Chapter 7

Project 2: Tip Calculator

Introduction

In this chapter, you will be introduced to creating a Tip Calculator. A Tip Calculator is a tool or app that helps you figure out how much tip to leave after having food at a restaurant. The tip amount varies depending on the country and cultural norms. In many countries, it is customary to leave a tip of around 15% to 20% of the total bill at restaurants. However, some places may have different tipping customs, and some people may choose to leave more or less than the usual percentage based on the quality of service or personal preferences. Also, when dining in a group, it is common to split the total bill, including the tip, equally among all the individuals to ensure a fair distribution of expenses. This method allows everyone to contribute their share, making it easier to handle the bill and the tip without any confusion or inconvenience. In this chapter, you will learn how to build a Tip Calculator that will allow users to enter the bill amount, select the desired tip percentage, and the number of people who will split the bill. It will help you instantly calculate the tip amount and the split amount for each user from the total bill amount.

Structure

In this chapter, we will discuss the following topics:

* Creating the layout
* Creating the data entry section
* Applying graphical design principles in UI design
* Performing the calculation

Objectives

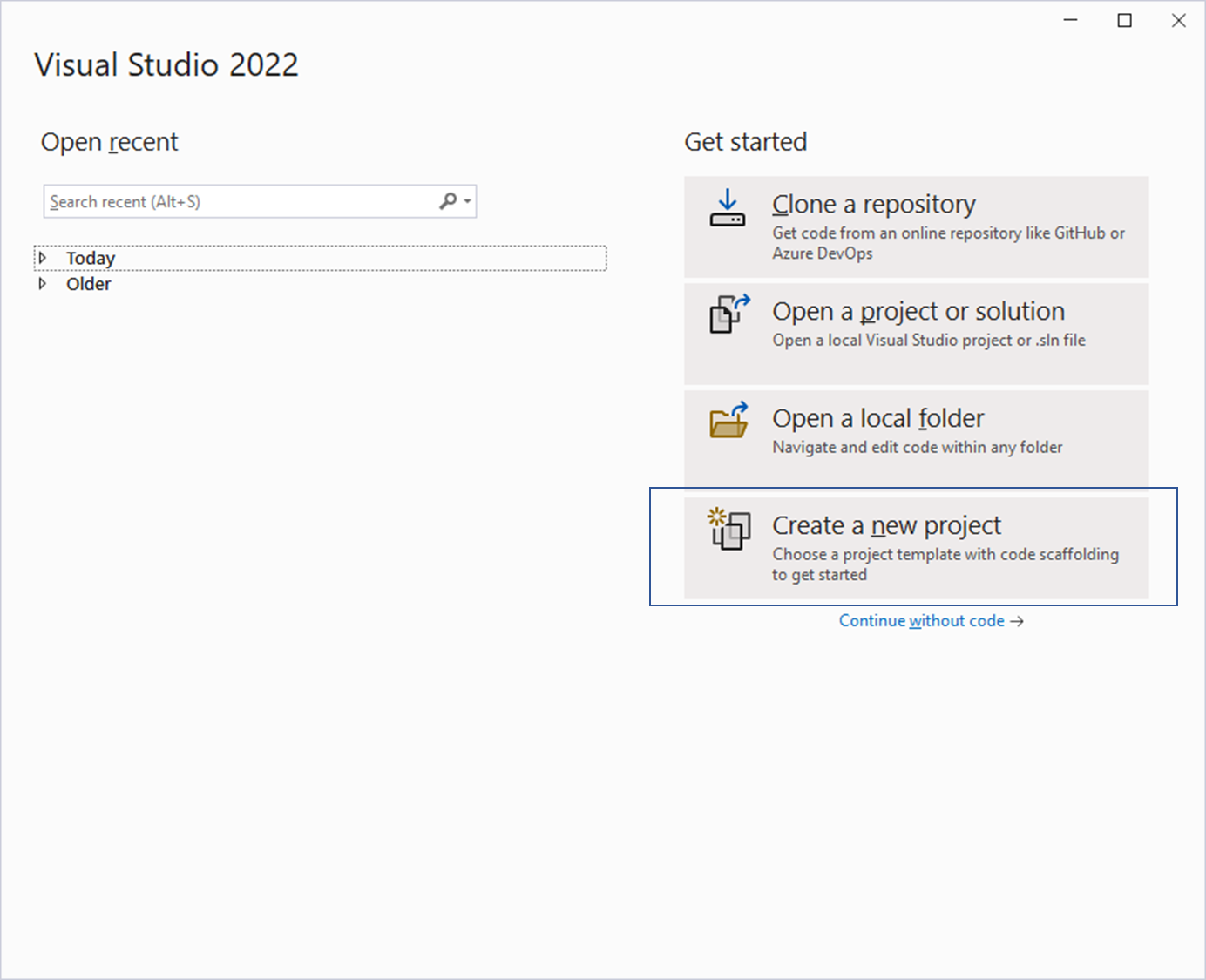
After reading this chapter, you will be able:

* To create a complex UI layout using .NET MAUI controls such as Grid, Frame, VerticalStackLayout, and HorizontalStackLayout controls.
* You will also be able to develop the UI in XAML. Further, you will understand how to work with commonly used controls, including Label, Entry, Slider, and Button.
* You will know how to implement the functionality to calculate the tip amount and split the amount in C#.

