.Core;assembly=C1.Xamarin.Forms.Core"
   ```
3. Initialize the C1ToggleButton control and set some of its basic properties such as Name, BackgroundColor, etc. by adding the given markup inside the <ContentPage></ContentPage> tags.

   XAML
   ```
   <c1tog:C1ToggleButton x:Name="Toggle" BackgroundColor="Green"
   HeightRequest="50" WidthRequest="100" HorizontalOptions="Center"
   VerticalOptions="Center" CheckedText="Checked" UncheckedText="Unchecked"/>
   ```
4. Expand the MainPage.xaml node in the Solution Explorer to open MainPage.xaml.cs and add the following code to the MainPage class to create an event handler method for changing color of the control on the basis of change in its state.

   C#
   ```
   private void Toggle_Checked(object sender, EventArgs e)
   {
       if(Toggle.IsChecked == true)
       {
           Toggle.BackgroundColor = Color.Green;
       }
       else if(Toggle.IsChecked == false)
       {
           Toggle.BackgroundColor = Color.Red;
       }
   }
   ```
5. Subscribe to the Toggle_Checked event in the constructor using the given code.

   C#
   ```
   Toggle.Checked += Toggle_Checked;
   ```

Back to Top

Step 2: Run the Project

1. In the Solution Explorer, double-click **App.cs** file to open it.
2. To return a Content Page, add the following code in the constructor App() as illustrated in the given code

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

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.Core.Platform.iOS.C1CoreRenderer.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.Core.Platform.UWP.C1CoreRenderer.Init();
```

        3. (Optional) In case you compile your UWP application in **Release** mode, you need to explicitly add the following code to the **OnLaunched** method in your **App.xaml.cs** to include the correct assemblies with your application.

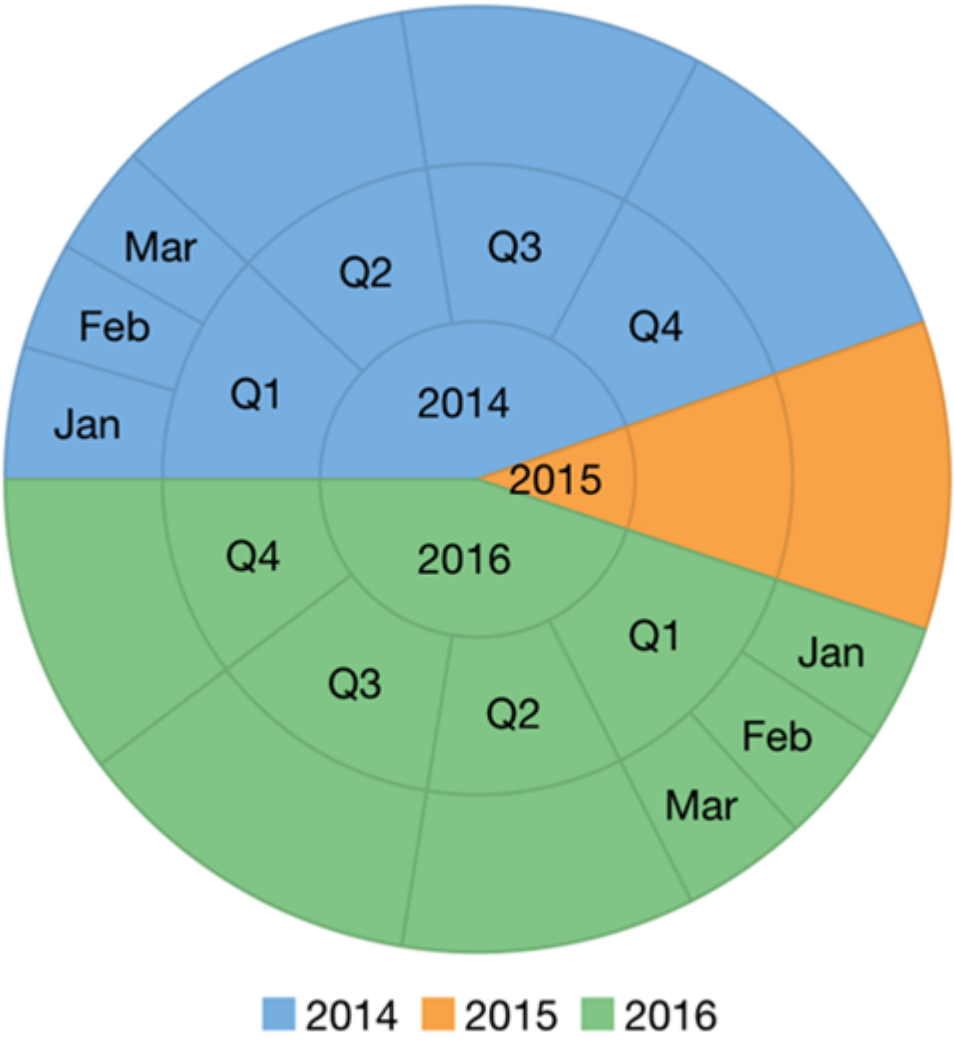
```
C#
var assembliesToInclude = new List<Assembly>
();assembliesToInclude.Add(typeof(C1.Xamarin.Forms.Core.Platform.UWP.C1CoreRenderer).GetTypeInfo().Assembly);Xamarin.Forms.Forms.Init(e,
assembliesToInclude);
```

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

Back to Top

Sunburst Chart

Sunburst chart is used to display hierarchical data, represented using concentric circles. The circle in the center represents the root node, with the data moving outside from the center. A section of the inner circle supports a hierarchical relationship to those sections of the outer circle which lie within the angular area of the parent section. Sunburst chart can be effectively used in scenarios where you want to display hierarchical data, represented using relationship between outer rings and inner rings.



Key Features and Properties

- **Donut Chart Support** - Specify the control's inner radius property to support donut charts.
- **Legend**: Provides a short description of data being rendered on chart and you can also change position of the legend as needed.
- **Selection**: Change the selection mode and customize the selected pie slice appearance.
- **Header and Footer**: Specifies the title header and footer text.

For more information on Sunburst chart, see QuickStart.

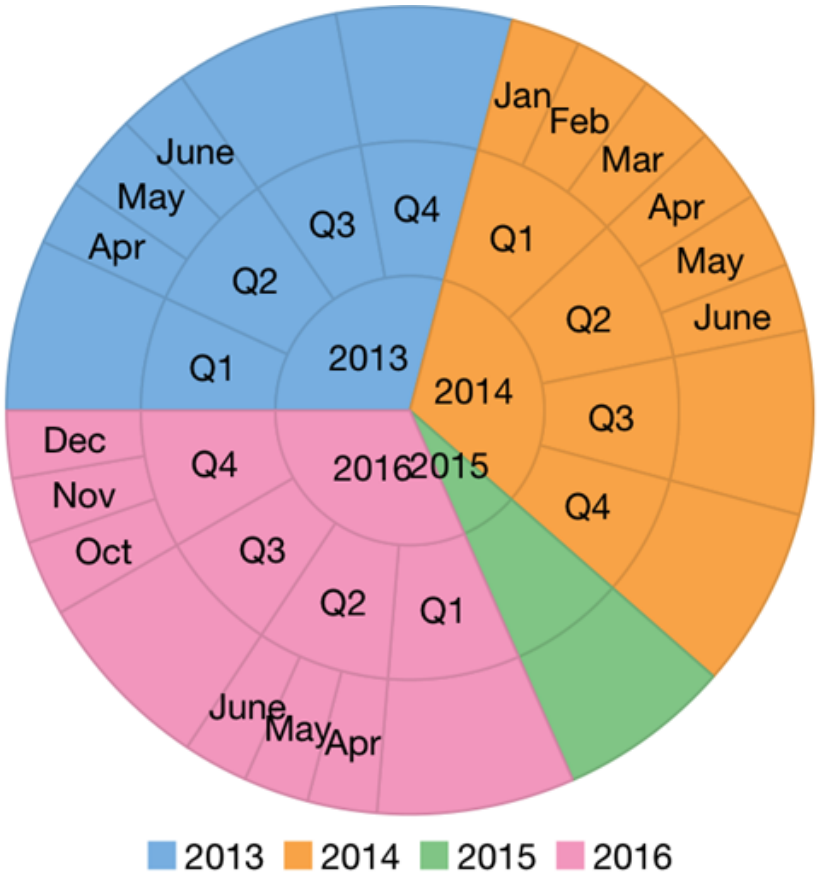
QuickStart

This section describes how to add a Sunburst Chart control to your portable or shared app and add data to it. For more 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 SunburstChart**
- **Step 2: Add a Sunburst Chart control**
- **Step 3: Run the Project**

