# AVALONIA UI

SUCCINCTLY®

BY ALESSANDRO DEL SOLE

SUCCINCTLY® EBOOK SERIES

Syncfusion®

# Avalonia UI Succinctly

**Alessandro Del Sole**

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 111
Morrisville, NC 27560
USA



ISBN: 978-1-64200-244-7



**Technical Reviewer:** James McCaffrey
**Copy Editor:** Courtney Wright
**Acquisitions Coordinator:** Tres Watkins, VP of content, Syncfusion, Inc.
**Proofreader:** Naomi Spicer, content producer, Syncfusion, Inc.

# Table of Contents

# The *Succinctly*® Series of Books

Daniel Jebaraj
CEO of Syncfusion®, Inc.

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

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

Each author is carefully chosen from a pool of talented experts who share our vision. The book before you and the others in this series are the result of our authors' tireless work. Within these pages, you will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

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

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

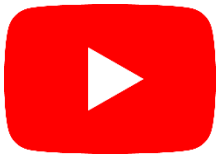
**Let us know what you think**

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctlyseries@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on social media and help us spread the word about the *Succinctly* series!



# About the Author

Alessandro Del Sole is a Xamarin Certified Mobile Developer and former Microsoft MVP. Awarded MVP of the Year in 2009, 2010, 2011, 2012, and 2014, he is internationally considered a Visual Studio expert and a .NET authority. Alessandro has authored many printed books and e-books on programming with Visual Studio, including *Visual Studio Code Distilled, .NET MAUI Succinctly*, *Visual Basic 2015 Unleashed*, and *Xamarin.Forms Succinctly*. He has also written tons of technical articles about .NET, Visual Studio, and other Microsoft technologies in Italian and English for many developer portals.

Alessandro has been a frequent speaker at Italian conferences, and he has also produced a number of instructional videos in both English and Italian. He works as a senior software engineer for Fresenius Medical Care, focusing on building mobile apps with .NET MAUI and Xamarin in the healthcare market. You can follow him on LinkedIn, and support him with a coffee.

# Introduction

Avalonia UI is an open-source framework for cross-platform development. More specifically, it allows for building desktop, mobile, web, and embedded applications using a single .NET codebase. In the modern industry of software development, cross-platform development is something that companies and individual developers cannot avoid anymore, and in most cases, it is also taken for granted. The variety of target systems and devices certainly requires building applications that run on most of them, if a company wants to reach the largest audience possible.

During the last years, many cross-platform development frameworks have been released, and in a sense, .NET is one of them since it allows for building applications on multiple systems. The real point is what we mean by cross-platform. Most cross-platform development frameworks target a number of devices and systems, but they do not target something else. This is the case with .NET MAUI, which targets iOS, Android, Windows, and macOS, but not the browser or Linux systems. This is just one example; other frameworks behave in the same way. And that's where Avalonia UI comes in.

After years of development and with a new stable release, Avalonia UI has the ambition to target the largest variety of systems possible by producing apps that work on mobile operating systems but also on desktop systems like Windows, macOS, and Linux, and that also run in the web browser. This is all possible due to the power of .NET, the technology behind Avalonia UI, which basically runs everywhere.

As you will discover shortly, Avalonia UI provides developers with a natural approach based on XAML and, more specifically, on the Windows Presentation Foundation flavor. At the time of this writing, the latest release of Avalonia UI is version 11. It introduces many framework improvements that will render the user interface even faster.

This book describes the Avalonia UI from a practical point of view, explaining everything you need to know, from setting up the development machine to leveraging the most advanced graphical features. As you can expect, topics are discussed with an approach that is typical for the *Succinctly* series, but at the end, you will be able to build powerful, advanced, and cross-platform projects easily.

It is highly recommended that you download or clone the companion code repository for this book from GitHub. This will make it easier for you to follow all the code examples.

# Chapter 1 Introducing Avalonia UI

This chapter introduces Avalonia UI, explaining what it is, its purposes, and the way you build applications with this framework. It also discusses target platforms and gives you useful suggestions about finding development resources.

## Introducing Avalonia UI

Avalonia UI is an open-source, cross-platform development framework for .NET. With Avalonia UI you can build native applications for Windows, macOS, Linux, iOS, Android, and WebAssembly from a single, shared codebase. From a development point of view, Avalonia UI is based on the *eXtensible Application Markup Language* (XAML) to design the user interface, and on C# and .NET for the imperative code, and the approach is very close to building apps with Windows Presentation Foundation (WPF).

So, if you have existing skills with this technology, building applications with Avalonia UI will be a straightforward experience. In general, any existing experience with other XAML-based development platforms (such as Avalonia UI, .NET MAUI, or Universal Windows Platform (UWP)) will make it easier to work with Avalonia UI, but WPF is certainly the closest in terms of application structure and codebase.

***Note: If you are unfamiliar with XAML, do not worry—you will get an introduction in Chapter 3, "Building the user interface with XAML." Obviously, having at least some basic knowledge is helpful to speed up your productivity and learning path.***

Avalonia UI also supports F# as a programming language, but this will not be covered in this book. If you are curious to understand how Avalonia UI relates to (and differs from) WPF and UWP, the development team has published a comparison page where you can see the list of features and controls that are available in the three platforms, compared to one another.

There is also a more specific page called WPF Developer Tips that explains in more detail the differences between Avalonia UI and WPF. These differences are mostly related to syntax, but in some cases, you will find important notes that will save you some time at coding time, so I recommend reading it.

## Development environment and targets

Avalonia UI works with any .NET version that supports the .NET Standard 2.0 specifications. In addition, official support is offered for Avalonia on Windows 8 and later, and on macOS, High Sierra 10.13 and later. Official support is also offered for the following Linux distributions:

- Debian 9 and later
- Ubuntu 16.04 and later
- Fedora 30 and later

The supported development environments are Microsoft Visual Studio 2017 and later, and JetBrains Rider 2020.3 and later. As you will learn in Chapter 2, you could set up an Avalonia UI project only by using the .NET Command Line Interface (CLI), but in practice, you will also need a visual designer and coding support that can only be offered by an integrated development environment like Visual Studio. At the moment, Visual Studio for Mac is not supported by Avalonia UI as an integrated development environment, so you will need JetBrains Rider if you plan to develop with Avalonia UI on the Mac.

### Cross-platform WPF with Avalonia XPF

The Avalonia UI is currently working on a commercial product called Avalonia XPF, which allows for building WPF applications that run on Windows, macOS, and Linux from a shared codebase. Avalonia XPF specifically targets WPF applications and will ship with a migration analysis tool that will make it easier to update existing WPF projects to be compatible with the Avalonia development framework. It will make them available on the three major desktop systems.

At the moment, Avalonia XPF is only available as an on-demand preview, and it is not generally available. It will be a paid product with different licenses and enterprise-level support. If you are interested in migrating your WPF projects to a cross-platform environment, you might want to keep an eye on the Avalonia XPF website.

## How Avalonia UI works

When you open the Avalonia UI website, the following phrase appears: *Pixel Perfect on Every Platform*. This is exactly the summary of how Avalonia UI works. It generates the user interface by drawing pixels. To do so, it relies on the popular SkiaSharp graphic library. The implication for this is the possibility of supporting any platform that can draw pixels and receive events.

Currently, Avalonia UI supports .NET, Mono, and .NET Framework, but support for other platforms is available via the Avalonia API. In addition, Avalonia UI is designed to work cross-platform, but it has no dependency on other cross-platform technologies like .NET MAUI, for example. It uses the same underlying technologies to achieve native performance, resulting in a fast environment that can also run on low-powered devices like the Raspberry Pi.

## System prerequisites

To follow along with the examples in this book, you'll need to have Microsoft Visual Studio 2022 installed on your machine. The free Community edition is sufficient. In the Visual Studio Installer, make sure you have the .NET Desktop Development workload enabled and that you have at least .NET 6—though .NET 7 is the recommended choice.

If you plan to target mobile devices, make sure the Android SDK Setup component is selected in the list of individual components. My suggestion for the best mobile development experience is selecting the .NET Multi-platform App UI development workload. Even if you will not work with .NET MAUI when using Avalonia UI, this workload will ensure that all the necessary runtime components, emulators, and SDKs for mobile app development will be installed on your machine.

## Source code and examples

As an open-source project, the full source code for the Avalonia UI framework can be found on GitHub. If you want to understand how the platform has been built, you can browse, download, and compile the code on your machine.

> 💡 ***Tip: You will need either .NET Core 3.1 SDK or .NET 6.0 SDK to build the Avalonia UI source code on your machine. In addition, the development team has published a guide with instructions on how to work with the source code effectively.***

There is also a specific repository for sample applications built upon Avalonia UI called Avalonia.Samples where you can find more sophisticated code that leverages advanced programming patterns, such as Model-View-ViewModel (MVVM). It is updated quite frequently, and it is a good reference.

## Chapter summary

Avalonia UI is an open-source framework that allows for creating cross-platform projects for desktop and mobile systems and web browsers. It is based on XAML and C#, and the approach is very close to building apps with WPF. Avalonia UI supports Visual Studio on Windows and JetBrains Rider as integrated development tools, but this book will only cover Visual Studio.

In the next chapter, you will learn how to configure the development environment to start building your applications.

# Chapter 2 Cross-Platform Development with Avalonia UI

Configuring the development environment for Avalonia UI is a fast and easy task. You will need to install the Avalonia UI project templates and a Visual Studio extension that enables the visual designer and the appropriate IntelliSense support. This chapter explains how to configure the development environment, how to create projects within Visual Studio, and how to generate cross-platform solutions for mobile devices. We will not cover setting up the environment for JetBrains Rider.

## Setting up the development environment

Avalonia UI requires either .NET Core 3.1 or .NET 6.0 (or later) on your machine. Since you are working with Visual Studio, you might already have installed this prerequisite. You can check this in the Visual Studio Installer program.

When the current installation is listed, click the **Modify** button and open the **Individual components** tab (see Figure 1).