Creating the layout

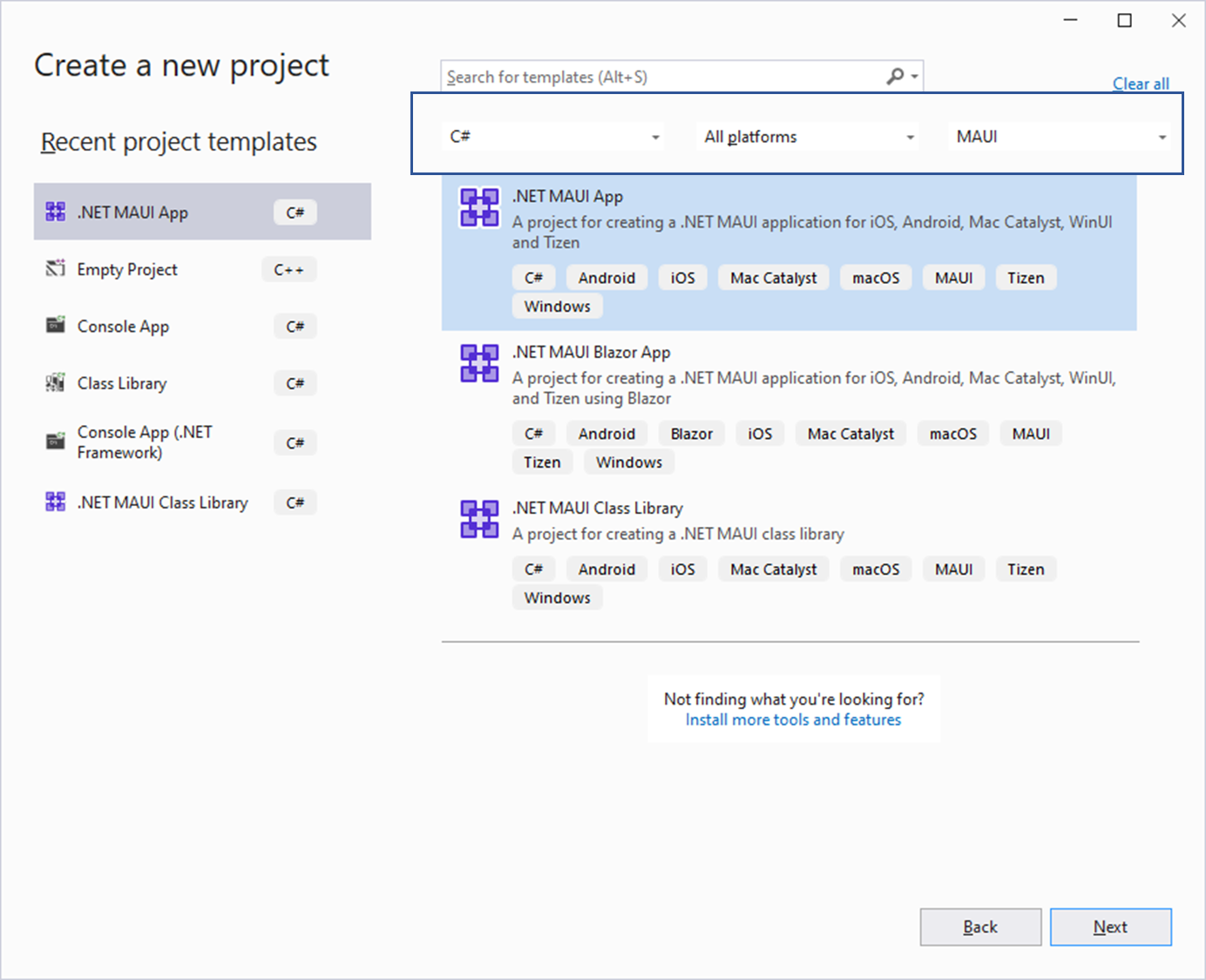
Follow these steps to create a layout for the app:

1. Start Visual Studio.
2. In the dialog that appears, select the option: **Create a new project,** as shown in *Figure 7.1*:



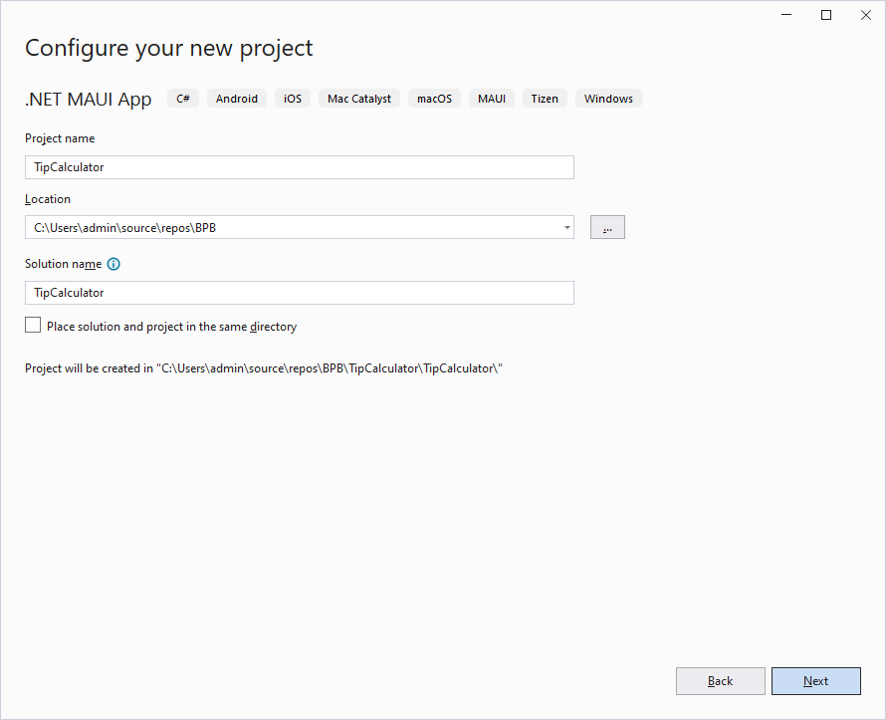
**Figure 7.1**: Creating a new project in Visual Studio