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



Step 1: Create a Data source for Sunburst Chart

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

C#

```
public class DataService
{
    Random rnd = new Random();
    static DataService _default;

    public static DataService Instance
    {
        get
        {
            if (_default == null)
            {
                _default = new DataService();
            }

            return _default;
        }
    }

    public static List<SunburstDataItem> CreateHierarchicalData()
    {
        Random rnd = Instance.rnd;

        List<string> years = new List<string>();
        List<List<string>> times = new List<List<string>>()
        {
```

```
                new List<string>() { "Jan", "Feb", "Mar"},
                new List<string>() { "Apr", "May", "June"},
                new List<string>() { "Jul", "Aug", "Sep"},
                new List<string>() { "Oct", "Nov", "Dec" }
            };

            List<SunburstDataItem> items = new List<SunburstDataItem>();
            var yearLen = Math.Max((int)Math.Round(Math.Abs(5 - Instance.rnd.NextDouble() * 10)),
3);
            int currentYear = DateTime.Now.Year;
            for (int i = yearLen; i > 0; i--)
            {
                years.Add((currentYear - i).ToString());
            }
            var quarterAdded = false;

            foreach (string y in years)
            {
                var i = years.IndexOf(y);
                var addQuarter = Instance.rnd.NextDouble() > 0.5;
                if (!quarterAdded && i == years.Count - 1)
                {
                    addQuarter = true;
                }
                var year = new SunburstDataItem() { Year = y };
                if (addQuarter)
                {
                    quarterAdded = true;

                    foreach (List<string> q in times)
                    {
                        var addMonth = Instance.rnd.NextDouble() > 0.5;
                        int idx = times.IndexOf(q);
                        var quar = "Q" + (idx + 1);
                        var quarters = new SunburstDataItem() { Year = y, Quarter = quar };
                        if (addMonth)
                        {
                            foreach (string m in q)
                            {
                                quarters.Items.Add(new SunburstDataItem()
                                {
                                    Year = y,
                                    Quarter = quar,
                                    Month = m,
                                    Value = rnd.Next(20, 30)
                                });
                            };
                        }
                        else
                        {
                            quarters.Value = rnd.Next(80, 100);
                        }
                        year.Items.Add(quarters);
                    };
                }
                else
                {
                    year.Value = rnd.Next(80, 100);
                }
                items.Add(year);
```

```
        };

        return items;
    }

    public static List<FlatDataItem> CreateFlatData()
    {
        Random rnd = Instance.rnd;
        List<string> years = new List<string>();
        List<List<string>> times = new List<List<string>>()
        {
            new List<string>() { "Jan", "Feb", "Mar"},
            new List<string>() { "Apr", "May", "June"},
            new List<string>() { "Jul", "Aug", "Sep"},
            new List<string>() { "Oct", "Nov", "Dec" }
        };

        List<FlatDataItem> items = new List<FlatDataItem>();
        var yearLen = Math.Max((int)Math.Round(Math.Abs(5 - rnd.NextDouble() * 10)), 3);
        int currentYear = DateTime.Now.Year;
        for (int i = yearLen; i > 0; i--)
        {
            years.Add((currentYear - i).ToString());
        }
        var quarterAdded = false;
        foreach (string y in years)
        {

            var i = years.IndexOf(y);
            var addQuarter = rnd.NextDouble() > 0.5;
            if (!quarterAdded && i == years.Count - 1)
            {
                addQuarter = true;
            }
            if (addQuarter)
            {
                quarterAdded = true;
                foreach (List<string> q in times)
                {
                    var addMonth = rnd.NextDouble() > 0.5;
                    int idx = times.IndexOf(q);
                    var quar = "Q" + (idx + 1);
                    if (addMonth)
                    {
                        foreach (string m in q)
                        {
                            items.Add(new FlatDataItem()
                            {
                                Year = y,
                                Quarter = quar,
                                Month = m,
                                Value = rnd.Next(30, 40)
                            });
                        };
                    }
                    else
                    {
                        items.Add(new FlatDataItem()
                        {
                            Year = y,
```

```
                        Quarter = quar,
                        Value = rnd.Next(80, 100)
                    });
                }
            };
        }
        else
        {
            items.Add(new FlatDataItem()
            {
                Year = y.ToString(),
                Value = rnd.Next(80, 100)
            });
        }
    };

    return items;
}

public static C1CollectionView<Item> CreateGroupCVData()
{
    var data = new List<Item>();
    var quarters = new string[] { "Q1", "Q2", "Q3", "Q4" };
    var months = new[]
    {
        new { Name = "Jan", Value = 1 },
        new { Name = "Feb", Value = 2 },
        new { Name = "Mar", Value = 3 },
        new { Name = "Apr", Value = 4 },
        new { Name = "May", Value = 5 },
        new { Name = "June", Value = 6 },
        new { Name = "Jul", Value = 7 },
        new { Name = "Aug", Value = 8 },
        new { Name = "Sep", Value = 9 },
        new { Name = "Oct", Value = 10 },
        new { Name = "Nov", Value = 11 },
        new { Name = "Dec", Value = 12 }
    };
    var year = DateTime.Now.Year;
    int yearLen, i, len = 100;
    var years = new List<int>();
    yearLen = 3;

    for (i = yearLen; i > 0; i--)
    {
        years.Add(year - i);
    }

    int y, q, m;
    for (i = 0; i < len; i++)
    {
        y = (int)Math.Floor(Instance.rnd.NextDouble() * yearLen);
        q = (int)Math.Floor(Instance.rnd.NextDouble() * 4);
        m = (int)Math.Floor(Instance.rnd.NextDouble() * 3);

        data.Add(new Item()
        {
            Year = years[y],
            Quarter = quarters[q],
            MonthName = months[q].Name,
```

```
                    MonthValue = months[q].Value,
                    Value = Math.Round(Instance.rnd.NextDouble() * 100)
                });
            }
            var cv = new C1CollectionView<Item>(data);

            //Sort cannot work synchronize with group in current CollectionView
            SortDescription yearSortDescription = new SortDescription("Year",
SortDirection.Ascending);
            SortDescription quarterSortDescription = new SortDescription("Quarter",
SortDirection.Ascending);
            SortDescription monthSortDescription = new SortDescription("MonthValue",
SortDirection.Ascending);
            SortDescription[] sortDescriptions = new SortDescription[] { yearSortDescription,
quarterSortDescription, monthSortDescription };
            cv.SortAsync(sortDescriptions);

            GroupDescription yearGroupDescription = new GroupDescription("Year");
            GroupDescription quarterGroupDescription = new GroupDescription("Quarter");
            GroupDescription monthGroupDescription = new GroupDescription("MonthName");
            GroupDescription[] groupDescriptions = new GroupDescription[] { yearGroupDescription,
quarterGroupDescription, monthGroupDescription };
            //   cv.GroupAsync(groupDescriptions);

            return cv;
        }
    }

    public class FlatDataItem
    {
        public string Year { get; set; }
        public string Quarter { get; set; }
        public string Month { get; set; }
        public double Value { get; set; }
    }
    public class SunburstDataItem
    {
        List<SunburstDataItem> _items;

        public string Year { get; set; }
        public string Quarter { get; set; }
        public string Month { get; set; }
        public double Value { get; set; }
        public List<SunburstDataItem> Items
        {
            get
            {
                if (_items == null)
                {
                    _items = new List<SunburstDataItem>();
                }

                return _items;
            }
        }
    }

    public class Item
    {
        public int Year { get; set; }
```

```
        public string Quarter { get; set; }
        public string MonthName { get; set; }
        public int MonthValue { get; set; }
        public double Value { get; set; }
    }
```

Back to Top

Step 2: Add a SunBurst Chart control

Complete the following steps to initialize a Sunburst Chart 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.Chart;
```

2. Instantiate a Sunburst Chart control in a new method GetChartControl( ).

C#

```
public static C1Sunburst GetChartControl()
        {
            C1.Xamarin.Forms.Chart.C1Sunburst c1Sunburst = new
C1.Xamarin.Forms.Chart.C1Sunburst();
            c1Sunburst.ItemsSource = DataService.CreateFlatData();
            c1Sunburst.Binding = "Value";
            c1Sunburst.BindingName = "Year,Quarter,Month";
            PieDataLabel pieDataLabel = new PieDataLabel();
            pieDataLabel.Content = "{}{name}";
            return c1Sunburst;
        }
```

In XAML

1. Add a new Content 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"
xmlns:c1="clr-namespace:C1.Xamarin.Forms.Chart;assembly=C1.Xamarin.Forms.Chart"
x:Class="SunBurst.MainPage">
```

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

XAML

```
<Grid Margin="10">
        <c1:C1Sunburst x:Name="sunburst"  Binding="Value" BindingName="Year,Quarter,Month"
ToolTipContent="{}{name}
{y}" LegendPosition="Bottom">
            <c1:C1Sunburst.DataLabel>
                <c1:PieDataLabel Position="Center" Content="{}{name}">
                    <c1:PieDataLabel.Style>
                        <c1:ChartStyle StrokeThickness="0"/>
                    </c1:PieDataLabel.Style>
                </c1:PieDataLabel>
            </c1:C1Sunburst.DataLabel>
        </c1:C1Sunburst>
    </Grid>
</ContentPage>
```

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 **ItemSource** property of Sunburst Chart to **DataService.CreateFlatData();**

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

```
C#
public QuickStart()
{
     InitializeComponent();
     this.sunburst.ItemsSource = DataService.CreateFlatData();
}
```

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 Sunburst Chart 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 Sunburst Chart 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 Content Page:** In the class constructor `App()`, set the Content 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.FlexChartRenderer.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.Chart.Platform.UWP.FlexChartRenderer.Init();
```

     4. (Optional) In case you compile your UWP application in **Release** mode, you need to explicitly add the following code to the **OnLaunched** method in your **App.xaml.cs** to include the correct assemblies within your application.