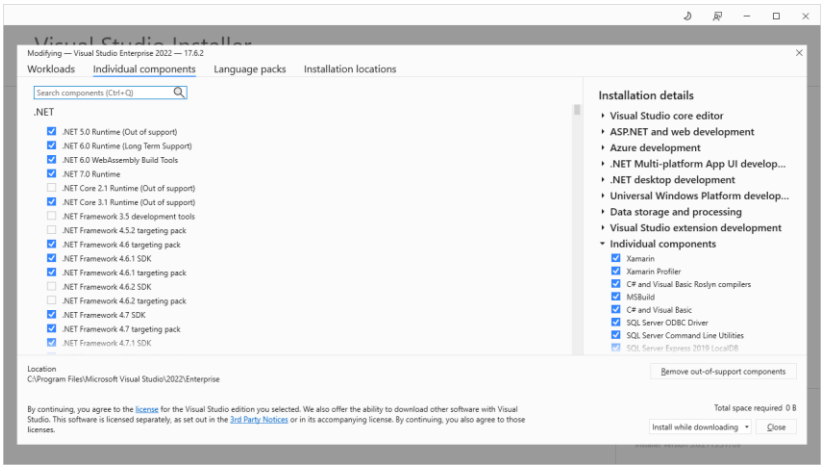



*Figure 1: Checking the .NET availability on the development machine*

The .NET group at the top shows the list of all the installed .NET components. If multiple versions are installed, the .NET CLI will use the most recent. Though Avalonia UI supports both .NET Core 3.1 and .NET 6.0, the recommendation is that you install .NET 7.0, which is the latest version available.

If this is not possible, make sure you install .NET 6.0: select the appropriate runtime in the list, click **Modify**, and wait for the installation to complete. Once the .NET prerequisite is installed, you can proceed with the Avalonia UI setup. This consists of two major blocks: specific project templates and a Visual Studio extension. These are described in the next paragraphs.

## Installing Avalonia UI project templates

In order to install the Avalonia UI project templates, you need the .NET CLI. From the Windows' Start menu, open the **Developer Command Prompt for VS 2022**. When you're ready, type the following command:

```
> dotnet new install Avalonia.Templates
```

This command will install the project templates described in Table 1.

*Table 1: Avalonia UI project templates*

| Project template | Description | Target platforms |
|---|---|---|
| Avalonia .NET Core App | A blank Avalonia UI project | Windows, Linux, macOS |
| Avalonia .NET Core MVVM App | An Avalonia UI project based on the MVVM pattern | Windows, Linux, macOS |
| Avalonia Cross Platform Application | Cross-platform solution for mobile and desktop systems | Android, iOS, web, Windows, Linux, macOS |
| Avalonia Resource Dictionary | Reusable collection of XAML resources | Android, iOS, web, Windows, Linux, macOS |
| Avalonia Styles | Reusable collection of XAML styles | Android, iOS, web, Windows, Linux, macOS |
| Avalonia TemplatedControl | Allows for restyling an existing control | Android, iOS, web, Windows, Linux, macOS |
| Avalonia UserControl | A project template to create a new user control | Android, iOS, web, Windows, Linux, macOS |
| Avalonia Window | Allows for creating an individual window | Windows, Linux, macOS |

The next step for setting up the development environment is installing the extension for Visual Studio.

***Note: Installing the project templates from the command line is a required task because the Visual Studio extension will not install any templates. This also gives you the option to work with Avalonia UI projects outside of Visual Studio. You will***

***also use this command line to update the Avalonia UI templates when a new version is released.***

## Installing the Visual Studio extension

Avalonia UI provides a convenient visual experience via the Visual Studio extension. This is available for Visual Studio 2017, 2019, and 2022, but this book discusses the latter. Having said this, open Visual Studio 2022. Select **Extensions** > **Manage Extensions**, and in the **Manage Extensions** dialog, search for **Avalonia**, as shown in Figure 2.

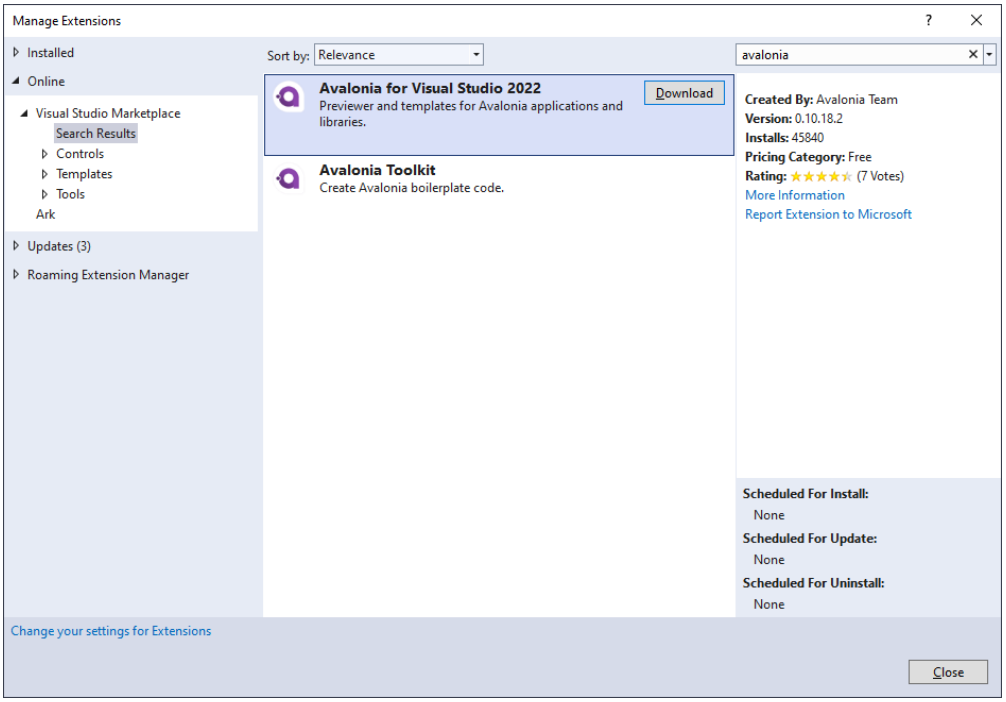



*Figure 2: Locating the Avalonia UI extension for VS 2022*

The official extension is called Avalonia for Visual Studio 2022. Click **Download**, then close Visual Studio so that the extension installer will start. The extension will add a visual designer for Avalonia UI, specific IntelliSense support, and the ability to create new projects from within the IDE.

Once the installation of the extension is complete, reopen Visual Studio 2022. You can choose the **Continue without code** option when the start page appears.

# Creating Avalonia UI projects

In Visual Studio 2022, select **File** > **New** > **Project**. In the **Create a new project** dialog, you can filter the list of project types by selecting **Avalonia**, as highlighted in Figure 3.

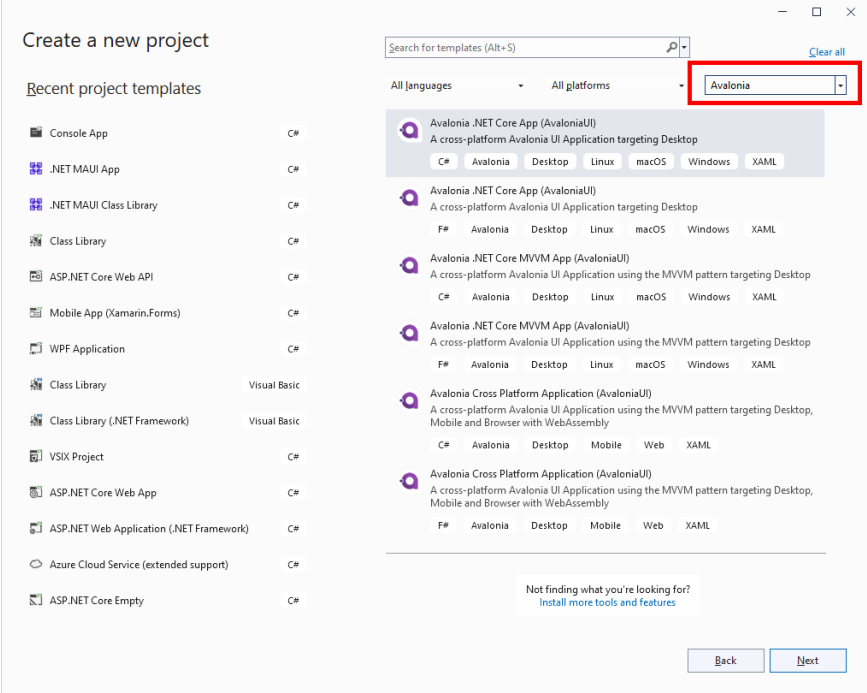



*Figure 3: Displaying the list of Avalonia project templates*

As you can see, the list of project templates targets both C# and F#, and it covers the templates related to desktop and cross-platform solutions.

> ***Tip: Since there is no integrated option to create a resource dictionary and custom control projects, you will need to manually create projects using the .NET CLI for these particular scenarios.***

You will now create two new empty projects with the goal of understanding the project structure of both desktop and cross-platform solutions.

## Creating desktop solutions

Avalonia UI provides the following two templates that target desktop applications:

- Avalonia .NET Core App
- Avalonia .NET Core MVVM App

The first template generates an empty project with one window, and it is the best choice for now, so you can focus on understanding the project structure. The second template provides a more sophisticated solution based on a boilerplate implementation of the MVVM pattern, which is an advanced topic that we will touch upon later in the book.

Now, select the **Avalonia .NET Core App (AvaloniaUI)** template (see Figure 3) and click **Next**. With templates targeting the desktop, Visual Studio will ask you to specify both the framework and the Avalonia version, as shown in Figure 4.

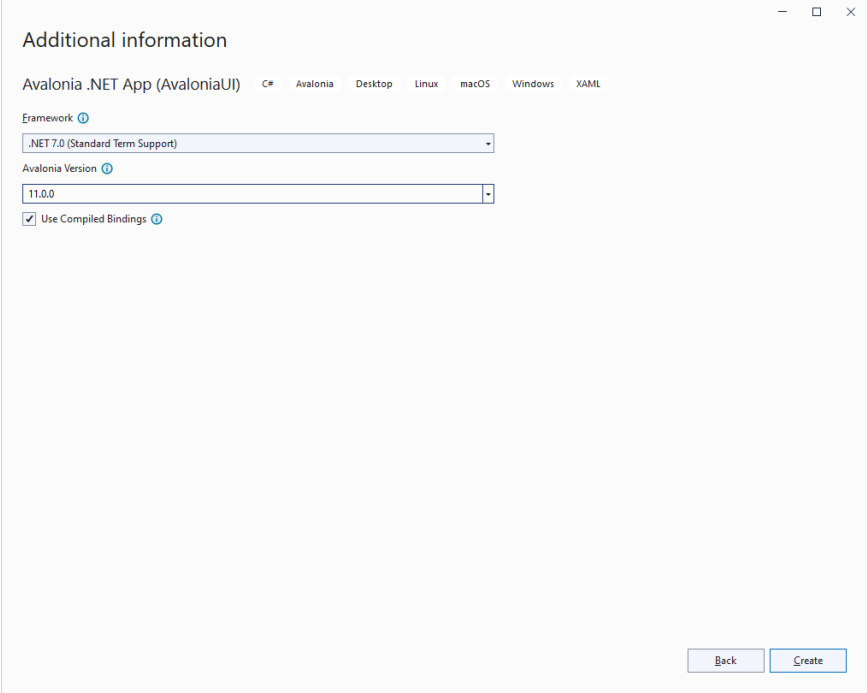



*Figure 4: Selecting platform versions*

In the **Framework** dropdown, make sure the highest version of .NET is selected. In the **Avalonia Version** dropdown, select the latest stable version. At this writing, the latest stable version is **11.0.0**. You can leave the **Use Compiled Bindings** option checked, as this will compile XAML binding expressions at build time rather than at runtime.