1. Select the programming language as C# and MAUI project to view the MAUI project templates, as shown in *Figure 7.2*:



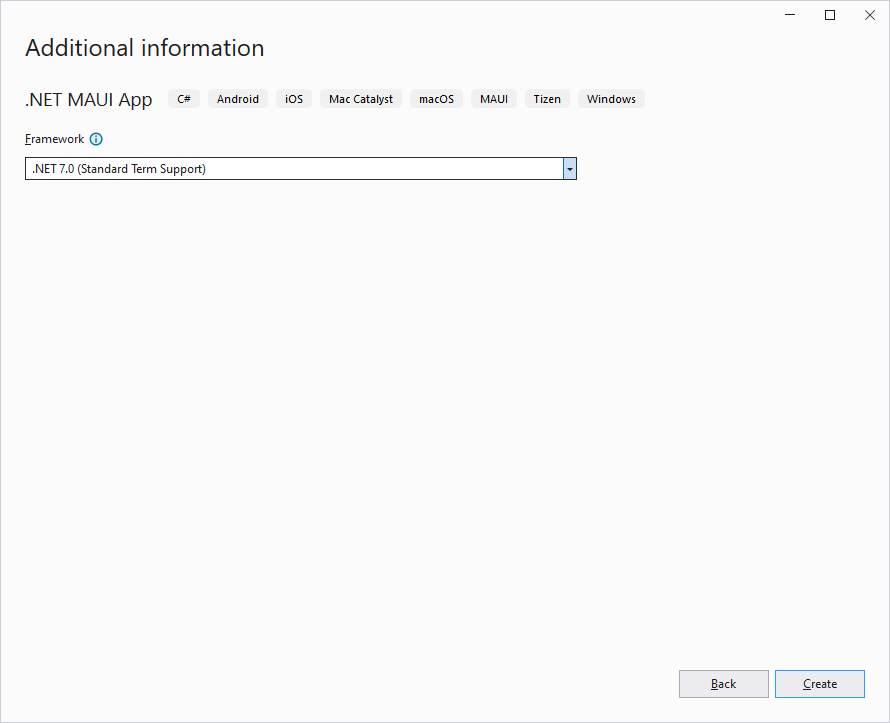
**Figure 7.2**: Selecting .NET MAUI project template

1. Select **.NET MAUI App** and click on **Next**.
2. Enter the Project name as **TipCalculator**.
3. Select the preferred destination folder or leave it with the default setting as shown in *Figure 7.3*:



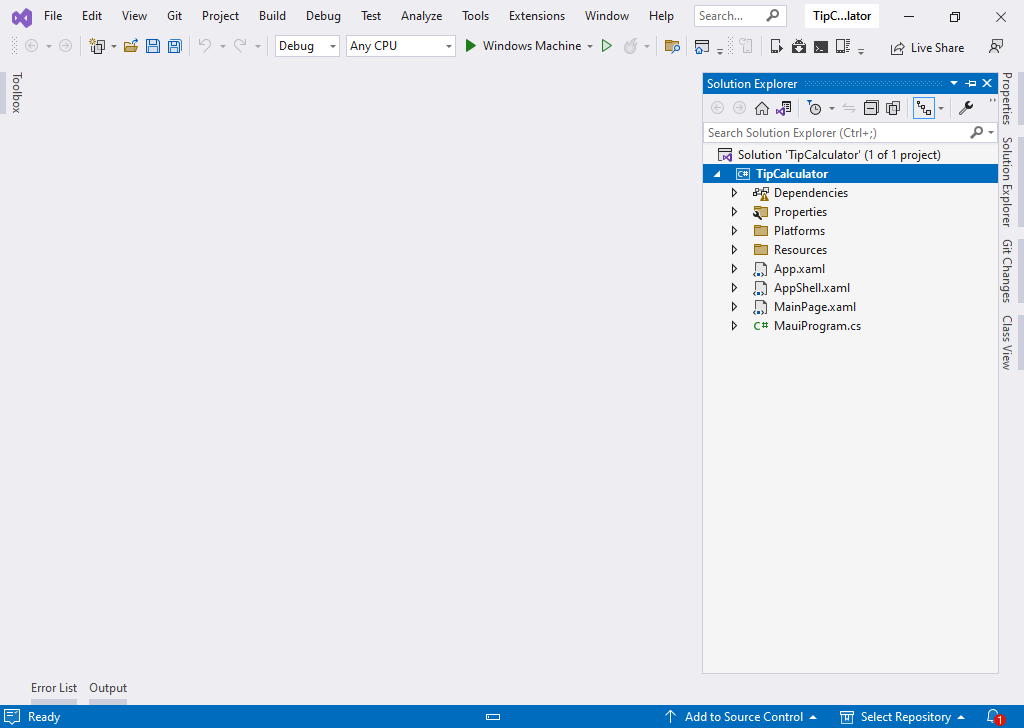
**Figure 7.3**: Configure your project

1. You can choose the latest **Framework** version, which is displayed under **Additional information,** and click on **Create,** as shown in the following figure:



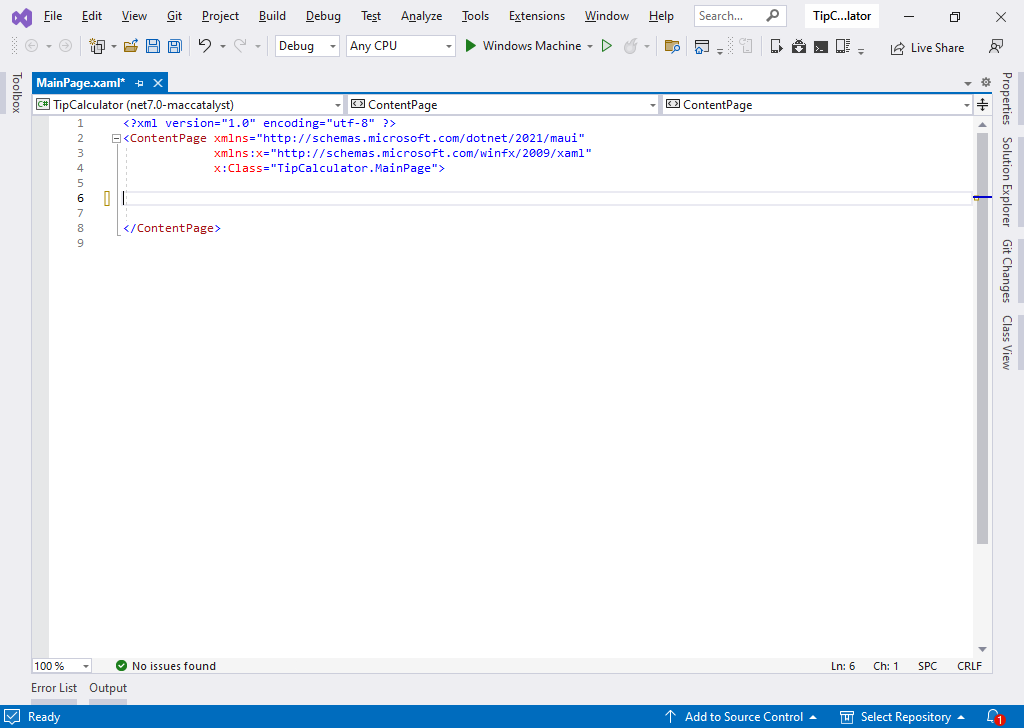
**Figure 7.4**: Selecting the framework version