```
C#
var assembliesToInclude = new List<Assembly>();
assembliesToInclude.Add(typeof(C1.Xamarin.Forms.Chart.Platform.UWP.FlexChartRenderer)
.GetTypeInfo().Assembly);
assembliesToInclude.Add(typeof(C1.UWP.Chart.FlexChart).GetTypeInfo().Assembly);
Xamarin.Forms.Forms.Init(e, assembliesToInclude);
```

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

Back to Top

Features

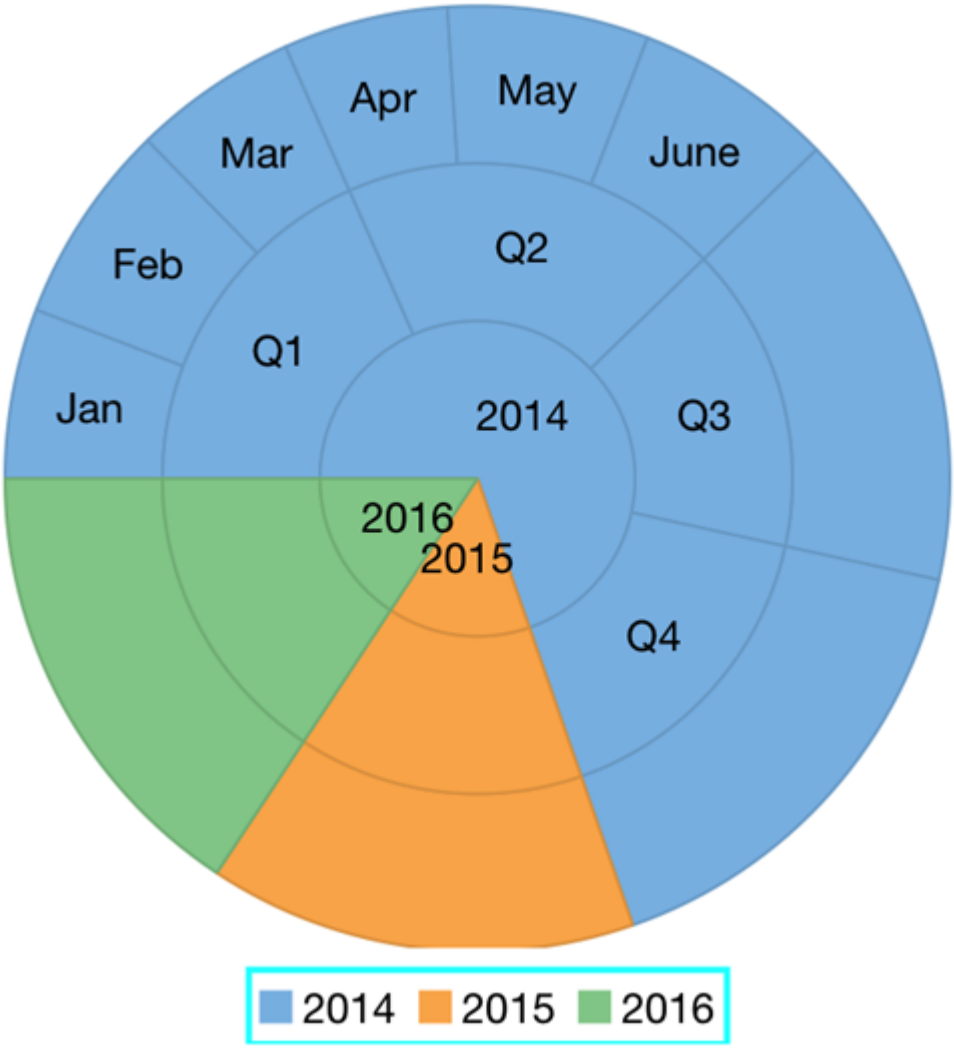
Legend

You can specify the position where you want to display the legend using the **LegendPosition** property of the Sunburst chart. Legend helps in displaying the series of a chart with a predefined symbol and name of the series.

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 Sunburst 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 Sunburst Chart control.

> 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 Sunburst Chart appears after applying the properties.



The following code example demonstrates how to set these properties in C# and XAML. This examples uses the sample created in the Quick Start section.

In Code

C#

```
sunburst.LegendPosition = ChartPositionType.Bottom;
sunburst.LegendOrientation = Orientation.Horizontal;
sunburst.LegendStyle.Stroke = Color.Aqua;
sunburst.LegendStyle.StrokeThickness = 2;
```

In XAML

XAML

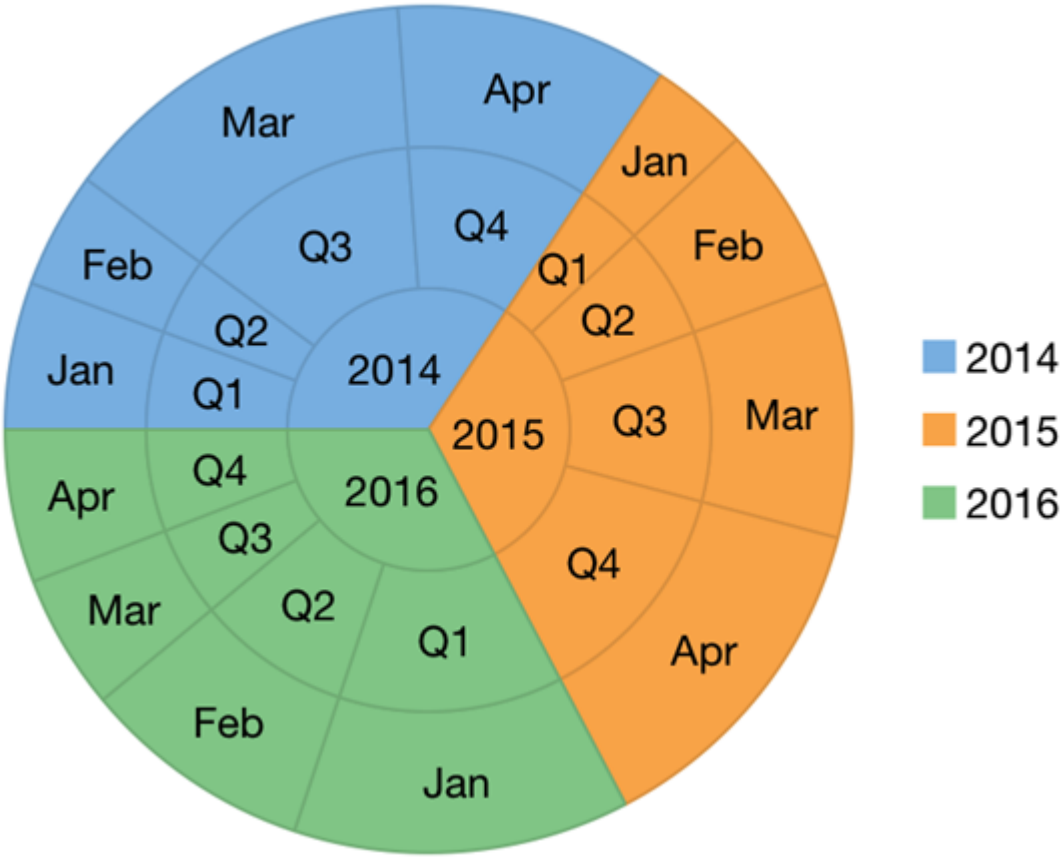
```
<c1:C1Sunburst x:Name="sunburst"  Binding="Value" BindingName="Year,Quarter,Month"
LegendPosition="Bottom"
LegendOrientation="Horizontal">
<c1:C1Sunburst.LegendStyle>
<c1:ChartStyle Stroke="Aqua" StrokeThickness="4"></c1:ChartStyle>
```

```
</c1:C1Sunburst.LegendStyle>
```

Grouping

The CollectionView class supports grouping through the ICollectionView interface, similar to the one in .NET. To enable grouping, add one or more GroupDescription objects to the **CollectionView.GroupDescription** property. GroupDescription objects are flexible, allowing you to group data based on value or on grouping functions. The below code uses the **GroupChanged** event and **SortChanged** event for performing grouping operation in Sunburst Chart.

The image below shows how to set Grouping in Sunburst Chart control.



The following code example demonstrates how to set these properties using C# and XAML. This examples uses the data created in the Quick Start section.

In XAML

XAML

```
<Grid Margin="10">
        <c1:C1Sunburst x:Name="sunburst"  Binding="Value">
            <c1:C1Sunburst.DataLabel>
                <c1:PieDataLabel Position="Center" Content="{}{name}">
                    <c1:PieDataLabel.Style>
                        <c1:ChartStyle StrokeThickness="0"/>
                    </c1:PieDataLabel.Style>
                </c1:PieDataLabel>
```

```
            </c1:C1Sunburst.DataLabel>
        </c1:C1Sunburst>
</Grid>
```

In Code

C#