When you're ready, click **Create**. After a few seconds, the new project is created and available.

> ***Tip: The first thing you must do when a project is created is build the solution. This will enable the full visual experience. You might also need to rebuild the solution when you manually add a new window.***

## Understanding the project structure

The structure of an Avalonia UI desktop project is very similar to a classic WPF project. Figure 5 shows how the project structure appears in Solution Explorer.

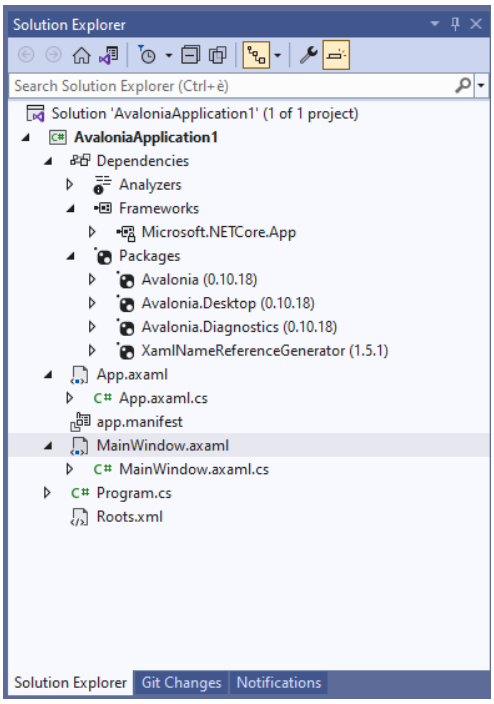



*Figure 5: The structure of an Avalonia desktop project*

The following is a list of the most relevant points, from top to bottom in Figure 5:

- The project is based on the .NET library (Dependencies node, Frameworks sub node).
- Avalonia UI ships via NuGet packages. More specifically, the Avalonia package provides the common runtime; the Avalonia.Desktop package provides support for desktop development; the Avalonia.Diagnostics package provides specific support for debugging and diagnosing problems.
- XAML files for Avalonia UI have the. axaml (Avalonia XAML) extension.
- Similarly to other XAML-based platforms, the App.axaml file contains shared resources available at the application level, and its C# code-behind file contains startup code.
- The MainWindow.axaml file represents a default, blank window together with its C# code-behind file.
- The Program.cs file represents the main entry point for the application.
- The Roots.xml file contains the list of assemblies that are part of the application bundle. By default, this includes the application assembly and the graphical theme.
- The app.manifest file is an XML file that contains a list of Windows versions that are supported by the application. You do not need to change this file for now.

As you can see, the structure of an Avalonia UI desktop project is very similar to a WPF one. In the next section, you will get to know the visual designer.

## Working with the visual designer and the XAML editor

The user interface of an Avalonia UI project is defined via XAML, exactly like with other platforms such as WPF, UWP, Avalonia UI, and .NET MAUI. Every window or page is designed with a specific .axaml file.

By default, an Avalonia UI desktop project contains a file called MainWindow.axaml, which represents the main window of your application. When you open a .axaml file, the Visual Studio editor will look like Figure 6.

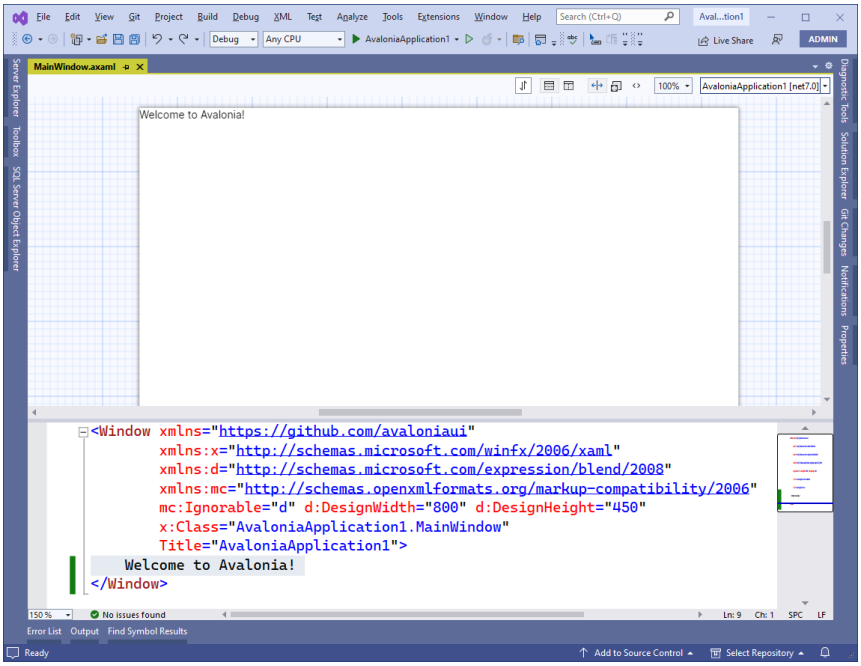



*Figure 6: The XAML editor and visual designer for .axaml files*

***Note: The view might also be split vertically, so Figure 6 might not match what you see on your machine.***

The active editor shows a visual designer and the XAML code editor. Actually, this works more as a previewer rather than a real designer, but the *visual designer* terminology is used for consistency with the official documentation. You can arrange the view using the toolbar at the upper-right corner via the following buttons (from left to right):

- **Swap Preview and XAML panes:** Allows you to swap the position of the designer and the XAML editor.
- **Horizontal Layout:** Arranges the designer and the XAML editor horizontally, like in Figure 6.
- **Vertical Layout:** Arranges the designer and the XAML editor vertically.
- **Split View:** Shows both the designer and the XAML editor in the active window. This is the default setting.
- **Design View:** Shows only the visual designer and hides the XAML editor.
- **XAML View:** Shows only the XAML editor and hides the visual designer.

Unlike other platforms, such as WPF, the Toolbox window shows no controls in Avalonia UI projects, which means that all of the user interface must be written manually in the XAML editor. However, the visual designer provides interaction with some controls once they are added to the design surface. For example, if you add a **`DatePicker`** as follows:

```
<Window xmlns="https://github.com/avaloniaui"
```

```
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
        x:Class="AvaloniaApplication1.MainWindow"
        Title="AvaloniaApplication1">
      <DatePicker />
</Window>
```

The visual designer displays the **DatePicker** and allows you to visually select a date, as you can see in Figure 7 (where the names of the month, day, and year are shown in Italian due to the localization options of my machine).

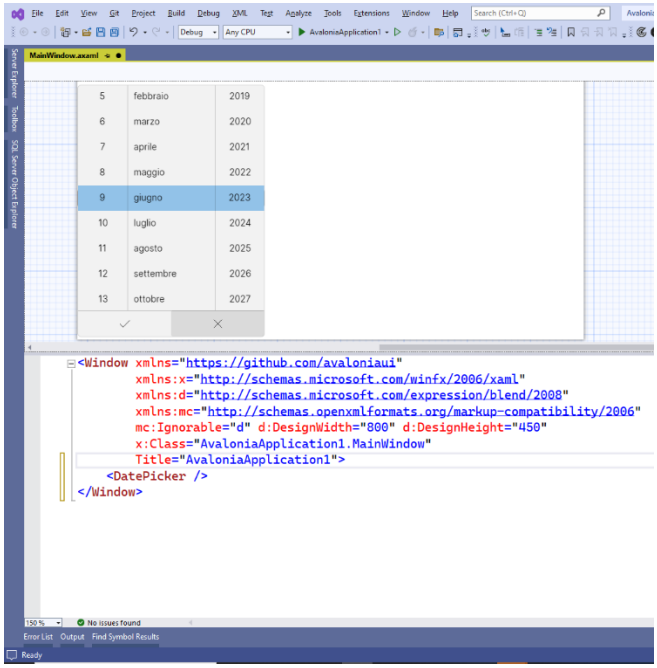



*Figure 7: Working with views in the visual designer*

It is not possible to move controls to a different position using the mouse pointer. This will be done by assigning the appropriate properties to each control. In general, the visual interaction with the designer is quite limited, but you still have the full possibilities in the XAML editor.

## Creating cross-platform solutions

Creating cross-platform solutions is probably the real reason why you are interested in Avalonia UI, so this section describes how to accomplish this.

In Visual Studio, select **File** > **New** > **Project**, and then in the **Create a new project** dialog, select the **Avalonia Cross Platform Application (AvaloniaUI)** template, as shown in Figure 8.

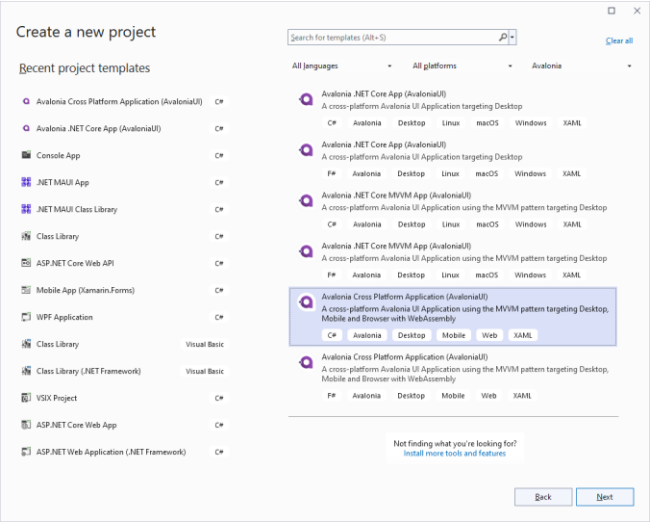



*Figure 8: Creating a new cross-platform solution*

When you click **Next**, you will first be asked to specify a name for the new solution, and then you will be asked to select the target framework (see Figure 4). In the case of cross-platform solutions, you will only see the list of stable .NET versions without previews.