1. The MAUI project is created, and within a few minutes, you should be able to see the project in the **Solution Explorer**, as shown in the following figure:



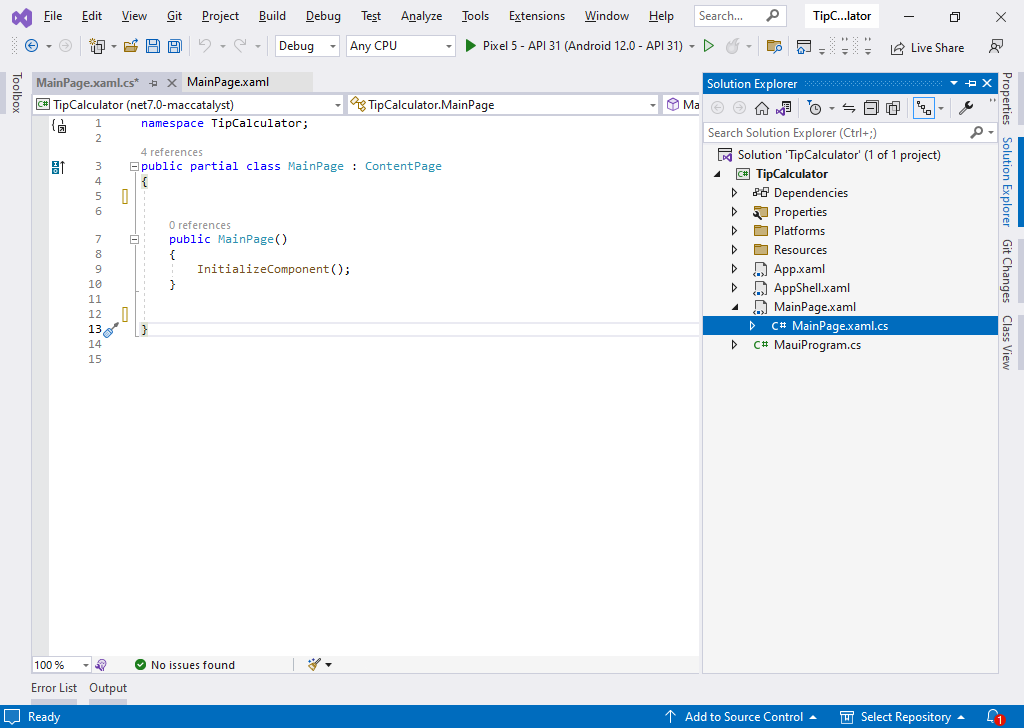
**Figure 7.5**: Default solution in Visual Studio

1. Delete the code between the **ContentPage** tags in the boilerplate code in **MainPage.xaml**. The **ContentPage** will serve as the parent for all the controls in your UI as shown in the following figure:



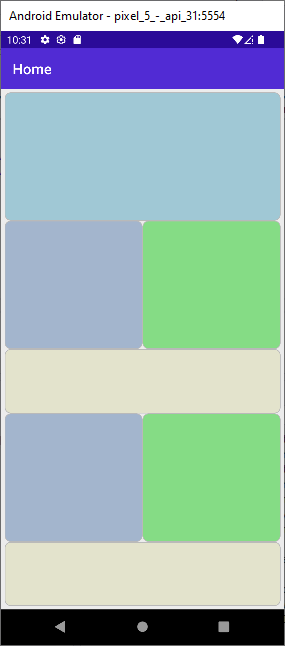
**Figure 7.6**: XAML markup after removing the default project code