```
public partial class MainPage : ContentPage
            {
        private static C1CollectionView<Item> cv;
        public MainPage()
        {
            InitializeComponent();
            Title = "SunBurst Chart";
            DataService model = new DataService();
            cv = DataService.CreateGroupCVData();

            cv.GroupChanged += View_GroupChanged;

            cv.SortChanged += Cv_SortChanged;
        }
        private void Cv_SortChanged(object sender, System.EventArgs e)
        {
            GroupDescription yearGroupDescription = new GroupDescription("Year");
            GroupDescription quarterGroupDescription = new
GroupDescription("Quarter");
            GroupDescription monthGroupDescription = new
GroupDescription("MonthName");
            GroupDescription[] groupDescriptions = new GroupDescription[] {
yearGroupDescription, quarterGroupDescription, monthGroupDescription };
            cv.GroupAsync(groupDescriptions);
        }

        private void View_GroupChanged(object sender, System.EventArgs e)
        {
            this.sunburst.ItemsSource = cv;
            this.sunburst.Invalidate();
        }
    }
```

Selection

The Sunburst Chart control allows you to select data points by clicking or touching a sunburst slice. Use the **SelectionMode** property to specify whether you want to allow selection by data point or no selection at all(default). The three different options provided are as follows:

- **None**: Does not select any element.
- **Point**: Highlights the pie slice that the user clicks.
- **Series**: Highlights the entire pie.

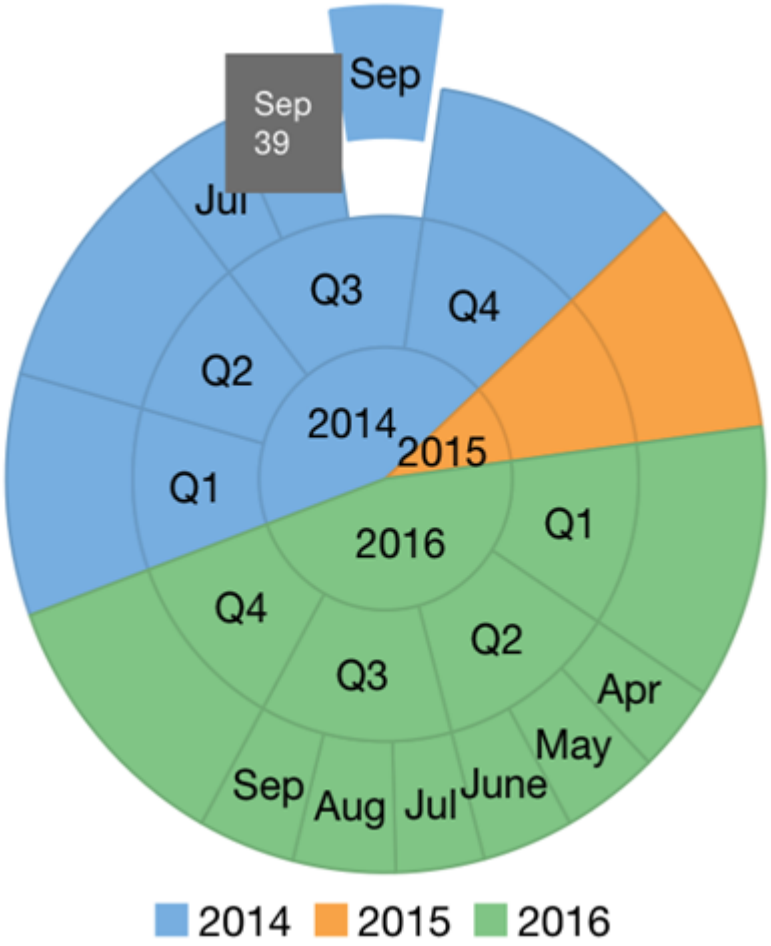
When the SelectionMode is set to **Point**, you can change the position of the selected sunburst slice by setting the

SelectedItemPosition property. Also, you can set the **SelectedItemOffset** property to move the selected sunburst slice away from the center. Setting the SelctionMode property to Point causes the Sunburst to update the selection property when the user clicks or touch on a sunburst slice.

The Sunburst offers two additional properties to customize the selection:

- **SelectedItemOffset**: Specifies the offset of the selected sunburst slice from the center of the control.
- **SelectedItemPosition**: Specifies the position of the selected sunburst slice. The available options are Top, Bottom, Left, Right, and None (default).

The image below show how the Sunburst chart appears after you set the **SelectionMode** property.



The following code example demonstrates how to set these properties in C# and XAML. This examples uses the sample created in the Quick Start section.

In Code

C#

```
this.sunburst.SelectionMode = ChartSelectionModeType.Point;
sunburst.SelectedItemPosition = ChartPositionType.Top;
sunburst.SelectedItemOffset = 0.2;
```

In XAML

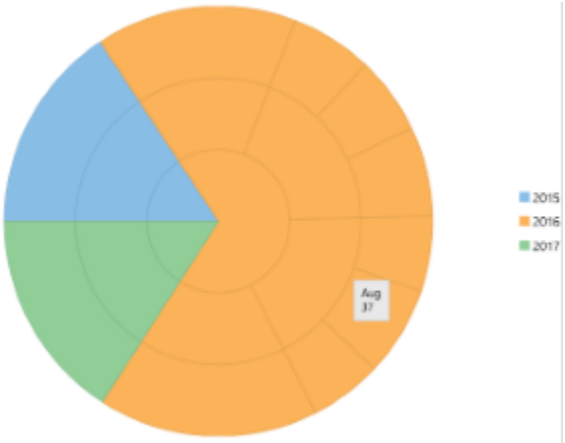
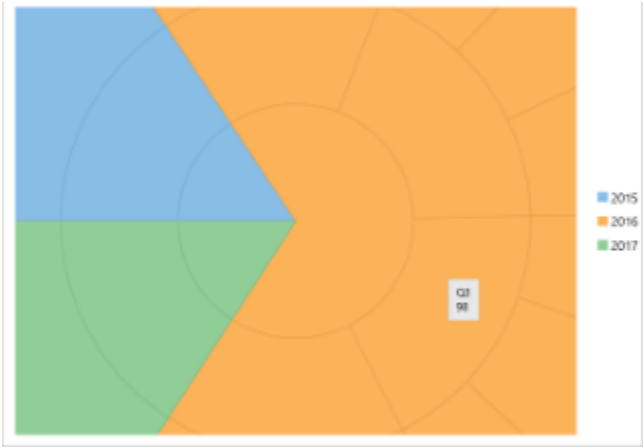
XAML

```
<c1:C1Sunburst x:Name="sunburst"  Binding="Value" BindingName="Year,Quarter,Month"
ToolTipContent="{}{name}
{y}" LegendPosition="Bottom" SelectionMode="Point" SelectedItemPosition="Top"
SelectedItemOffset="0.2">
```

Zooming and Panning

Zooming can be performed in the Sunburst chart using ZoomBehavior class. To implement zooming, you need to create an object of the ZoomBehavior class available in the C1.Xamarin.Forms.Chart.Interaction namespace and pass it as a parameter to the Add method. This method adds zoom behavior to the behavior collection by accessing it through Behaviors property of the ChartBase class.

The image below shows how Sunburst appears on zooming.

Before Zooming	After Zooming
	

The following code examples demonstrate how to implement zooming in C#. These examples use the sample created in the Quick Start section.

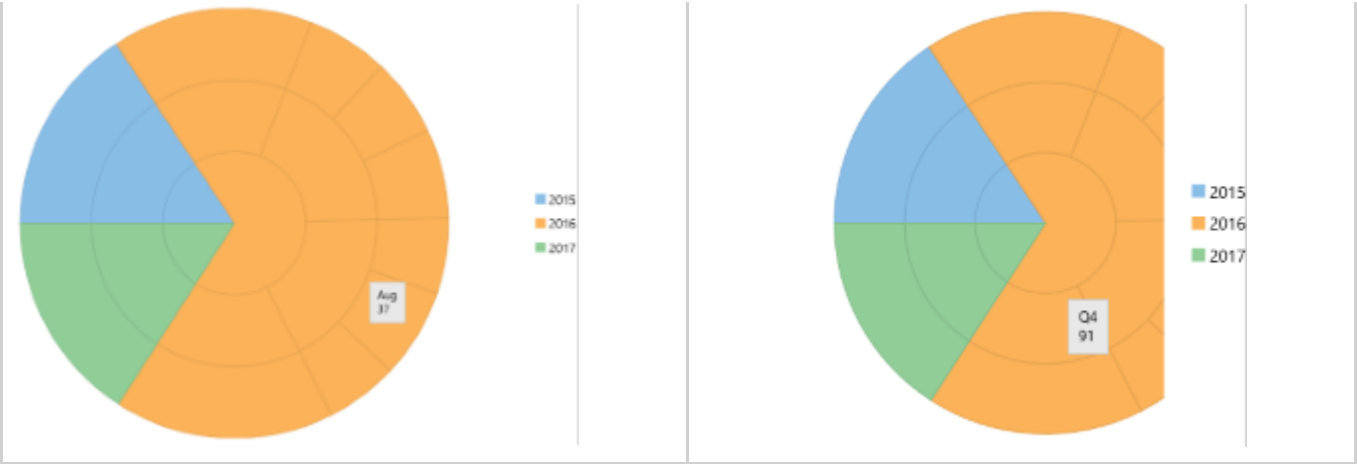
C#