At the end of the project creation, you will see the solution structure in Solution Explorer, which appears as in Figure 9.

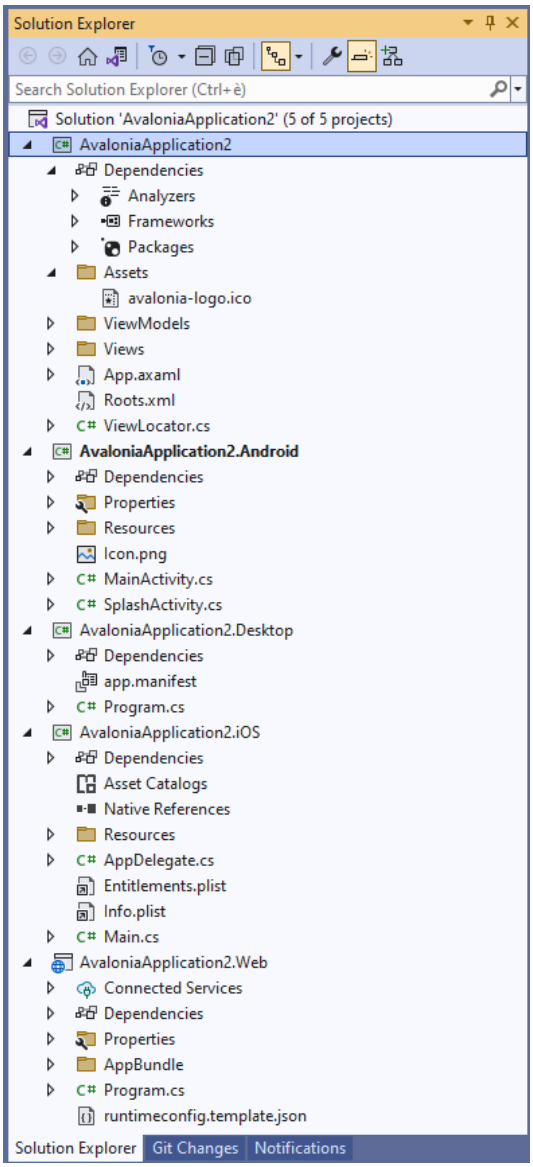



*Figure 9: The structure of an Avalonia UI cross-platform solution*

The solution is made of five projects: one for each target platform (Android, Desktop, iOS, and web), and at the top, a shared project where you will write all the cross-platform code once.

The purpose of an Avalonia UI cross-platform solution is to provide an option to define all the user interface inside the shared project. If you look at Solution Explorer and at the projects in the new solution, you can see that the platform projects do not contain anything but the startup code and specific assets, such as manifests and icons.

The sample solution provides boilerplate code based on the MVVM pattern, which you can replace with your code. Like for the desktop template, the cross-platform project provides a startup view called **`MainWindow`**, defined in the MainWindow.axaml file. When you open a XAML file for editing, the visual designer is shown and requires a solution rebuild the first time. With a cross-platform solution, you can select a different preview, depending on the target platform, by using the combo box at the upper-right corner of the designer, as shown in Figure 10.

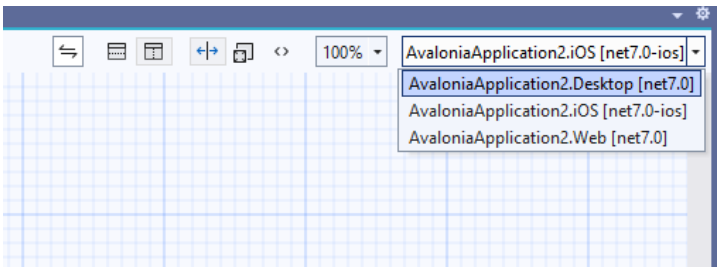



*Figure 10: Previewer selection*

When you do this, you will need to build the solution so that the designer can show the platform-specific view.

## Troubleshooting the visual designer

The Avalonia UI visual designer is generally a stable tool, but there are two known issues that you might encounter:

- When you double-click an .axaml file, the WPF designer is opened instead of the Avalonia UI one. If this happens, simply close the active editor and reopen the file.
- The preview is empty. If this happens, rebuild the solution.

Rebuilding the solution might be required, especially when you select a different project for rendering.

## Working with platform projects

The purpose of this book is to provide general guidance on building applications with Avalonia UI, but it is not possible to explain how individual platforms and OSes work. Though the biggest benefit of a cross-platform environment like Avalonia UI is that you code once using your existing .NET skills without the need to know languages such as Java for Android and Swift for iOS, configuring and publishing platform projects requires some specific knowledge.

For example, for Android you need to customize your application properties and permissions in the ApplicationManifest.xml file, located under the Properties node of the Android project. The official Avalonia documentation contains a group called Platforms, which is organized into pages that explain, among other things, how to configure the development machine for mobile app development, depending on the target OS. You cannot reach such a page directly, so you need to open the documentation root page and locate and expand the **Platforms** node under the How-To Guides group.

Next, you need to know how the Android and iOS applications work from a development point of view. For Android, the best starting point is the Android Developer portal, whereas for iOS, you can start from the Apple Developer portal. Here you will also find guides that explain how to prepare and package your app binaries for distribution.

Avalonia UI allows for running apps in the browser via WebAssembly, so you can look at the official WebAssembly portal for further information. As a general rule, you can test your code on different devices and OSes by selecting a platform project in Solution Explorer as the startup project, and then pressing F5, as you would do with any other type of project supported in Visual Studio.

## Debugging Avalonia UI projects

There is actually no difference in the way you debug an Avalonia UI solution compared to other .NET solutions. You can still use full debugging tools in Visual Studio, such as (but not limited to) breakpoints, data tips, watch and local windows, and so on.

For desktop projects, you start an application for debugging by pressing F5. With cross-platform solutions, you need to first set the startup project and then press F5.

In addition, Visual Studio allows for debugging multiple instances of the same project targeting different devices. For example, look at Figure 11. You can see the previously created project running on the desktop and in an Android device.

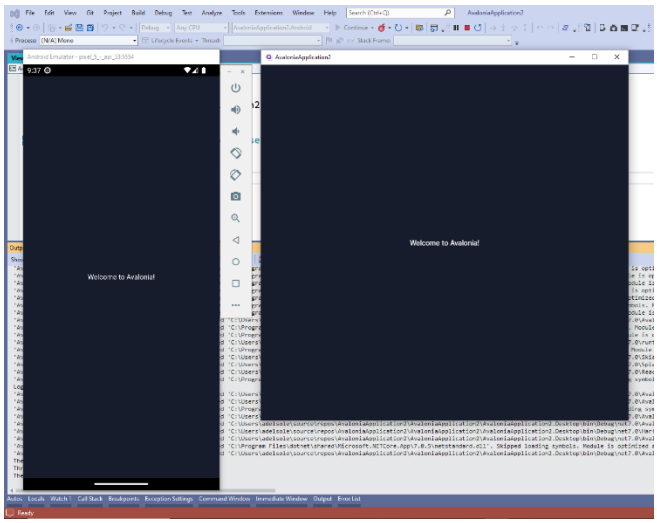



*Figure 11: Debugging cross-platform solutions*

To accomplish this, start debugging the current startup project. Then, in Solution Explorer, right-click the other project you would like to debug concurrently and select **Debug** > **Start New Instance**. This is very useful to compare the behavior of an application in real time. However, remember that debugging multiple instances requires a huge amount of system resources, so it might not be an optimal choice with older machines.

## Understanding the application lifetime

The way the application is being initialized depends on the platform it is running on. If you create a cross-platform solution, you will find the following method inside the App.axaml.cs file:

```
public override void OnFrameworkInitializationCompleted()
{
    if (ApplicationLifetime is
        IClassicDesktopStyleApplicationLifetime desktop)
    {
        desktop.MainWindow = new MainWindow
        {
            DataContext = new MainViewModel()
        };
    }
    else if (ApplicationLifetime is ISingleViewApplicationLifetime
             singleViewPlatform)
    {
        singleViewPlatform.MainView = new MainView
        {
            DataContext = new MainViewModel()
        };
    }

    base.OnFrameworkInitializationCompleted();
}
```

The Avalonia UI runtime first detects the current platform. The **`ApplicationLifetime`** object is of type **`IApplicationLifetime`**, and specialized types are **`IClassicDesktopStyleApplicationLifetime`** or **`ISingleViewApplicationLifetime`**. The first type represents desktop environments, whereas the second type represent platforms that work on a single view, such as mobile apps and the browser. Depending on the platform, the runtime creates an instance of the preferred startup object.

# Chapter summary

Avalonia UI provides a convenient development environment via project templates and a Visual Studio Extension that simplify the approach to this platform. You can quickly create desktop and cross-platform solutions, and you can quickly define the user interface via the XAML editor, the visual designer, and IntelliSense. In the next chapter, you will start learning XAML and how the user interface is organized in Avalonia UI projects.

# Chapter 3 Building the User Interface with XAML

Avalonia UI is, at its core, a library that allows you to create native user interfaces from a single C# codebase by sharing code. This chapter provides the foundations for building the user interface in an Avalonia UI solution. Then, in the next three chapters, you will learn in more detail about layouts, controls, windows, pages, and navigation.

## The structure of the user interface in Avalonia UI

Whether you build applications for the desktop or cross-platform solutions, in Avalonia UI you define the user interface via the *eXtensible Application Markup Language* (XAML). As its name implies, XAML is a markup language that you can use to write the user interface definition in a declarative fashion.

XAML was first introduced more than 15 years ago with Windows Presentation Foundation, and it has always been available in platforms such as Silverlight, Windows Phone, UWP, Xamarin, and .NET MAUI.

XAML derives from XML and offers the following benefits, among others:

- XAML makes it easy to represent structures of elements in a hierarchical way, where pages, layouts, and controls are represented with XML elements and properties with XML attributes.
- It provides clean separation between the user interface definition and the C# logic.
- Being a declarative language separated from the logic, it allows professional designers to work on the user interface without interfering with the imperative code.

The way you define the user interface with XAML is unified across platforms, meaning that you design the user interface once, and it will run on iOS, Android, and Windows.