1. Open the code-behind file **MainPage.xaml.cs** and remove the default project code, as shown in the following figure:



**Figure 7.7**: Code-behind file after removing the default project code

1. For this app, we are going to have a grid with multiple rows and columns which will serve as the container for the data entry controls that we will add next.
2. By default, a grid will be created with one row and one column. To create multiple rows and columns, we use the **Grid.RowDefinitions** and **Grid.ColumnDefinitions** properties of the **Grid**. Each **RowDefinition** declared inside the **Grid.RowDefinitions** creates a row, and similarly, each **ColumnDefinition** declared inside **ColumnDefinitions** creates a column. For our app, we will need 5 rows and 2 columns.
3. The height of the rows and width of the columns can be defined in three ways:
   1. **Fixed value**: To assign a fixed size of logical units.
   2. **Auto**: To take up only as much space as is required for the controls in that specific row/column.
   3. **Star**: Use star notation for proportional sizing to specify sizes relative to each other based on proportions or percentages. This allows us to allocate available space dynamically among the rows or columns based on their specified proportions, which is better for creating flexible and responsive layouts.
4. We will use the proportional sizing approach to define the heights of the rows and width of the columns. Since the height of each row in the Grid must be different depending on the controls that we will be placing inside the respective rows, we will define the proportions accordingly.
5. The two columns can be of equal width and hence can be declared to have equal proportions. Hence, let us define the Grid’s **RowDefinitions** and **ColumnDefinitions** accordingly. Refer to the following code:
6. <Grid Padding="5">
7. <Grid.RowDefinitions>
8. <RowDefinition Height="0.25\*"></RowDefinition>
9. <RowDefinition Height="0.25\*"></RowDefinition>
10. <RowDefinition Height="0.125\*"></RowDefinition>
11. <RowDefinition Height="0.25\*"></RowDefinition>
12. <RowDefinition Height="0.125\*"></RowDefinition>
13. </Grid.RowDefinitions>
14. <Grid.ColumnDefinitions>
15. <ColumnDefinition Width="0.5\*"></ColumnDefinition>
16. <ColumnDefinition Width="0.5\*"></ColumnDefinition>
17. </Grid.ColumnDefinitions>
18. </Grid>
19. Any control can be placed within a **Grid** by using its **Grid.Row** and **Grid.Column** properties that represent which column and which row the control will be placed in. The values of rows and columns start with 0. That means, if there are two columns in the grid, the first column will be represented by the value **Grid.Column=“0”** and the second column by the value **Grid.Column=“1”**. Row numbers are also determined in a similar way.
20. For our app, we will place **Frame** controls inside the rows and columns. A Frame control is used to wrap a view or layout with a border that can be configured with color, shadow, and other options. Frame controls are generally used to create borders around controls but can also be used to create more complex UI. It may be worth mentioning here that a border can also be used in place of a Frame.
21. For each frame, we will provide a background color using the background property.
22. The frame controls in rows 1, 3 and 5 will have to stretch across the screen spanning two columns. Set the **Grid.ColumnSpan** property of the **Frame** control to 2, so that it stretches across two columns. Refer to the following code:
23. <Grid Padding="5">
24. <Grid.RowDefinitions>
25. <RowDefinition Height="0.25\*"></RowDefinition>
26. <RowDefinition Height="0.25\*"></RowDefinition>
27. <RowDefinition Height="0.125\*"></RowDefinition>
28. <RowDefinition Height="0.25\*"></RowDefinition>
29. <RowDefinition Height="0.125\*"></RowDefinition>
30. </Grid.RowDefinitions>
31. <Grid.ColumnDefinitions>
32. <ColumnDefinition Width="0.5\*"></ColumnDefinition>
33. <ColumnDefinition Width="0.5\*"></ColumnDefinition>
34. </Grid.ColumnDefinitions>
35. <Frame Background="LightBlue"
36. Grid.ColumnSpan="2"></Frame>
37. <Frame Background="LightSteelBlue"
38. Grid.Row="1"></Frame>
39. <Frame Background="LightGreen"
40. Grid.Row="1"
41. Grid.Column="2"></Frame>
42. <Frame Background="Beige"
43. Grid.Row="2"
44. Grid.ColumnSpan="2"></Frame>
45. <Frame Background="LightSteelBlue"
46. Grid.Row="3"></Frame>
47. <Frame Background="LightGreen"
48. Grid.Row="3"
49. Grid.Column="2"></Frame>
50. <Frame Background="Beige"
51. Grid.Row="4"
52. Grid.ColumnSpan="2"></Frame>
53. </Grid>
54. At this stage, the UI should look like the illustration in *Figure 7.8:*



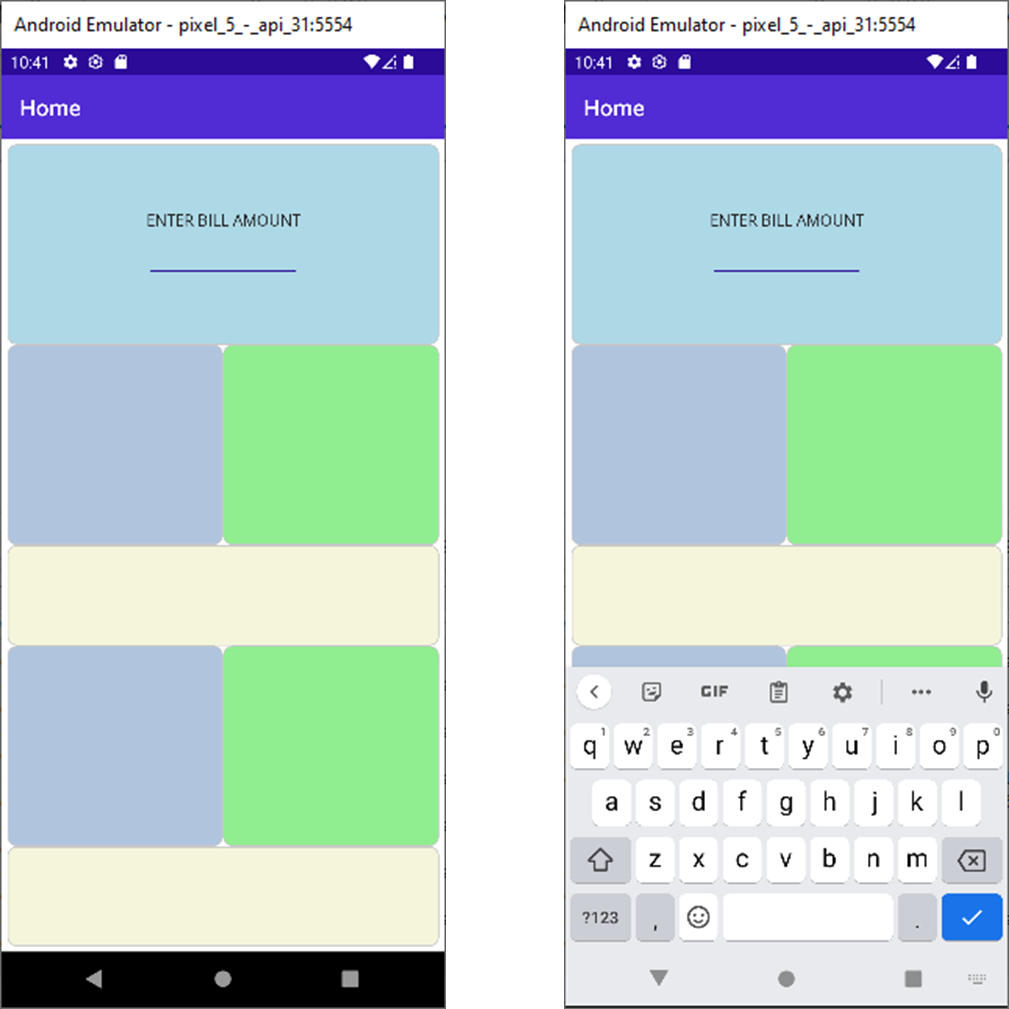
**Figure 7.8**: Layout of the app

Creating the data entry section

Follow these steps to insert the necessary controls to develop the UI that forms the data entry section:

1. In the previous section, we added **Frames** inside the **Grid**. As the **Frame** can take only one child, we will use a layout container like **VerticalStackLayout** or **HorizontalStackLayout** as the child of the **Frame** and then place the other controls of the UI inside it.
2. To position the child controls of the **VerticalStackLayout** in the center, you can set the two properties **HorizontalOptions=Center**and **VerticalOptions =Center**. However, we need to declare this for all the remaining **VerticalStackLayout** controls that we will be adding to the UI later. Therefore, it is advisable to define a **Style** and reuse it.
3. A style can be defined at any level of a page hierarchy, and all the descendant controls can use the style. For our app, we will declare it as a resource at the **Grid** level so that it is available for all the children of the **Grid**.
4. When declaring style, two properties are mandatorily required:
   1. **TargetType** property, which indicates which type of control can use this **Style**.
   2. **X:Key** property provides a name for the style.
5. You can define multiple properties inside a style using **Setter** element. The **Setter** element requires two attributes to be set, **Property** and **Value**.
6. Once you have defined the style, you can apply the style in the target control using the markup extension syntax as shown. Remember that the curly braces are mandatory here. Refer to the following code:
7. <Grid.Resources>
8. <Style TargetType="Frame"
9. x:Key="centerFrame">
10. <Setter Property="HorizontalOptions"
11. Value="Center" />
12. <Setter Property="VerticalOptions"
13. Value="Center" />
14. </Style>
15. </Grid.Resources>
16. <Frame Background="LightBlue"
17. Grid.ColumnSpan="2">
18. <VerticalStackLayout Style="{StaticResource centerFrame}">
20. </VerticalStackLayout>
21. </Frame>
22. For the first row, add a **Label** and an **Entry** control for the bill amount, as shown in the following code:
23. <Grid.Resources>
24. <Style TargetType="Frame"
25. x:Key="centerFrame">
26. <Setter Property="HorizontalOptions"
27. Value="Center" />
28. <Setter Property="VerticalOptions"
29. Value="Center" />
30. </Style>
31. </Grid.Resources>
32. <Frame Background="LightBlue"
33. Grid.ColumnSpan="2">
34. <VerticalStackLayout Style="{StaticResource centerFrame}">
35. <Label Text="ENTER BILL AMOUNT"/>
36. <Entry />
37. </VerticalStackLayout>
38. </Frame>

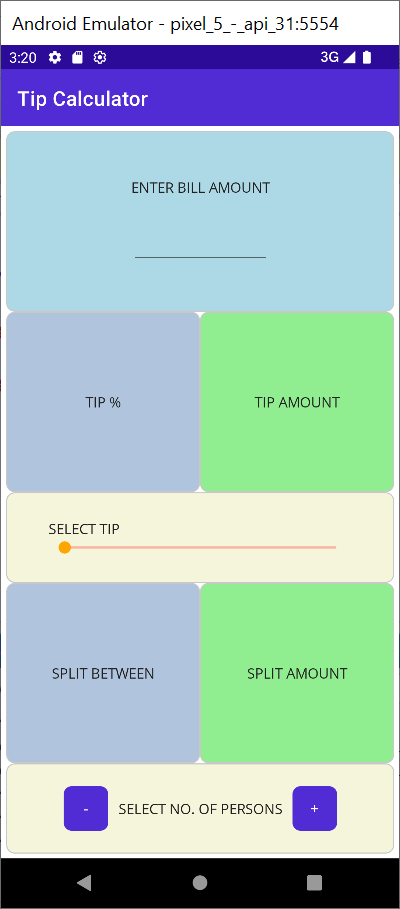
As we have applied the style to the **VerticalStackLayout** to center align the child controls, the **Label** and **Entry** controls will be positioned in the center of the layout as shown in *Figure 7.9*:



**Figure 7.9**: Controls for entering Bill Amount

Note: Notice the Entry displays alphanumeric keyboard by default.

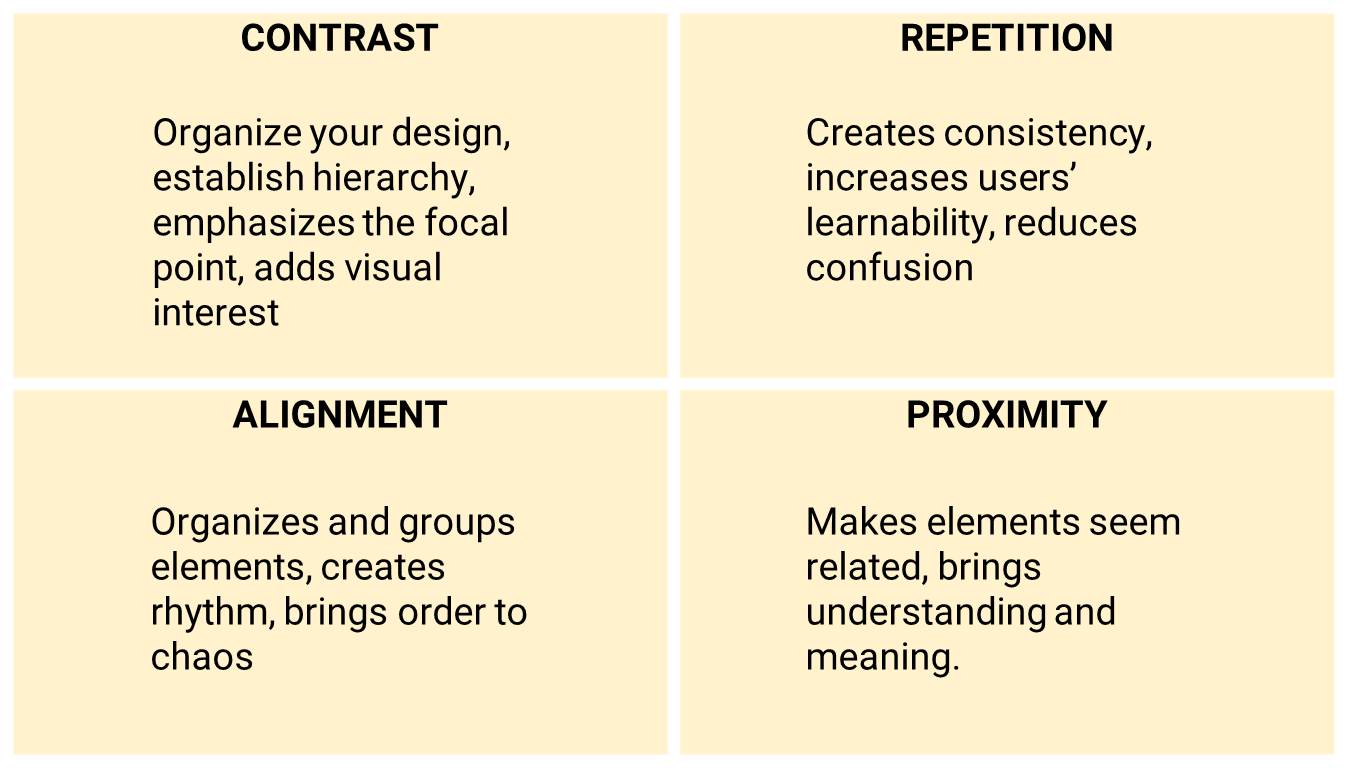
1. Let us add the remaining controls for the UI in the same way. In row 2, for the Tip %, add a label to denote the tip selected by the user using a slider control in row 3. The tip amount will be calculated using C# later and it will appear under the tip amount.
2. In row 3, place a slider to adjust the tip % mentioned in *Step 9*.
3. In row 4, we have two sets of controls. In the first column, we will display the number of people who will split the total bill amount (that is, bill + tip amount). The number of people will be set by button controls in row 5. The second column will display the amount that each person has to pay, that is, the split amount.
4. After completing the above steps, the UI will appear as shown in *Figure 7.10:*