```
ZoomBehavior z = new ZoomBehavior();
sunburst.Behaviors.Add(z);
```

Similarly, panning can be implemented in the Sunburst chart by creating an object of TranslateBehavior class available in the C1.Xamarin.Forms.Chart.Interaction namespace and passing it as a parameter to the Add method. This method adds translation behavior to the behavior collection by accessing it through Behaviors property of the ChartBase class.

The image below shows how Sunburst appears on panning.

Before Panning	After Panning



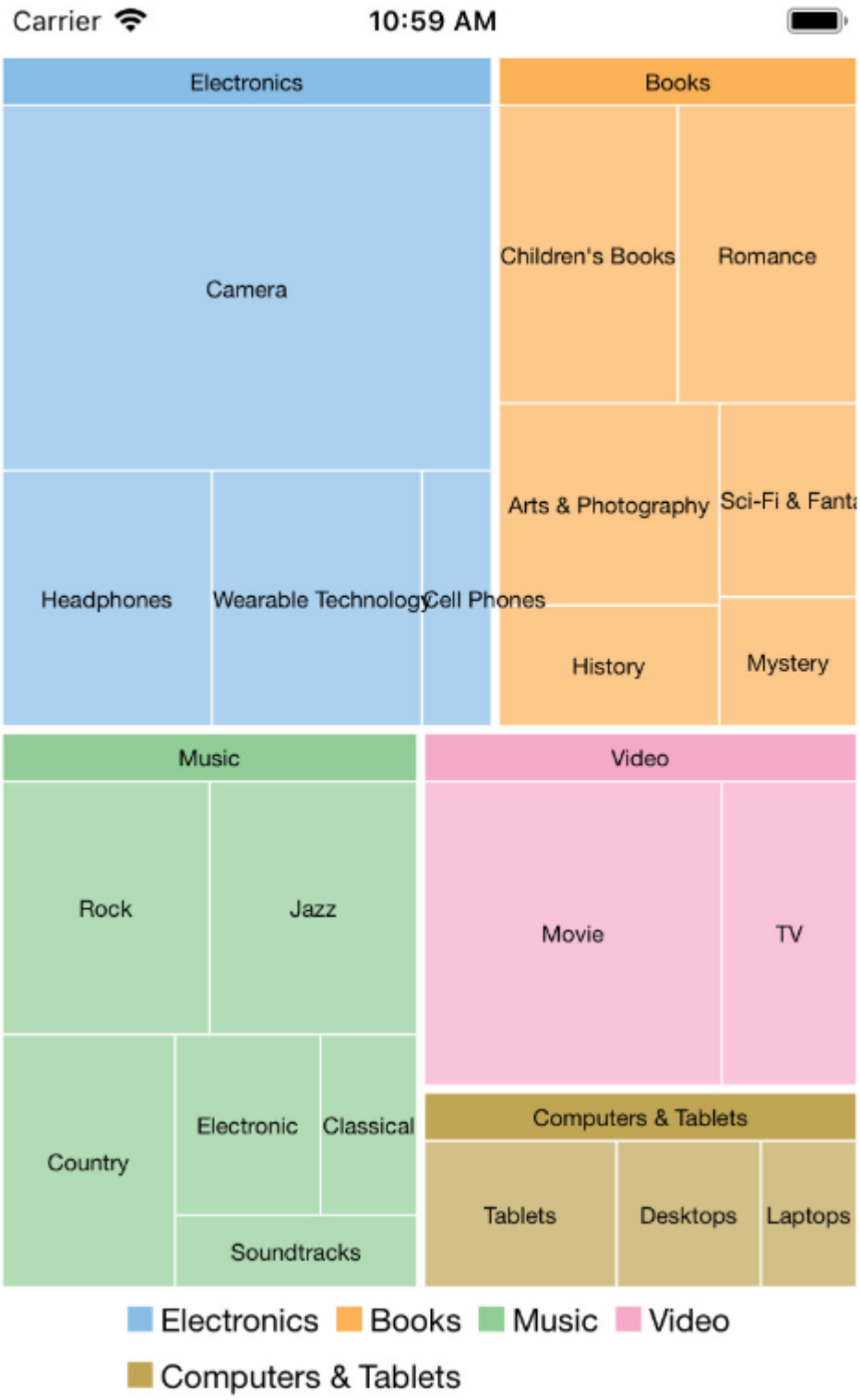
The following code examples demonstrate how to implement panning in C#. These examples use the sample created in the Quick Start section.

C#