***Note: XAML in Avalonia UI adheres to Microsoft's XAML 2009 specifications, and the differences with WPF are few. Remember that XAML is case-sensitive for object names and their properties and members.***

For example, when you create an Avalonia UI desktop solution, you can find a file in the project called MainPage.axaml, whose markup is represented in Code Listing 1.

*Code Listing 1*

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
```

```
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
        x:Class="AvaloniaApplication1.MainWindow"
        Title="AvaloniaApplication1">
    Welcome to Avalonia!
</Window>
```

XAML files in Avalonia UI normally contain the definition for a window or a custom view. The root element is a **`Window`** object, which represents its C# class counterpart and is rendered as an individual window for desktop apps and as a page in mobile apps.

The content of a window (or panel) goes inside the **`Window.Content`** property. However, in XAML the **`Content`** property is implicit, meaning you do not need to write a **`Window.Content`** element. The compiler assumes that the visual elements you enclose between the **`Window`** tags are assigned to the **`Window.Content`** property. In the case of an empty project, the content for the window is the **`Welcome to Avalonia!`** text.

XAML allows for better organization and visual representation of the structure of the user interface. If you look at the root element, you can see a number of attributes whose definition starts with **`xmlns`**. These are referred to as XML namespaces and are important because they make it possible to declare visual elements defined inside specific namespaces or XML schemas.

For example, **`xmlns`** points to the root XAML namespace defined inside a specific XML schema and allows for adding to the UI definition all the visual elements defined by Avalonia UI; **`xmlns:x`** points to an XML schema that exposes built-in types; and **`xmlns:d`** points to a namespace that simplifies the design-time definition of the visual elements.

As you will learn in Chapter 4, visual elements are arranged inside *panels* (also referred to as *layouts*). Each window or panel can only contain one visual element. In the case of the autogenerated MainPage.axaml page, you cannot add other visual elements to the page unless you organize them into a panel. Let's provide an example with code that declares some text and a button. This requires you to have a **`Button`** below a **`TextBlock`**, and both must be wrapped inside a container such as the **`StackPanel`**, as demonstrated in Code Listing 2.

***Tip: IntelliSense will help you add visual elements faster by showing element names and properties as you type. You can then simply press Tab or double-click to quickly insert an element.***

*Code Listing 2*

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
```

```
        x:Class="AvaloniaApplication1.MainWindow"
        Title="AvaloniaApplication1">
        <StackPanel Orientation="Vertical" Margin="10">
               <TextBlock Text="Welcome to Avalonia UI!" />

               <Button Name="Button1" Content="Click here!"
            Margin="0,10,0,0"/>
        </StackPanel>
</Window>
```

If you did not include both controls inside the panel, Visual Studio will raise an error. You can nest other layouts inside a parent layout and create complex hierarchies of visual elements. Notice the **`Name`** assignment for the **`Button`**. Generally speaking, with **`Name`**, you can assign an identifier to any visual element so that you can interact with it in C# code, for example, if you need to retrieve a property value.

If you have never seen XAML before, you might wonder how you can interact with visual elements in C# at this point. In the Solution Explorer, if you expand the **MainWindow.axaml** file, you will see a nested file called **MainWindow.axaml.cs**. This is the so-called code-behind file, and it contains all the imperative code for the current page. In this case, the simplest form of a code-behind file, the code contains the definition of the **`MainWindow`** class, which inherits from **`Window`**, and the page constructor, which makes an invocation to the **`InitializeComponent`** method of the base class and initializes the window. You will access the code-behind file often from Solution Explorer, especially when you need to respond to events raised by the user interface.

## Limitations of the XAML editor and productivity tools

At the moment, the XAML editor for Avalonia UI and the related productivity tools do not offer the same features and power that you can expect in other platforms like WPF or .NET MAUI. For example, tools like quick actions, Go to Definition, and Peek Definition are not available in Avalonia UI. Also, when you type an event name, IntelliSense will not give you the possibility to quickly generate a new handler by pressing Tab, as you would do in any other XAML project type.

With regard to events, you will first need to manually generate an event handler in the code-behind file, and then you will be able to assign the handler to the event declaration in XAML; otherwise, the Avalonia editor will not be able to recognize the event handler. It is reasonable to expect improvements in the near future, but this is how it works today.

## Responding to events

Events are fundamental for the interaction between the user and the application, and controls in Avalonia UI raise events, as normally happens in any platform. Events are handled in the C# code-behind file. For instance, suppose you want to perform an action when the user taps the button defined in the previous code. The **`Button`** control exposes an event called **`Click`** that you assign the name of an event handler, but you first need to declare an event handler as follows:

```
private void Button1_Click(object sender, RoutedEventArgs eventArgs)
{
    // Do actions here...
}
```

Then, assign the handler to the event in XAML, like in the following example:

```
<Button Name="Button1" Text="Click here!" Margin="0,10,0,0"
 Click="Button1_Click"/>
```

***Tip: As mentioned in the previous paragraph, the way you declare and assign events in Avalonia UI has some limitations if compared to other platforms.***

About visual elements, their event handlers' signatures require two parameters: one of type **`object`** representing the control that raised the event, and one object of type **`RoutedEventArgs`** containing information about the event. In many cases, event handlers work with derived versions of **`RoutedEventArgs`**, but these will be highlighted when appropriate.

### Understanding routed events

Avalonia UI implements *routed events*, exactly like WPF. To understand routed events, you need to understand that Avalonia UI handles events differently due to its hierarchical structure. When an event is triggered, it passes through the entire visual tree, and each element in the visual tree re-raises the event until it reaches its destination. The Avalonia UI runtime will figure out which element triggered the event, and which element is the recipient of the event itself. For a better understanding, consider the following code snippet:

```
<Button Text="Click here!" Margin="0,10,0,0" Click="Button_Click"/>
```

The **`Button`** has no name, but it has an event handler assigned, which looks like the following:

```
private void Button_Click(object sender, RoutedEventArgs eventArgs)
{
    // Do actions here...
}
```

Though the button has no name, the routed event handler is still able to understand if a click event has been raised from that button. And even if you had the following situation:

```
<Button Text="Click here!" Margin="0,10,0,0" Click="Button_Click"/>
```

```
<Button Text="Do not click here!" Margin="0,10,0,0" Click="Button_Click"/>
```

The runtime could understand which control raised the event without specifying a name. One of the benefits of this approach is that you can have one common event handler for multiple views. You would need code like the following to cast the sender to a typed object and determine which control has actually raised the event:

```
private void Button_Click(object sender, RoutedEventArgs eventArgs)
{
    Button senderObject = sender as Button;
    if(senderObject.Content == "Click here!")
    {

    }
    else
    {

    }
}
```

## Understanding the routing strategies

Routed events can go through three different directions across the tree of visual elements. Such directions are referred to as routing strategies and are represented by values from the **`RoutingStrategy`** enumeration. Not only are they important for your general knowledge of the platform, but also because you can provide a strategy when creating custom routed events. Routing strategies are defined as follows:

- **`Tunnel`**: Events are raised from the visual element and pass through the entire visual tree until they arrive at the event recipient. This is the most common strategy.
- **`Bubble`**: Events start from the event recipient and go back through the entire visual tree until they arrive at the object that raised the action.
- **`Direct`**: Events are directly raised against the recipient. You can compare this strategy to the way events are raised in Windows Forms, and in general, platforms that are not based on XAML.

The previous code snippet, related to the **`Click`** event handler for a button, is based on the **`Tunnel`** strategy: the user clicks on the button and the event passes through the visual tree (the **`Window`**, the container, and finally the button).

Tunneling is certainly the most common strategy that you will see across the book, but don't forget to bookmark the documentation page about routed events—it is a very important topic.

### Understanding type converters

If you look at Code Listing 2, you will see that the **`Orientation`** property of the **`StackPanel`** is assigned the **`Vertical`** value, and it is of type **`Orientation`**, whereas the **`Margin`** property assigned to the **`Button`** is of type **`Thickness`**. However, these properties are assigned values passed in the form of strings in XAML.

Avalonia UI (and all the other XAML-based platforms) implements the so-called *type converters*, which automatically convert a string into the appropriate value for a number of known types. Another common example of a type converter is about assigning an individual **`Color`** to control properties such as **`Foreground`** or **`Background`**, which are used instead of type **`IBrush`**. In this case, the built-in type converter converts the color into a **`SolidColorBrush`** (brushes will be discussed in Chapter 9, “Brushes, Graphics, and Animations”).

Summarizing all the available type converters and known target types is neither possible nor necessary at this point; you simply need to remember that, in most cases, strings you assign as property values are automatically converted into the appropriate type on your behalf.

## Coding the user interface in C#

In Avalonia UI, you can also create the user interface of an application in C# code. For instance, Code Listing 3 demonstrates how to create a page with a layout that arranges controls in a stack containing a label and a button. For now, do not focus on element names and their properties (they will be explained in the next chapter). Rather, focus on the hierarchy of visual elements that the code introduces.

*Code Listing 3*

```
var newWindow = new Window();
newWindow.Title = "New window";

var newLayout = new StackPanel();
newLayout.Orientation = Avalonia.Layout.Orientation.Vertical;

var newTextBlock = new TextBlock();
newTextBlock.Text = "Welcome to Avalonia UI!";

var newButton = new Button();
newButton.Content = "Click here!";
newButton.Margin = new Thickness(0, 10, 0, 0);

newLayout.Children.Add(newTextBlock);
newLayout.Children.Add(newButton);

newWindow.Content = newLayout;
```

Here you have full IntelliSense support. However, as you can imagine, creating a complex user interface entirely in C# can be challenging for at least the following reasons:

- Representing a visual hierarchy made of tons of elements in C# code is extremely difficult.
- You must write the code in a way that allows you to distinguish between user interface definition and other imperative code.
- As a consequence, your C# becomes much more complex and difficult to maintain.

This should clarify why using XAML to declare the user interface is the most convenient way and that coding visual elements in C# should only be done when you need to generate new visual elements at runtime.

## Chapter summary

This chapter provided a high-level overview of how you define the user interface with XAML, based on a hierarchy of visual elements. You have seen how to add visual elements and how to assign their properties; you have seen how type converters allow for passing string values in XAML and how the compiler converts them into the appropriate types. You also learned the important concept of routed events and how they can be used to create common event handlers.

After this overview of how the user interface is defined in Avalonia UI, it is time to discuss important concepts in more detail, and we will start by organizing the user interface with panels.