**Figure 7.10**: Initial state of Data Entry section

Applying graphical design principles in UI design

Principles of graphical design give designers a set of guidelines for how to design visually appealing compositions that create wonderful user experiences. By following basic principles of design like **Contrast, Repetition, Alignment, and Proximity (C.R.A.P.),** you can create UIs that people love to use. A gist of the C.R.A.P. principles is given in *Figure 7.11*:



**Figure 7.11**: C.R.A.P. principles of design

Right now, our UI looks basic and crude. We will now apply some graphical design principles to make the UI look even better. Follow these steps to achieve it:

1. Firstly, we will introduce some space between the **Frame** controls. This can be done in two ways:
   1. By declaring **Grid.RowSpacing** and **Grid.ColumnSpacing** properties in the **Grid**.
   2. By applying margins to the **Frame** controls.
   3. We will use the latter approach.
2. The margin property of a control represents the distance between the element and its adjacent elements or neighbors. There are three possible ways to assign a margin to the control:
   1. Margin defined by a single value (for example, **Margin=“10”**): The single value is applied to the left, top, right, and bottom sides of the element.
   2. Margin defined by two values (for example, **Margin=“5,10”**): The first value is applied to the left and right of the element and the second value is applied to the top and bottom of the element.
   3. Margin defined by four distinct values (for example, **Margin=“5,10,15,20”**): The values are applied to the left, top, right, and bottom of the element.
3. Using the above understanding, apply appropriate margins to the **Frame** controls so that they are not touching each other, but at the same time, related controls appear together to indicate some kind of association between these controls.
4. Next, let us customize the button controls. We can use the button properties for **BackgroundColor**, **VerticalOptions**, **HorizontalOptions**, **WidthRequest**, and **FontSize** to customize the **Button**. As the style has to be applied to both the buttons in row 5, we can define this as a style in **Grid**. Resources collection and then using the markup extension syntax we can bind the style property of the button to the customized style that we create.
5. Now, we will customize the labels displaying the numerical values in the UI so that there is an emphasis on the values displayed in the UI. We need to use **FontSize** and **FontAttributes** to our custom style for the **Label** control. Then apply the style to all the Labels which have to display numerical data.
6. After the above changes are done, the XAML markup under **Grid.Resources** would appear as shown in the following code:
7. <Grid.Resources>
8. <Style TargetType="Frame"
9. x:Key="centerFrame">
10. <Setter Property="HorizontalOptions"
11. Value="Center" />
12. <Setter Property="VerticalOptions"
13. Value="Center" />
14. </Style>
15. <Style TargetType="Button"
16. x:Key="splitButton">
17. <Setter Property="BackgroundColor"
18. Value="Orange" />
19. <Setter Property="VerticalOptions"
20. Value="Center" />
21. <Setter Property="HeightRequest"
22. Value="50" />
23. <Setter Property="WidthRequest"
24. Value="50" />
25. <Setter Property="FontSize"
26. Value="Body" />
27. </Style>
28. <Style TargetType="Label"
29. x:Key="displayValue">
30. <Setter Property="FontSize"
31. Value="36" />
32. <Setter Property="FontAttributes"
33. Value="Bold" />
34. </Style>
35. </Grid.Resources>

To apply the styles to the **individuaLcontrols**, apply the styles using the markup extensions as follows:

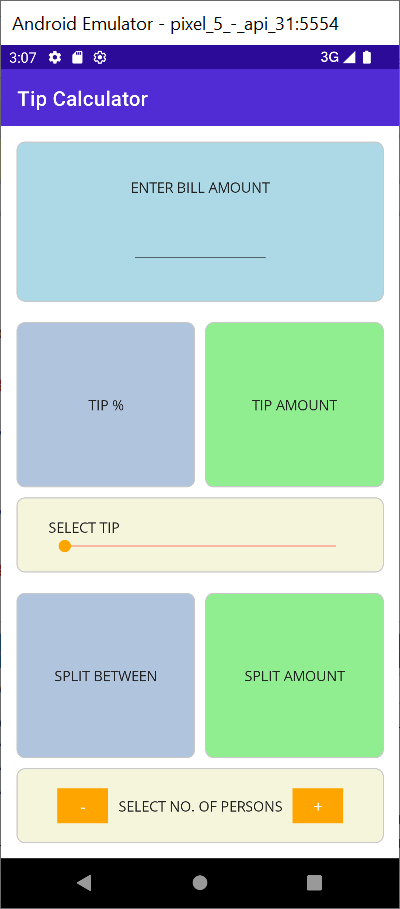
1. <Label x:Name="lblTip" Style="{StaticResource displayValue}" />

Apply the style that we created for **Frame** under window. Resources to different Frames. Refer to the following code:

1. <Frame Background="LightSteelBlue"
2. Margin="10,10,5,5"
3. Grid.Row="1">
4. <VerticalStackLayout Style="{StaticResource centerFrame}">