```
TranslateBehavior t = new TranslateBehavior();
sunburst.Behaviors.Add(t);
```

TreeMap

The TreeMap control display hierarchical data as a set of nested rectangles. Hierarchical data are useful in varied walks of life and setups, be it family tree, programming, organization structure, or directories. Visualizing such a data and spotting information in them is a difficult task, especially if the data is huge. TreeMap control enables visualization of hierarchical data as nested rectangles on a limited space. It is useful in having a quick glimpse of patterns in large data and comparing proportions.



TreeMap control supports data binding to show hierarchy, and allows user to drill down the data further to numerous levels for detailed analysis. The control can be customized to display data in horizontal, vertical, and squarified layouts of constituting rectangles.

Key Features

TreeMap provides many different features that enable the developers to build intuitive and professional-looking applications. The main features of TreeMap are as follows:

- **Multiple Layouts**
  TreeMap supports multiple display arrangements, where the tree branches can be shown as squares, horizontal rectangles or vertical rectangles.

- **Customize Appearance**
  TreeMap enables users to stylize the control and modify its appearance as per their preference. A set of varied color palettes are available to clearly display categories in a TreeMap.

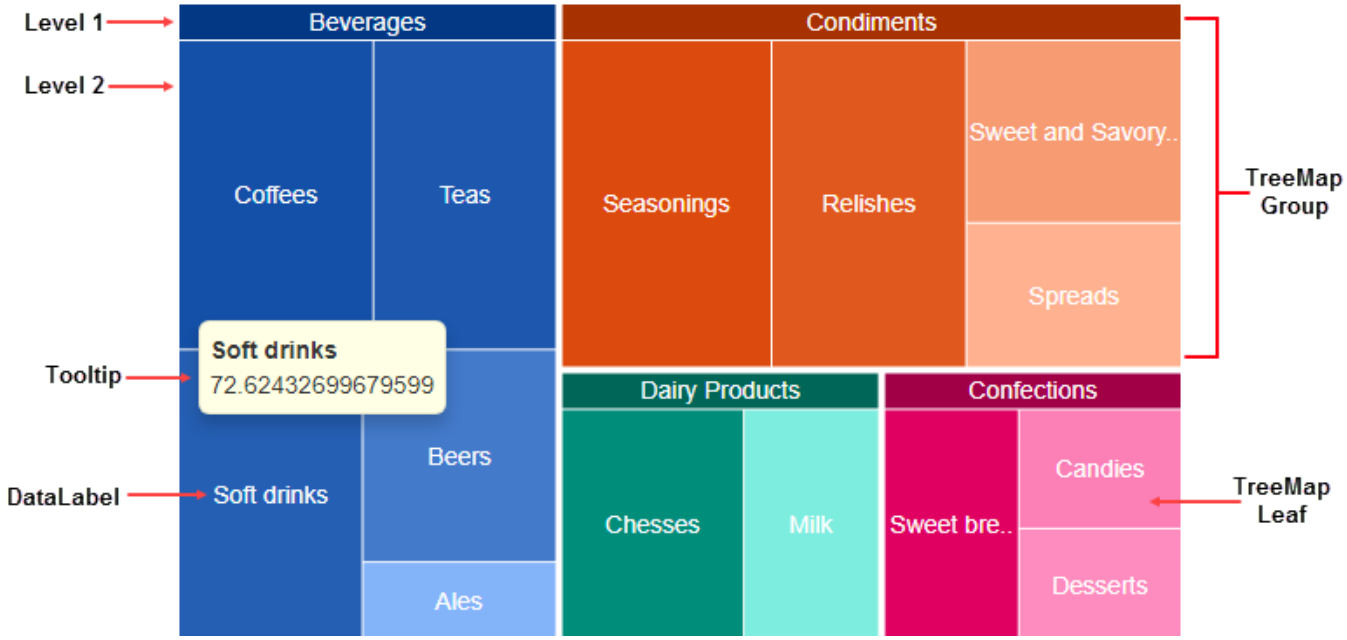
- **Custom Hierarchical Levels**
  TreeMap enables users to vary the depth of data to be visualized and further drill down (or reverse drill down) the data for analysis and comparison.

- **Space Utilization**
  TreeMap is ideal for compact display and visualization of huge data. The nested rectangles and groups constituting the TreeMap adjust their size to fit the display area.

Elements

The TreeMap control is composed of rectangles, representing individual data items, which are grouped into categories, to represent the hierarchical nature of data. The individual data items which make group are known as leaf nodes. The sizes of these nodes are proportional to the data they represent.

The following image exhibits main elements of TreeMap control.

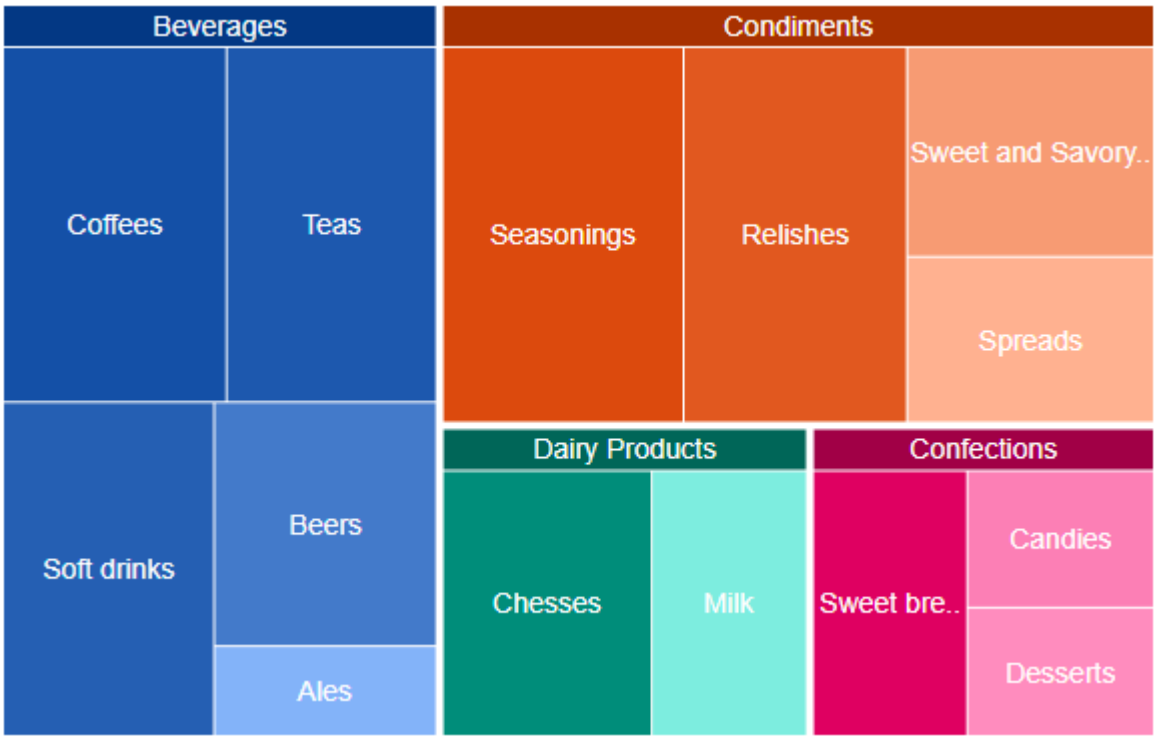


Layouts

TreeMap enables its data items and groups, represented as rectangles, to be displayed in a variety of arrangements. The tree map rectangles can be arranged into squarified, horizontal, and vertical layouts. To set the desired tree map layout, you need to use Type property of **TreeMap** class, which takes the value fromTreeMapType Enum. The default layout of the TreeMap chart control is squarified.

Squarified

The squarified layout tries to arrange the tree map rectangles (data items and groups) as approximate squares. This layout makes it easier to make comparisons and point patterns, as the accuracy of presentation is enhanced in squarified arrangement. This layout is very useful for large data sets.

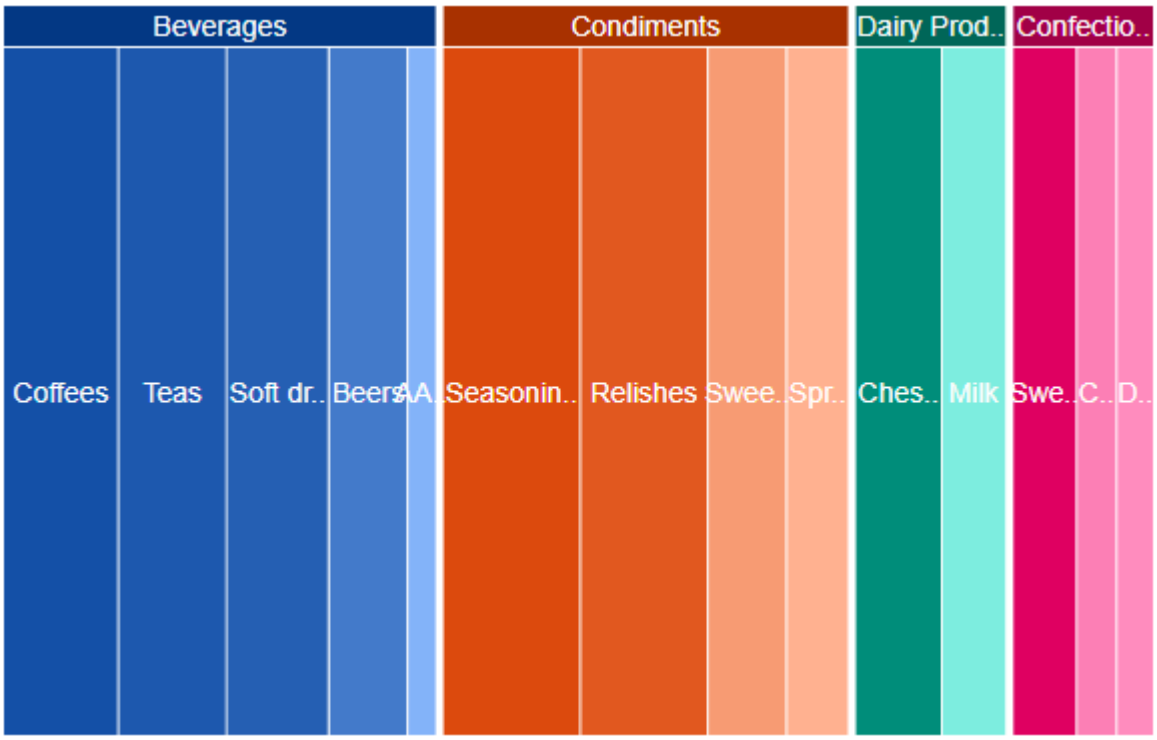


Horizontal

The horizontal layout stacks the tree map rectangles one over the other as rows. Here the width of the rectangles is greater than their height.

Vertical

The vertical layout arranges the tree map rectangles adjacent to each other as columns. Here the height of the rectangles is greater than their width.



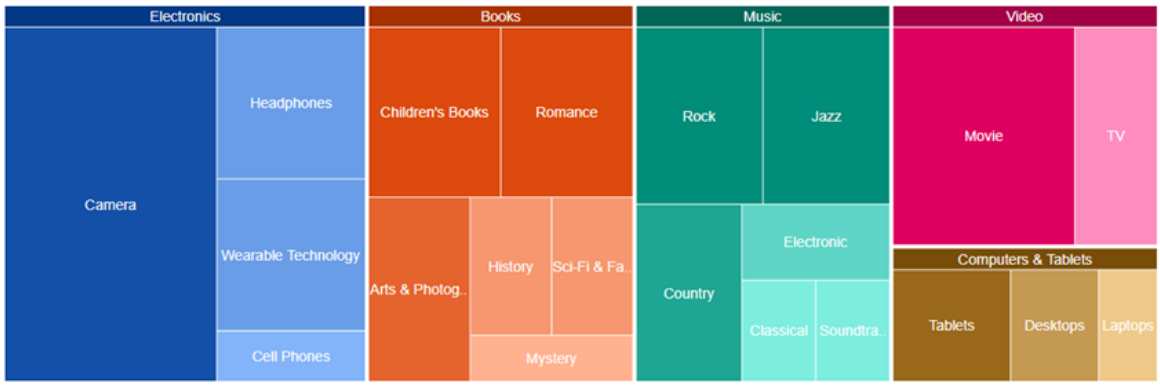
QuickStart

This section describes how to add a TreeMap control to your portable or shared app and add data to it. For more 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 TreeMap**
- **Step 2: Add a TreeMap Chart control**
- **Step 3: Run the Project**

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



Step 1: Create a Data source for TreeMap Chart

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

C#

```
public class ThingSale
    {
        private static List<string> Categories = new List<string> { "Music", "Video", "Books",
"Electronics", "Computers & Tablets" };
        private static Dictionary<string, List<string>> AllCategories = new Dictionary<string,
List<string>>();
        public string Category { get; set; }
        public double? Sales { get; set; }
        public List<ThingSale> Items { get; set; }

        public static void EnsureInitAllCategories()
        {
            if (AllCategories.Count > 0)
            {
                return;
            }
            AllCategories.Add("Music", new List<string> { "Country", "Rock", "Classical",
"Soundtracks", "Jazz", "Electronic" });
            AllCategories.Add("Country", new List<string> { "Classic Country", "Cowboy Country",
"Outlaw Country", "Western Swing", "Roadhouse Country" });
            AllCategories.Add("Rock", new List<string> { "Hard Rock", "Blues Rock", "Funk Rock",
"Rap Rock", "Guitar Rock", "Progressive Rock" });
            AllCategories.Add("Classical", new List<string> { "Symphonies", "Chamber Music" });
            AllCategories.Add("Soundtracks", new List<string> { "Movie Soundtracks", "Musical
Soundtracks" });
            AllCategories.Add("Jazz", new List<string> { "Smooth Jazz", "Vocal Jazz", "Jazz Fusion",
"Swing Jazz", "Cool Jazz", "Traditional Jazz" });
            AllCategories.Add("Electronic", new List<string> { "Electronica", "Disco", "House" });

            AllCategories.Add("Video", new List<string> { "Movie", "TV" });
            AllCategories.Add("Movie", new List<string> { "Kid & Family", "Action & Adventure",
"Animation", "Comedy", "Drama", "Romance" });
            AllCategories.Add("TV", new List<string> { "Science Fiction", "Documentary", "Fantasy",
"Military & War", "Horror" });

            AllCategories.Add("Books", new List<string> { "Arts & Photography", "Children's Books",
"History", "Mystery", "Romance", "Sci-Fi & Fantasy" });
            AllCategories.Add("Arts & Photography", new List<string> { "Architecture", "Graphic
Design", "Drawing", "Photography", "Performing Arts" });
            AllCategories.Add("Children's Books", new List<string> { "Beginning Readers", "Board
Books", "Chapter Books", "Coloring Books", "Picture Books", "Sound Books" });
            AllCategories.Add("History", new List<string> { "Ancient", "Medieval", "Renaissance" });
            AllCategories.Add("Mystery", new List<string> { "Thriller & Suspense", "Mysteries" });
            AllCategories.Add("Romance", new List<string> { "Action & Adventure", "Holidays",
"Romantic Comedy", "Romantic Suspense", "Western", "Historical" });
            AllCategories.Add("Sci-Fi & Fantasy", new List<string> { "Fantasy", "Gaming", "Science
Fiction" });