# Chapter 4 Organizing the UI with Panels

One of the goals of XAML is simplifying the generation of dynamic user interfaces that automatically rearrange visual elements depending on page size and screen factors. Cross-platform development is even more important because phones, tablets, and laptops have different screen sizes and form factors. They also support both landscape and portrait orientations. Therefore, the user interface must dynamically adapt to the system, screen, and device so that visual elements can be automatically resized or rearranged based on the form factor and device orientation. In Avalonia UI, this is accomplished with panels, which is the topic of this chapter.

## Understanding the concept of panels

***Tip: If you have previous experience with WPF, UWP, or .NET MAUI, the concept of panels is something you already know.***

One of the goals of Avalonia UI is providing the ability to create dynamic interfaces that can be rearranged according to the user's preferences, or to the device and screen size. Because of this, controls in applications you build with Avalonia UI should not have a fixed size or position on the UI, except in a very limited number of scenarios. To make this possible, Avalonia UI controls are arranged within special containers, known as *panels*. Panels are classes that allow for arranging visual elements in the UI, and Avalonia UI provides many of them.

In this chapter, you'll learn about available panels and how to use them to arrange controls. The most important thing to keep in mind is that controls in Avalonia UI have a hierarchical logic; therefore, you can nest multiple panels to create complex user experiences. Table 2 summarizes the available panels. You'll learn about them in more detail in the sections that follow.

*Table 2: Layouts in Avalonia UI*

| Layout | Description |
|---|---|
| `StackPanel` | Allows you to place visual elements near each other horizontally or vertically on a single line. |
| `WrapPanel` | Allows you to place visual elements near each other horizontally or vertically. Wraps visual elements to the next row or column if not enough space is available. |
| `Grid` | Allows you to organize visual elements within rows and columns. |

| Layout | Description |
|---|---|
| **`Canvas`** | A panel placed at a specified, fixed position. |
| **`RelativePanel`** | A panel whose position depends on relative constraints. |
| **`ScrollViewer`** | Allows you to scroll the visual elements it contains. |
| **`DockPanel`** | Allows you to arrange visual elements relative to each other, either horizontally or vertically |
| **`Panel`** | Lays out all the children to fill the bounds of the panel itself. It works like a **`Grid`** with no rows and columns. |

Remember that only one root panel is assigned to the **`Content`** property of a **`Window`**, and that panels can then contain nested visual elements and panels.

***Tip: The `Panel` container will not be discussed, since its usage is less common and can be covered by talking about the `Grid`.***

## Alignment and spacing options

Panels can be aligned by assigning the **`HorizontalAlignment`** and **`VerticalAlignment`** properties with one of the property values from the **`HorizontalAlignment`** and **`VerticalAlignment`** enumerations, respectively. These are summarized in Table 3. Though not mandatory, providing an alignment option is very common.

*Table 3: Alignment options in Avalonia UI*

| Alignment | Enumeration | Description |
|---|---|---|
| **`Center`** | **`VerticalAlignment`** | Aligns the visual element at the center, vertically |
| **`Top`** | **`VerticalAlignment`** | Aligns the visual element at the top |
| **`Bottom`** | **`VerticalAlignment`** | Aligns the visual element at the bottom |

| Alignment | Enumeration | Description |
|---|---|---|
| **`Stretch`** | **`VerticalAlignment`** | Expands the visual element's bounds to fill the available space vertically |
| **`Center`** | **`HorizontalAlignment`** | Aligns the visual element at the center, horizontally |
| **`Right`** | **`HorizontalAlignment`** | Aligns the visual element at the right |
| **`Left`** | **`HorizontalAlignment`** | Aligns the visual element at the left |
| **`Stretch`** | **`HorizontalAlignment`** | Expands the visual element's bounds to fill the available space horizontally |

You can also control the space between visual elements with two properties: **`Spacing`** and **`Margin`**, summarized in Table 4.

*Table 4: Spacing options in Avalonia UI*

| Spacing | Description |
|---|---|
| **`Margin`** | Represents the distance between the current visual element and its adjacent elements with either a fixed value for all four sides, or with comma-separated values for the left, top, right, and bottom. It is of type **`Thickness`** and XAML has a built-in type converter for it. |
| **`Spacing`** | Available only in the **`StackPanel`** container, it allows you to set the amount of space between each child element. |

I recommend you spend some time experimenting with how alignment and spacing options work in order to understand how to get the appropriate result in your user interfaces.

# The StackPanel

The **`StackPanel`** container allows the placing of controls near each other, as in a stack that can be arranged both horizontally and vertically. As with other containers, the **`StackPanel`** can contain nested panels. The following code shows how you can arrange controls horizontally and vertically. Code Listing 4 shows an example with a root **`StackPanel`** and two nested layouts.

*Code Listing 4*

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
        x:Class="Ch4_Panels.StackPanelWindow"
        Title="StackPanelWindow">
       <StackPanel Orientation="Vertical">
              <StackPanel Orientation="Horizontal" Margin="5">
                     <TextBlock Text="Sample controls" Margin="5"/>
                     <Button Content="Test button" Margin="5"/>
              </StackPanel>
              <StackPanel Orientation="Vertical" Margin="5">
                     <TextBlock Text="Sample controls" Margin="5"/>
                     <Button Content="Test button" Margin="5"/>
              </StackPanel>
       </StackPanel>
</Window>
```

The result of the XAML in Code Listing 4 is shown in Figure 12.

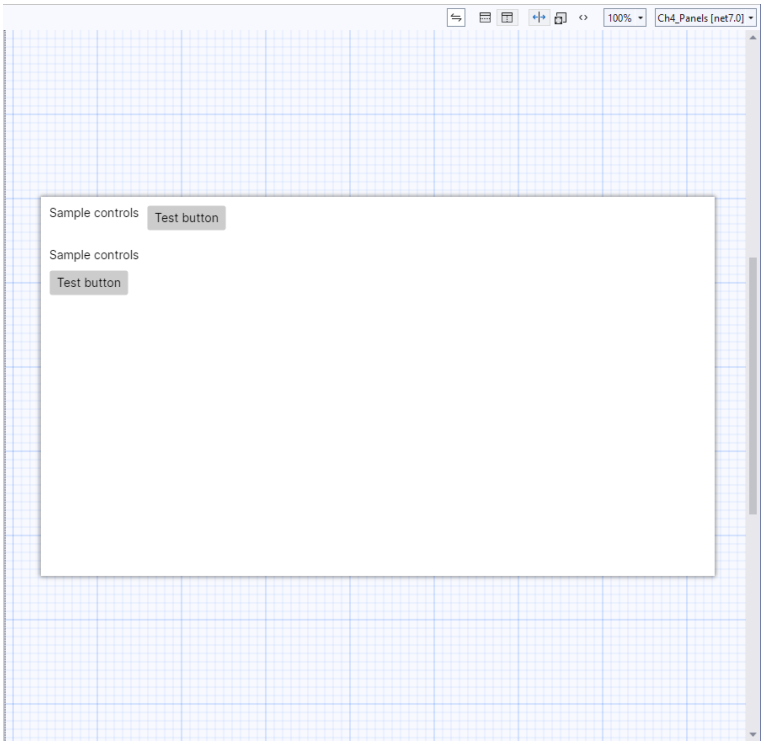



*Figure 12: Arranging visual elements with the StackPanel*

The **Orientation** property can be set as **Horizontal** or **Vertical**, and this influences the final layout. If not specified, **Vertical** is the default. One of the main benefits of XAML code is that element names and properties are self-explanatory, and this is the case in **StackPanel**'s properties, too.

Remember that controls within a **StackPanel** are automatically resized according to the orientation. If you do not like this behavior, you need to specify **Width** and **Height** properties on each control, which represent the width and height, respectively.

**Spacing** is a property that you can use to adjust the amount of space between child elements; this is preferred to adjusting the space on the individual controls with the **Margin** property.

## The WrapPanel

The **WrapPanel** works like a **StackPanel**, since it arranges child visual elements vertically or horizontally, but the difference is that it is also able to wrap the child visual elements if there is not enough space in a single row or column. Code Listing 5 provides an example and shows how easy it is to work with this layout.

*Code Listing 5*

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
        x:Class="Ch4_Panels.WrapPanelWindow"
        Title="WrapPanelWindow">
       <WrapPanel>
              <WrapPanel Orientation="Horizontal" Margin="5">
                     <TextBlock Text="Sample controls" Margin="5"/>
                     <Button Content="Test button" Margin="5"/>
              </WrapPanel>
              <WrapPanel Orientation="Vertical" Margin="5">
                     <TextBlock Text="Sample controls" Margin="5"/>
                     <Button Content="Test button" Margin="5"/>
              </WrapPanel>
       </WrapPanel>
</Window>
```

Figure 13 shows a sample result for Code Listing 5 obtained by assigning the **d:DesignWidth** property of the **Window** with **200**. By reducing the window's width, the **WrapPanel** brings visual elements to the next row when necessary, making them still visible.

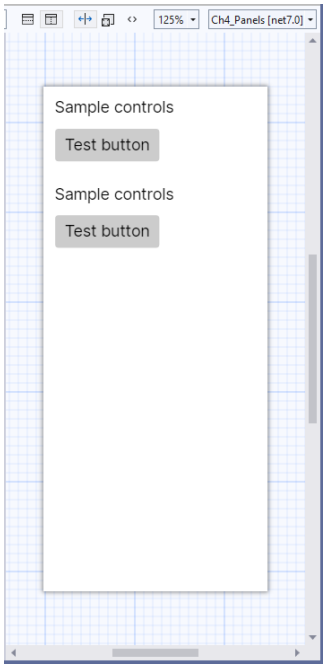



*Figure 13: Arranging visual elements with the FlexLayout*

Remember that, like for the **`StackPanel`**, visual elements in a **`WrapPanel`** are aligned from left to right.

# The Grid

The **`Grid`** is one of the easiest layouts to understand, and probably the most versatile. It allows you to create tables with rows and columns. In this way, you can define cells, and each cell can contain a control or another layout storing nested controls. The **`Grid`** is versatile in that you can just divide it into rows or columns, or both.

The following code defines a **`Grid`** that is divided into two rows and two columns:

```
<Grid>
   <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
   </Grid.RowDefinitions>
   <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
   </Grid.ColumnDefinitions>
</Grid>
```

**RowDefinitions** is a collection of **RowDefinition** objects, and the same is true for **ColumnDefinitions** and **ColumnDefinition**. Each item represents a row or a column within the **Grid**, respectively. You can also specify a **Width** or a **Height** property to delimit row and column dimensions; if you do not specify anything, both rows and columns are dimensioned at the maximum size available. When resizing the parent container, rows and columns are automatically rearranged.