Similarly, add the style to the **Button**. Refer to the following code:

1. <Button x:Name="btnPlus"
2. Style="{StaticResource splitButton}"
3. Text="+"
4. Clicked="btnPlus\_Clicked"></Button>
5. With the above XAML markup our UI should look better than before, as shown in *Figure 7.12*:

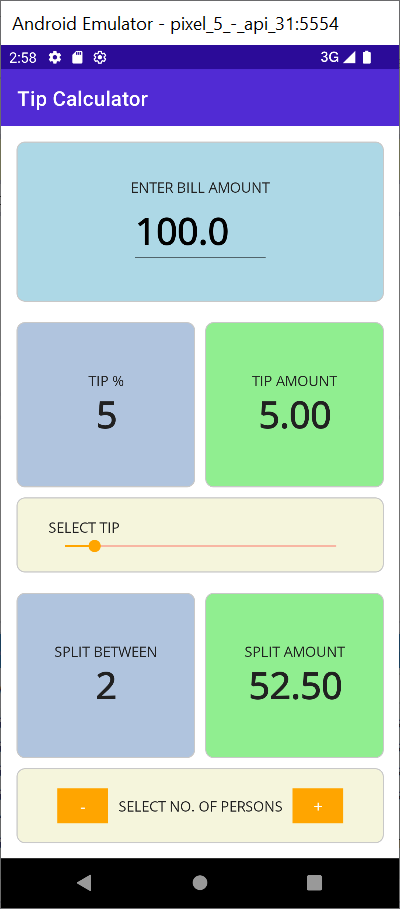


**Figure 7.12**: Improvised UI

Performing the calculation

Now that the UI is complete, let us turn our attention to the functionality. Follow these steps to define the functionality of the app:

1. In the code-behind file, let us define variables to define the bill amount, tip %, tip amount, number of people who will be splitting the bill and the amount to be paid by each person. Now, define the variables **billAmount**, **tipAmount** and **amountPerPerson** variables as shown in the following code:
2. decimal billAmount = 0.00m;
4. int tip = 0;
5. decimal tipAmount = 0.00m;
6. int split = 1;
7. decimal amountPerPerson = 0.00m;
8. Observe the use of the suffix m which is required while assigning values to variables of type decimal.
9. Next, let us declare an **UpdateDisplay()** function which will convert the values of the above variables to String and assign to the respective labels that we have already created in the UI:
10. private void UpdateDisplay()
11. {
12. lblTip.Text = tip.ToString();
13. lblTipAmount.Text = tipAmount.ToString("F2");
14. lblSplit.Text = split.ToString();
15. lblSplitAmount.Text = amountPerPerson.ToString("F2");
16. }
17. We can call this function when the page loads. We will invoke this method in the Page Constructor as shown in the following code:
18. public MainPage()
19. {
20. InitializeComponent();
21. UpdateDisplay();
22. }
23. Whenever the user enters the bill amount in the **Entry** control, it is named as **tbTotal** in this app, we want to calculate the tip amount and the split amount.
24. Let us implement the code in the completed event handler of the entry:
25. private void tbTotal\_Completed(object sender, EventArgs e)
26. {
27. billAmount = decimal.Parse(tbTotal.Text);
28. Calculate();
29. tbTotal.Unfocus();
30. slrTip.Focus();
31. }
32. private void Calculate()
33. {
34. tipAmount = billAmount \* tip / 100;
35. amountPerPerson = (billAmount + tipAmount) / split;
36. UpdateDisplay();
37. }
38. Notice that the code also includes statements to disable focus on the **Entry** and focus on the **Slider** to adjust the tip %. Whenever the slider value is changed, we want the values to be updated, which is achieved with the following code:
39. private void slrTip\_ValueChanged(object sender, ValueChangedEventArgs e)
40. {
41. tip = (int)slrTip.Value;
42. Calculate();
43. }
44. To adjust the number of people who would be sharing the bill, we have two buttons. The event handlers of these buttons can be used to calculate the updated values. Refer to the following code:
45. private void btnMinus\_Clicked(object sender, EventArgs e)
46. {
47. if (split > 1)
48. split--;
49. Calculate();
50. }
51. private void btnPlus\_Clicked(object sender, EventArgs e)
52. {
53. split++;
54. Calculate();
55. }
56. Run the app after making the above changes and test all the functionalities.
57. Now that our app is complete, we have one more change to do. Notice that the header in the UI has been displaying the default text **Home**. Edit the **AppShell.xaml** file and replace the **Title** attribute of the **ShellContent** class with the string **Tip Calculator** or your preferred app name. Refer to the following code:
58. <?xml version="1.0" encoding="UTF-8" ?>
59. <Shell
60. x:Class="TipCalculator.AppShell"
61. xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
62. xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
63. xmlns:local="clr-namespace:TipCalculator"
64. Shell.FlyoutBehavior="Disabled">
65. <ShellContent
66. Title="Tip Calculator"
67. ContentTemplate="{DataTemplate local:MainPage}"
68. Route="MainPage" />
69. </Shell>
70. Run the app after making the above change, and you should be able to see the final app view, as shown in the following figure:



**Figure 7.13**: Final view of the UI

Conclusion

In this chapter, you have developed your second project using .NET MAUI and will be able to run it on your device as well. If you face any errors, check the source code provided for reference, compare it with your code, and make the necessary corrections. In the next chapter, we will improve the skills further by developing a BMI calculator.

Points to remember

Here are some key takeaways from this chapter:

* Ensure that you choose the right .NET Framework version while creating the project.
* This app demonstrated how you can develop a complex UI using different .NET MAUI controls. We have used a Grid inside the ContentPage and then placed frames inside the Grid which in turn hosted the VerticalStackLayout and HorizontalStackLayout controls. However, you may also develop the UI using alternative arrangement of these controls.
* We discussed on following graphical design principles while laying out the controls. While the C.R.A.P. principles are important to improve the usability and aesthetics of the app, in a real-world app, there are many other design features that are equally important which is beyond scope of this book.
* If you have successfully tested the app on the emulator, you can also try deploying it to your mobile device to check the rendering of the app.