            AllCategories.Add("Electronics", new List<string> { "Camera", "Headphones", "Cell
Phones", "Wearable Technology" });
            AllCategories.Add("Camera", new List<string> { "Digital Cameras", "Film Photography",
"Lenses", "Video", "Accessories" });
            AllCategories.Add("Headphones", new List<string> { "Earbud headphones", "Over-ear
headphones", "On-ear headphones", "Bluetooth headphones", "Noise-cancelling headphones", "Audiophile
headphones" });
            AllCategories.Add("Cell Phones", new List<string> { "Cell Phone", "Accessories" });
            AllCategories.Add("Accessoriess", new List<string> { "Batteries", "Bluetooth Headsets",
"Bluetooth Speakers", "Chargers", "Screen Protectors" });
            AllCategories.Add("Wearable Technology", new List<string> { "Activity Trackers", "Smart
```

```
Watches", "Sports & GPS Watches", "Virtual Reality Headsets", "Wearable Cameras", "Smart Glasses"
});

            AllCategories.Add("Computers & Tablets", new List<string> { "Desktops", "Laptops",
"Tablets" });
            AllCategories.Add("Desktops", new List<string> { "All-in-ones", "Minis", "Towers" });
            AllCategories.Add("Laptops", new List<string> { "2 in 1 laptops", "Traditional laptops"
});
            AllCategories.Add("Tablets", new List<string> { "IOS", "Andriod", "Fire OS", "Windows"
});
        }
        public static IEnumerable<ThingSale> GetData()
        {
            EnsureInitAllCategories();
            var result = new List<ThingSale>();
            Categories.ForEach(cat =>
            {
                result.Add(Create(cat));
            });

            return result;
        }

        private static ThingSale Create(string category)
        {
            var rand = new Random(0);
            var item = new ThingSale { Category = category };
            if (!AllCategories.ContainsKey(category))
            {
                item.Sales = rand.NextDouble() * 100;
            }
            else
            {
                item.Items = new List<ThingSale>();
                AllCategories[category].ForEach(subCat =>
                {
                    item.Items.Add(Create(subCat));
                });
            }
            return item;
        }
    }
```

Back to Top

Step 2: Add a TreeMap Chart control

Complete the following steps to initialize a TreeMap 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.Chart;
   ```

2. Instantiate a TreeMap Chart control in the MainPage( ) method.

   C#

   ```
   public partial class MainPage : ContentPage
       {
           public MainPage()
   ```

```
        {
            InitializeComponent();
            treeMap.ChartType = C1.Xamarin.Forms.Chart.TreeMapType.Squarified;
            treeMap.Binding = "Sales";
            treeMap.BindingName = "Category";
            treeMap.MaxDepth = 2;
            treeMap.ShowTooltip = true;
            treeMap.ChildItemsPath = "Items";
            treeMap.ItemsSource = ThingSale.GetData();
            treeMap.DataLabel = new C1.Xamarin.Forms.Chart.ChartDataLabel() {Content = "{name}
{type}", Position = C1.Xamarin.Forms.Chart.ChartLabelPosition.Center};
        }
    }
```

In XAML

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

XAML

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
xmlns:local="clr-namespace:TreeMap_QS"
xmlns:c1="clr-namespace:C1.Xamarin.Forms.Chart;assembly=C1.Xamarin.Forms.Chart"
x:Class="TreeMap_QS.MainPage">
```

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

XAML

```
<c1:C1TreeMap x:Name="treeMap" Binding="Sales" BindingName="Category" MaxDepth="2"
ChildItemsPath="Items">
        <c1:C1TreeMap.DataLabel>
            <c1:ChartDataLabel Content="{}{type}">
                <c1:ChartDataLabel.Style>
                    <c1:ChartStyle FontSize="10" />
                </c1:ChartDataLabel.Style>
            </c1:ChartDataLabel>
        </c1:C1TreeMap.DataLabel>
    </c1:C1TreeMap>
</ContentPage>
```

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 **ItemSource** property of TreeMap Chart to **ThingSale.GetData();**

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

C#

```
public QuickStart()
{
     InitializeComponent();
     this.TreeMap.ItemsSource = ThingSale.GetData();
}
```

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 TreeMap Chart 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 MainPage( ) defined in the previous procedure, **Step 2: Add a TreeMap**

Control.

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

```
C#
public App()
{
InitializeComponent();
MainPage = new MainPage();
}
```

- **To return a Content Page:** In the class constructor App(), set the Content 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.FlexChartRenderer.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.Chart.Platform.UWP.FlexChartRenderer.Init();
```

     4. (Optional) In case you compile your UWP application in **Release** mode, you need to explicitly add the following code to the **OnLaunched** method in your **App.xaml.cs** to include the correct assemblies within your application.

```
C#
var assembliesToInclude = new List<Assembly>();
assembliesToInclude.Add(typeof(C1.Xamarin.Forms.Chart.Platform.UWP.FlexChartRenderer)
.GetTypeInfo().Assembly);
assembliesToInclude.Add(typeof(C1.UWP.Chart.FlexChart).GetTypeInfo().Assembly);
Xamarin.Forms.Forms.Init(e, assembliesToInclude);
```

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

Back to Top

Features

Drilldown

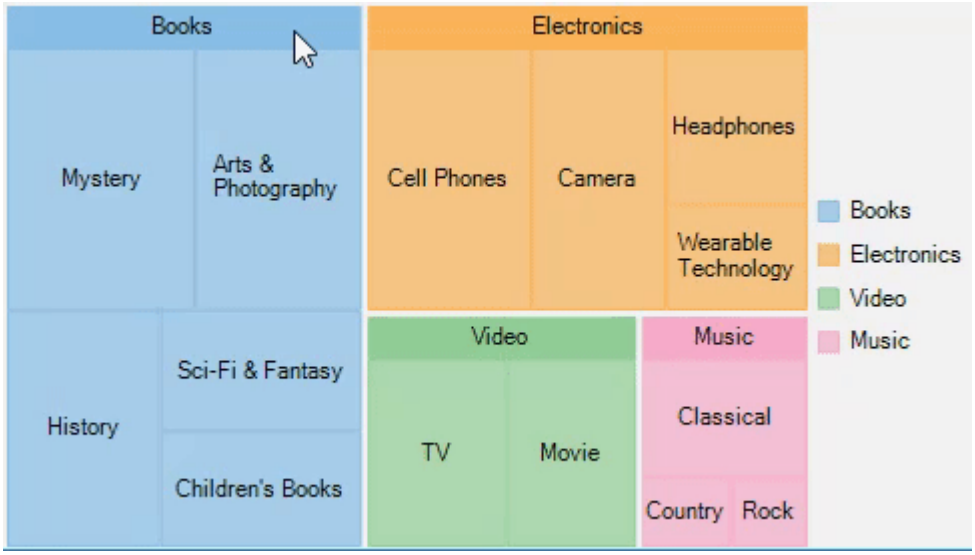
TreeMap allows drilling down the data items of its data further for detailed analysis. End users can access the lower levels in the data hierarchy by simply clicking the desired node. Whereas, to move back up in the hierarchy, users simply need to right-click in the plot area.

To implement the drilldown functionality in the TreeMap control, set the **MaxDepth** property to a value greater that

0. This property defines the levels of hierarchical data in the TreeMap control.

Note that the more levels you show the less understandable your TreeMap might become (depends on the levels' number and values they represent). In our example, we will set **MaxDepth** property to 2.

The following gif image demonstrates drilling-down by showing data points of the clicked TreeMap node.



The following code example demonstrates how to set the MaxDepth property of the TreeMap in C# to enable DrillDown. This example uses the sample created in the QuickStart section.

In Code

Drilldown.cs

```
treeMap.MaxDepth = 2;
```

In XAML

Drilldown.xaml

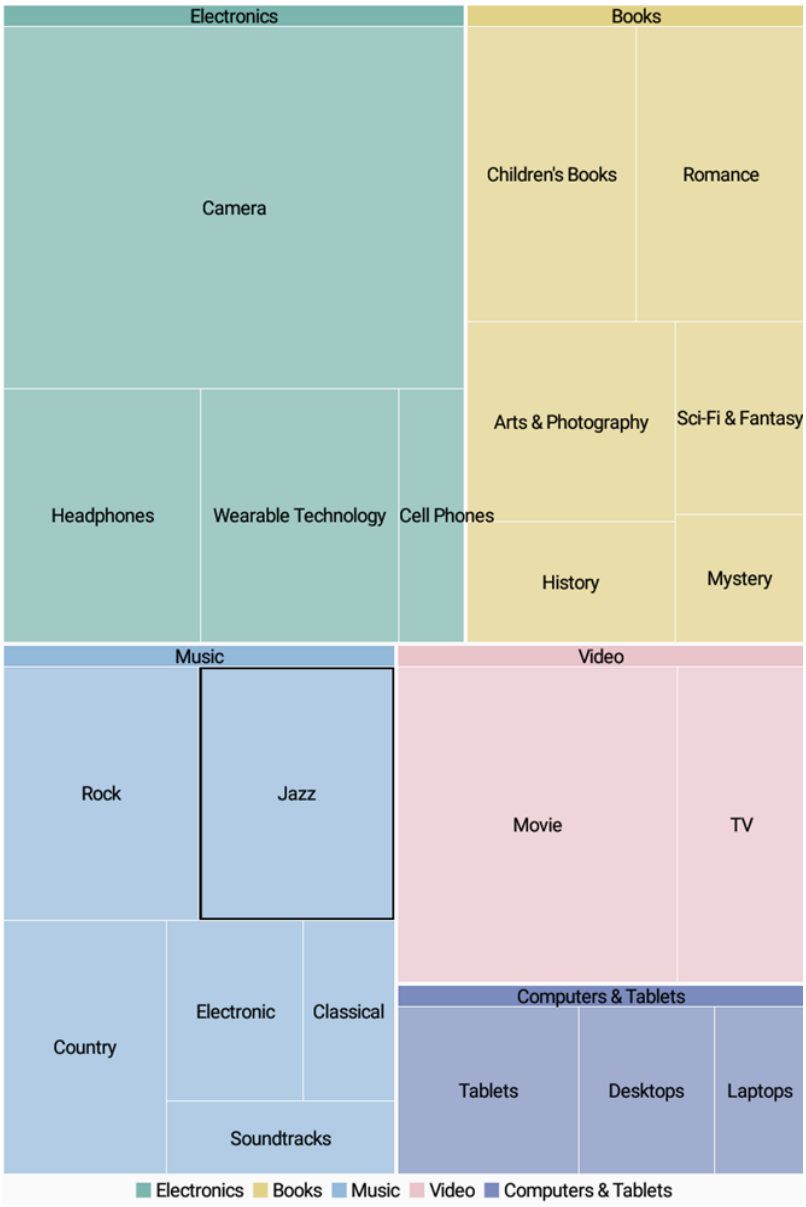
```
<c1:C1TreeMap x:Name="treeMap" Binding="Sales" BindingName="Category" MaxDepth="2"
ChildItemsPath="Items">
</c1:C1TreeMap>
```