The preceding code creates a table with four cells. To place controls in the **Grid**, specify the row and column position via the **Grid.Row** and **Grid.Column** properties, known as attached properties, on the control.

***Note: The Avalonia UI documentation explains how to create and consume new attached properties, but if you want to discover more about this topic, I recommend that you have a look at the WPF documentation.***

Attached properties allow for assigning properties of the parent container from the current visual element. The index of both is zero-based, meaning that **0** represents the first column from the left and the first row from the top. You can place nested layouts within a cell or a single row or column. The code in Code Listing 6 shows how to nest a grid into a root grid with child controls.

***Tip:*** *Grid.Row="0"* ***and*** *Grid.Column="0"* ***can be omitted.***

*Code Listing 6*

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
        x:Class="Ch4_Panels.GridPanelWindow"
        Title="GridPanelWindow">
       <Grid>
              <Grid.RowDefinitions>
                     <RowDefinition />
                     <RowDefinition />
              </Grid.RowDefinitions>
              <Grid.ColumnDefinitions>
                     <ColumnDefinition />
                     <ColumnDefinition />
              </Grid.ColumnDefinitions>
              <Button Content="First Button" />
              <Button Grid.Column="1"
                            Content="Second Button"/>

              <Grid Grid.Row="1">
                     <Grid.RowDefinitions>
                            <RowDefinition />
```

```
                        <RowDefinition />
                    </Grid.RowDefinitions>
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition />
                        <ColumnDefinition />
                    </Grid.ColumnDefinitions>
                    <Button Content="Button 3" />
                    <Button Content="Button 4" Grid.Column="1" />
                </Grid>
        </Grid>
</Window>
```

Figure 14 shows the result of this code.


*Figure 14: Arranging visual elements with the Grid*

The **`Grid`** layout is very versatile and is also a good choice (when possible) in terms of performance.

## Spacing and proportions for rows and columns

You have fine-grained control over the size, space, and proportions of rows and columns. The **`Height`** and **`Width`** properties of the **`RowDefinition`** and **`ColumnDefinition`** objects can be set with values from the **`GridUnitType`** enumeration as follows:

- **`Auto`:** Automatically sizes to fit content in the row or column.
- **`Star`:** Sizes columns and rows as a proportion of the remaining space.
- **`Absolute`:** Sizes columns and rows with specific, fixed height and width values.

XAML has type converters for the **`GridUnitType`** values, so you simply pass no value for **`Auto`**, a **`*`** for **`Star`**, and the fixed numeric value for **`Absolute`**, such as:

```
<Grid.ColumnDefinitions>
   <ColumnDefinition />
   <ColumnDefinition Width="*"/>
   <ColumnDefinition Width="20"/>
</Grid.ColumnDefinitions>
```

If you wish, you can also explicitly pass **`Width="Auto"`** instead of passing no value.

### Introducing spans

In some situations, you might have elements that should occupy more than one row or column. In these cases, you can assign the **`Grid.RowSpan`** and **`Grid.ColumnSpan`** attached properties with the number of rows and columns a visual element should occupy.

## The Canvas

Sometimes you might need to place controls on the UI in a fixed position, regardless of the dynamic rearrangement. In these cases, you can use the **`Canvas`** panel, which allows for the so-called *absolute positioning* of its child elements, which are placed via the **`Canvas.Top`**, **`Canvas.Left`**, **`Canvas.Bottom`**, and **`Canvas.Right`** attached properties. These represent the distance from the upper margin, the distance from the left margin, the distance from the lower margin, and the distance from the right margin. They must be used in couples: **`Canvas.Top`** with **`Canvas.Left`**, and **`Canvas.Bottom`** with **`Canvas.Right`**. Other combinations are not valid. Consider the following code snippet:

```
<Canvas>
      <TextBlock Text="This text will never change position"
         Canvas.Left="50" Canvas.Top="30"/>
</Canvas>
```

The **`TextBlock`** control is positioned 50 pixels from the left edge, and 30 pixels from the top edge of the canvas. If you run the code, you will notice that the position of the control will never be rearranged as the interface changes.

> 💡 ***Tip: The `Canvas` child elements are spaced from the canvas itself, and not from the window. This means that if the `Canvas.Margin` property is assigned or if the `Canvas` is contained in a `StackPanel` or `WrapPanel`, the location will be evaluated based on the***

***organization of those panels. For this reason, the `Canvas` should be the root panel within a `Window` or `UserControl`.***

## The RelativePanel

The **`RelativePanel`** allows you to arrange elements by specifying their relative positioning with respect to other elements and the panel itself. By default, elements are positioned at the upper-left corner of the panel. You use some attached properties to govern the layout of these elements. Table 5 summarizes these properties.

*Table 5: Attached properties to control the RelativePanel*

| Panel alignment | Relative alignment | Relative Position |
|---|---|---|
| **`AlignTopWithPanel`** | **`AlignTopWith`** | **`Above`** |
| **`AlignBottomWithPanel`** | **`AlignBottomWith`** | **`Below`** |
| **`AlignLeftWithPanel`** | **`AlignLeftWith`** | **`LeftOf`** |
| **`AlignRightWithPanel`** | **`AlignRightWith`** | **`RightOf`** |
| **`AlignHorizontalCenterWithPanel`** | **`AlignHorizontalCenterWith`** | |
| **`AlignVerticalCenterWithPanel`** | **`AlignVerticalCenterWith`** | |

The attached properties listed in the Panel Alignment column of Table 5 are of type **`bool`** and determine the way the child element is aligned with respect to the panel. The attached properties listed in the Relative Alignment column of Table 5 take the name of a relative control as value, and specify the alignment. Attached properties listed in the Relative Position column specify the position of the child element and take the name of the relative control as the value. Code Listing 7 provides an example of **`RelativePanel`**.

*Code Listing 7*

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
        x:Class="Ch4_Panels.RelativePanelWindow"
        Title="RelativePanelWindow">
```

```
        <RelativePanel Background="Gray">

        <Rectangle x:Name="OrangeRectangle" Fill="Orange" Height="44"
                   Width="44"/>
                <Rectangle x:Name="RedRectangleangle" Fill="Green"
                                Height="44" Width="88"
                           RelativePanel.RightOf="OrangeRectangle" />

                <Rectangle x:Name="RedRectangle" Fill="Red"
                                Height="44"
                                RelativePanel.Below="OrangeRectangle"
                           RelativePanel.AlignLeftWith="OrangeRectangle"

           RelativePanel.AlignRightWith="RedRectangleangle"/>
                <Rectangle Fill="Blue"
                                RelativePanel.Below="RedRectangle"
                           RelativePanel.AlignLeftWith="RedRectangleangle"
                                RelativePanel.AlignRightWithPanel="True"

           RelativePanel.AlignBottomWithPanel="True"/>
        </RelativePanel>
</Window>
```

The example is based on some **`Rectangle`** elements, which are an appropriate choice due to the fact they can fill colored areas on screen, and make it easier to understand their relative positioning. As you can see, the attached properties listed in Table 5 allow for specifying the position of each rectangle, with respect to the panel and the other rectangles. Figure 15 shows the result of Code Listing 7.

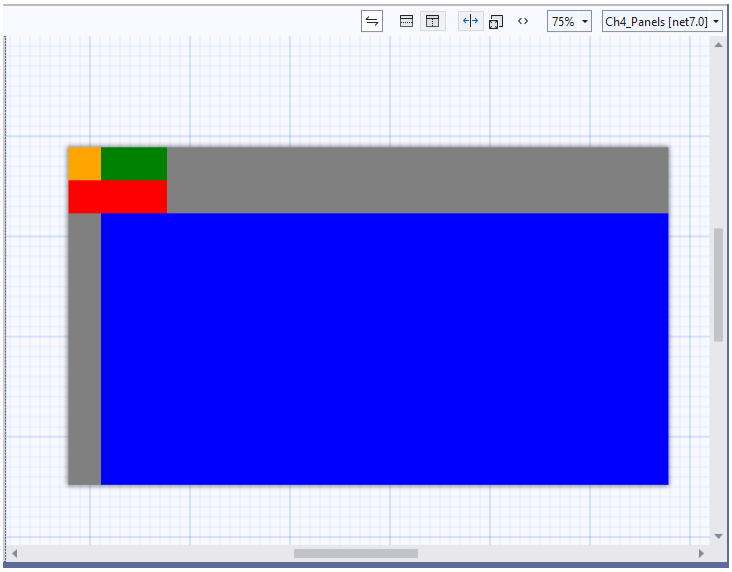



*Figure 15: Arranging visual elements within a RelativePanel*

## The ScrollViewer

The **ScrollViewer** allows you to present content that cannot fit on one screen, and therefore should be scrolled. Technically speaking, this is considered a control, but actually you use it as a panel. Its usage is very simple and is demonstrated in Code Listing 8.

*Code Listing 8*

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
       Height="500"
        x:Class="Ch4_Panels.ScrollViewerWindow"
        Title="ScrollViewerWindow">
       <ScrollViewer Name="Scroll1">
              <StackPanel>
                     <TextBlock Text="My favorite color:"
                                Name="TextBlock1"/>
                     <Rectangle Height="600" Fill="BlueViolet"/>
              </StackPanel>
       </ScrollViewer>
</Window>
```

You basically add a panel or visual element inside the **ScrollViewer** and, at runtime, the content will be scrollable if its area is bigger than the screen size. You can also decide whether to display the scroll bars through the **HorizontalScrollbarVisibility** and **VerticalScrollbarVisibility** properties that can be assigned with self-explanatory values such as **Visible**, **Disable**, and **Hidden**.

Notice how, in order to simulate content that is longer than the actual screen size, a **Height** of **500** has been assigned to the **Window**, whereas a **Height** of **600** has been assigned to the rectangle. Remember to avoid nesting **ScrollViewer** views and to include controls like **DataGrid** and **ListBox** inside a **ScrollViewer**, since these include their own scrolling mechanism.

## The DockPanel

The **DockPanel** container also has many similarities to the **StackPanel**, because it allows you to tile the controls it contains. The main difference is that child controls are docked to the sides of the **DockPanel** in the specified direction, they are also docked to each other, and their size and position can be specified. The typical use of the **DockPanel** is therefore related to the creation of interfaces in which menus and toolbars are connected to one another. The XAML in Code Listing 9 shows how to create a **DockPanel** and how to add a menu bar and a status bar, docked to each other:

*Code Listing 9*

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="500" d:DesignHeight="450"
        x:Class="Ch4_Panels.DockPanelWindow"
        Title="DockPanelWindow">
       <DockPanel LastChildFill="True"
                        VerticalAlignment="Top">
              <Menu DockPanel.Dock="Top"
                        Background="LightGray"
            Name="MainMenu" >
                        <MenuItem Header="File"/>
                        <MenuItem Header="Edit"/>
              </Menu>

              <StackPanel DockPanel.Dock="Top">
                        <TextBlock Text="Command bar" />
              </StackPanel>
       </DockPanel>
</Window>
```

The result of the code is shown in Figure 16.