Selection

TreeMap lets you enable selection of its data items and groups. User can select a node and draw focus on it by simply clicking it. You need to set the SelectionMode property provided by the ChartBase class to either of the following values in the ChartSelectionMode enumeration:

- **None (default)**: Selection is disabled.
- **Point**: A point is selected.

The following image illustrates default selection of a data point along with its children nodes in the hierarchy.



The following code snippet shows how to set the **SelectionMode** property for a tree map control.

In Code

Selection.cs

```
// Implement selection mode
treeMap.SelectionMode = ChartSelectionModeType.Point;
```

In XAML

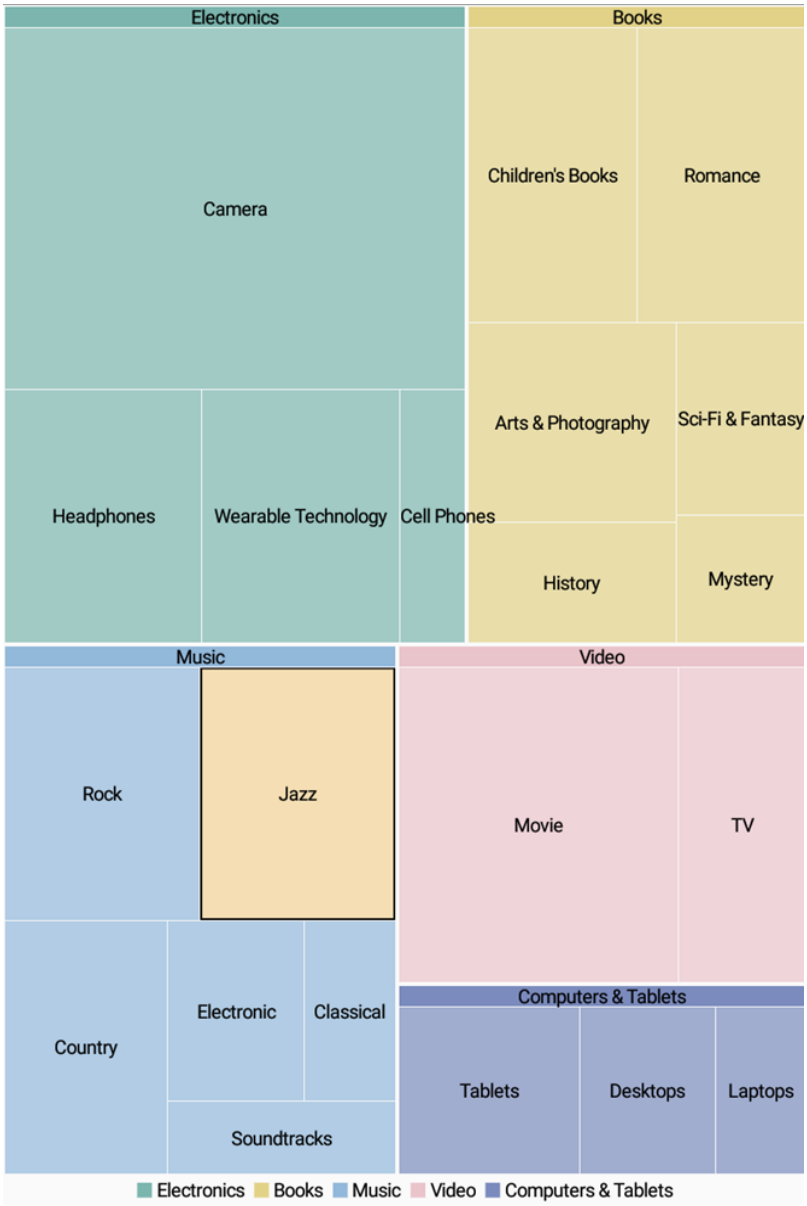
Selection.xaml

```
<c1:C1TreeMap x:Name="treeMap" Binding="Sales" BindingName="Category"
SelectionMode="Point" ChildItemsPath="Items">
</c1:C1TreeMap>
```

Customized TreeMap Selection

To customize the TreeMap selection, you can use SelectionStyle property and style the selected item as illustrated in

the following image.



The following code snippet demonstrates using of **SelectionStyle** property to change fill color of the selected TreeMap node.

In Code

Selection.cs

```
// Apply custom color to the selection
treeMap.SelectionStyle.Fill = Color.Wheat;
```

In XAML

Selection.xaml

```
<c1:C1TreeMap x:Name="treeMap" Binding="Sales" BindingName="Category"
SelectionStyle="Color.Wheat" ChildItemsPath="Items">
</c1:C1TreeMap>
```

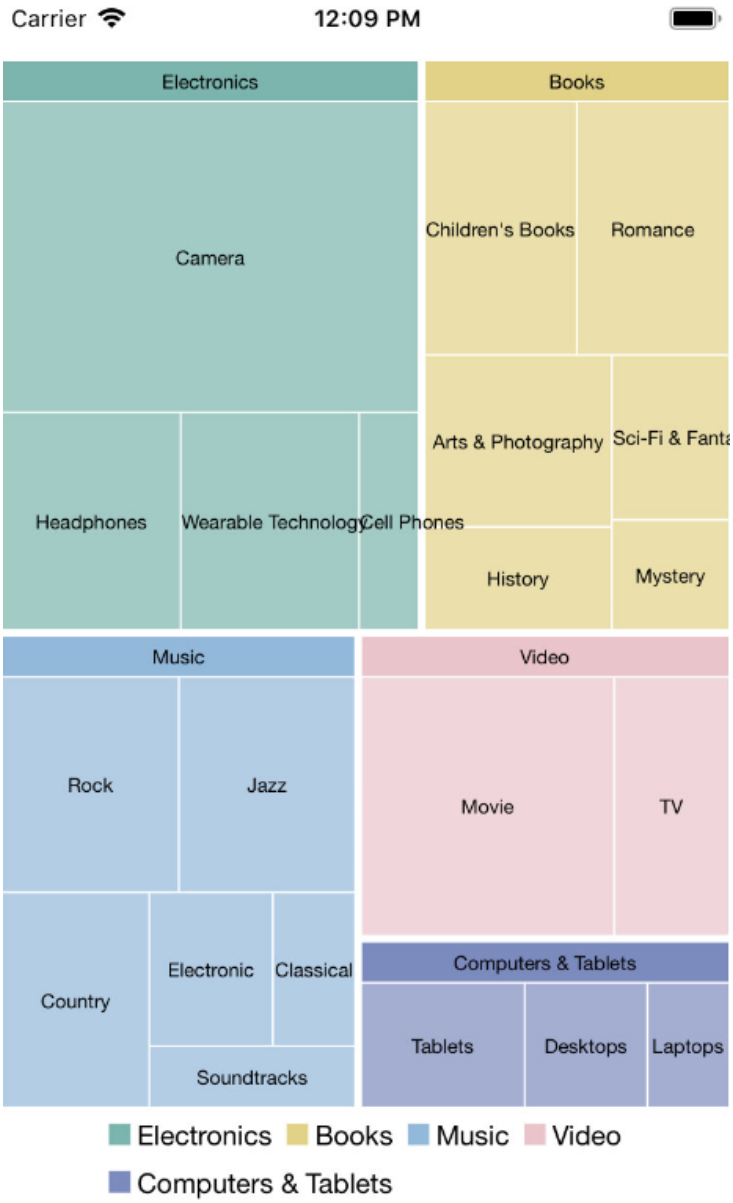
Additionally, you can customize the behavior of TreeMap selection by handling **SelectionChanged** event. Also, you

can utilize **SelectedIndex** and **SelectedItem** properties, and reuse the obtained information in your application.

Theming

The TreeMap control allows you to customize its appearance by using the **Palette** property. This property accepts value from the **Palette** enumeration provided by the **ChartBase** class.

The following image shows how a TreeMap control appears after applying a theme using the Palette property.



The following code examples demonstrate how to set **Palette** property in C#. These examples use the sample created in the Quick Start section.

In Code

Theming.cs

```
treeMap.Palette = Palette.Zen;
```

In XAML

Theming.xaml

```
<c1:C1TreeMap x:Name="t
reeMap" Binding="Sales" BindingName="Category" Palette="Zen" ChildItemsPath="Items">
</c1:C1TreeMap>
```