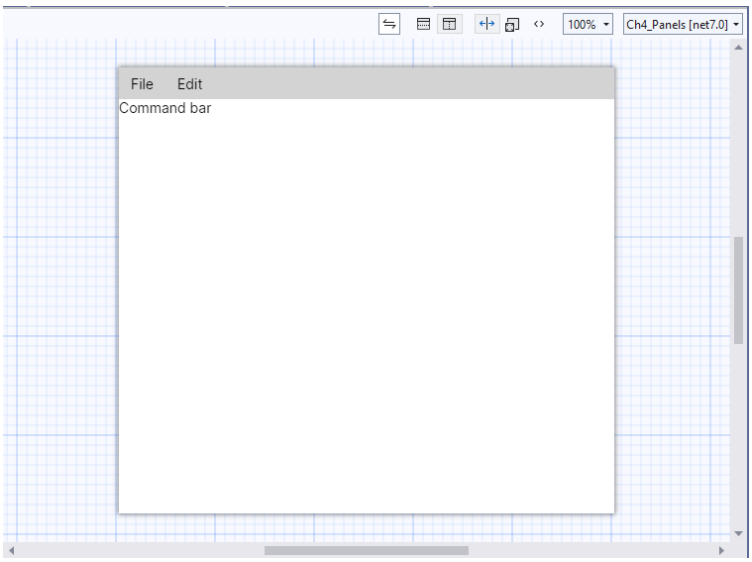



*Figure 16: Arranging visual elements within a DockPanel*

The orientation of the controls contained in the **DockPanel** is set through the **VerticalAlignment** property, whose values can be **Top**, **Bottom**, **Center**, or **Stretch** (to fill the available space). Through the **LastChildFill** property, the container is told whether the controls it contains should fill all the available space. The example assigns **True**, but you can set to **False** to tell the difference.

Finally, by assigning the **DockPanel.Dock** attached property, it is possible to establish the position in which the controls must be docked within the **DockPanel**.

## Chapter summary

Modern apps require dynamic user interfaces that can automatically adapt to the screen size of different device form factors. In Avalonia UI, creating dynamic user interfaces is possible through a number of panels.

The **StackPanel** allows you to arrange controls near one another both horizontally and vertically. The **WrapPanel** does the same, but it is also capable of wrapping visual elements. The **Grid** allows you to arrange controls within rows and columns; the **Canvas** allows you to give controls an absolute position; the **RelativePanel** allows you to arrange controls based on the size and position of other controls or containers; the **ScrollViewer** allows you to scroll the content of visual elements that do not fit in a single page; and the **DockPanel** allows you to dock child elements to the specified edge.

Now that you have a basic knowledge of panels, it’s time to discuss common controls in Avalonia UI that allow you to build the functionalities of the user interface, arranged within the panels you learned in this chapter.

# Chapter 5 Avalonia Common Controls

Avalonia UI ships with a rich set of common controls that you can use to build cross-platform user interfaces easily and without the need for the complexity of platform-specific features. As you can imagine, the benefit of these common controls is that they run on Android, iOS, Windows, and the web browser from the same codebase. In this chapter, you'll learn about common controls, their properties, and their events. Controls whose purpose is displaying data and lists will be covered in the next chapter.

## Working with the companion code

In order to follow the examples in this chapter, you can create a blank Avalonia UI solution or open the solution in the **Ch5_Controls** folder of the companion code repository. If you go for the first option, make sure you follow these steps:

1. Create an Avalonia UI .NET Core App. You can also create a cross-platform solution, but remember that the companion code is based on this template. The name for the new solution is up to you.
2. Make sure you select the latest Avalonia UI version possible, even if it is a pre-release version. For example, controls like the **`SplitButton`** and the **`ToggleSplitButton`** are only available with the latest pre-release version.
3. For each control or group of controls of the same family, add a new item of type Window (Avalonia) to the project, rather than working on the main window.

Any additional steps will be explained where required.

## Understanding controls

In Avalonia UI, a *control* is the building block of any user interface. Put succinctly, a control represents what you would call a widget in Android, a view in iOS, and a control in Windows. Controls derive from the **`TemplatedControl`** class, which inherits from **`Control`**. The **`TemplatedControl`** class provides the possibility to define (and redefine) the appearance of a control, whereas **`Control`** implements any basic property and behavior that controls share. In Chapter 6, "Resources and Data-binding," you will learn more about styles and control templates.

### Controls' common properties

Controls share a number of properties that are important for you to know in advance. They are many, and they are inherited from both **`TemplatedControl`** and **`Control`**, but the most relevant and used are summarized in Table 6.

*Table 6: Controls' common properties*

| Property | Type | Description |
|---|---|---|
| **Width** | **double** | Gets or sets the width of the control |
| **Height** | **double** | Gets or sets the height of the control |
| **Margin** | **Thickness** | Gets or sets the distance of the control from other controls or from its parent |
| **Background** | **IBrush** | Fills the control's background with a brush, like a solid color or gradient |
| **BorderThickness** | **Thickness** | Gets or sets the thickness for the control's border |
| **BorderBrush** | **IBrush** | Assigns the control's border with a brush, like a solid color or gradient |
| **FontFamily** | **FontFamily** | Gets or sets the font type for the text in the control |
| **FontSize** | **double** | Gets or sets the font size |
| **FontWeight** | **FontWeight** | Gets or sets the strength of the font (for example, from **Thin** to **Ultrablack**) |
| **FontStyle** | **FontStyle** | Gets or sets the style of the font (**Normal**, **Italic**, **Oblique**) |
| **Foreground** | **IBrush** | Gets or sets the foreground color |
| **CornerRadius** | **CornerRadius** | The radius of the edges around the control |
| **HorizontalAlignment** | **HorizontalAlignment** | Specifies the horizontal alignment for the control. Supported values are **Center**, **Left**, **Right**, **Stretch** |

| Property | Type | Description |
|---|---|---|
| `VerticalAlignment` | `VerticalAlignment` | Specifies the vertical alignment for the control. Supported values are `Center`, `Bottom`, `Top`, `Stretch` |

Follow these steps to discover values in the `FontFamily`, `FontWeight`, and `FontStyle` enumerations:

1. In C# code, declare the following line:

   ```
   Control ctrl;
   ```

2. Right-click **Control** and select **Go to Definition**.
3. When the source code for the Control class appears, locate the aforementioned properties, right-click the desired enumeration, and select **Go to Definition**.

In this way, you will be able to investigate the type definitions. This actually works with any other .NET type. When you're done, remove the line you wrote previously.

# Introducing common controls

This section provides a high-level overview of common controls offered by Avalonia UI and their most utilized properties. Remember to add the official documentation about the user interface to your bookmarks for a more detailed reference.

## User input with buttons

Most of the time, users interact with the user interface by clicking visual elements. The `Button` control is certainly one of the most-used controls in every user interface, but in Avalonia UI you have several types of buttons available.

> ***Note: Figure 17 groups all the examples related to button controls, so keep this as a reference for the whole paragraph.***

You already saw a couple examples of the `Button`, but here is a quick summary. This control exposes the properties already summarized in Table 6, and you declare it like this:

```
<Button Name="Button1" Content="Click here" Foreground="Orange"
 BorderBrush="Red" BorderThickness="2" CornerRadius="10"
 Click="Button1_Click"/>
```

As you might remember, you can write the click event handler as follows:

```
private void Button1_Click(object sender, RoutedEventArgs e)
```

```
{

}
```

Notice how the content for the button is represented by the **`Content`** property. This is of type **`object`** and can contain any visual element, not just text. For example, you could display an image and some text as the content of a button as follows:

```
<Button>
      <Button.Content>
            <StackPanel Orientation="Horizontal">
                  <Image Source="image.png"/>
                  <TextBlock Text="Custom button"/>
            </StackPanel>
      </Button.Content>
</Button>
```

This is a very powerful feature, because you can create very attractive buttons that can also include videos, complex elements, and so on. Just keep in mind that, most of the time, the simpler the user interface, the better the user experience.

***Tip: As you will discover shortly, there are many controls that give you the ability to display complex visual elements through the `Content` property. These are usually referred to as content controls.***

## Continuous clicks with the RepeatButton

The **`RepeatButton`** is similar to the button, but it continues to raise click events when you keep it pressed. The following is a simple example of **`RepeatButton`**:

```
<RepeatButton Name="Button1" Content="Click here" Click="Button1_Click"/>
```

The only difference with the **`Button`** is that the **`RepeatButton`** keeps raising click events if you keep it pressed, whereas the **`Button`** raises one click event only. They share all the other properties and events, including the way you handle click events.

## Handling states with the ToggleButton

The **`ToggleButton`** implements a checked or unchecked state that can be changed on click by the user. You declare it like this:

```
<ToggleButton Name="Button3" Content="Click to change state"
 IsChecked="true"/>
```

You can programmatically control the checked state via the **`IsChecked`** property, of type **`bool`**. When the state changes, the control raises a **`Click`** event that you can use to detect the state as follows:

```
private void Button3_Click(object sender, RoutedEventArgs e)
```

```
{
    switch(Button3.IsChecked)
    {
        case true:
            // Take an action
            break;
        default:
            // Take an action
            break;
    }
}
```

## Implementing user choice with the RadioButton

The **RadioButton** control is available in most development platforms, so you might already be familiar with it. It allows for providing the user the option to pick up one among multiple choices. The following code shows an example:

```
<RadioButton IsChecked="true" Name="Option1" GroupName="First Group"
 Content="First choice"/>
<RadioButton IsChecked="false" Name="Option2" GroupName="First Group"
 Content="Second choice"/>
<RadioButton IsChecked="false" Name="Option3" GroupName="First Group"
 Content="Third choice"/>
```

The **IsChecked** property determines whether the control is selected. The **GroupName** property allows for logically grouping controls. You can assign a name to each **RadioButton** so that you can interact with it in C# code, as well as binding the **IsChecked** property to a **bool** value (data-binding is discussed later in the book).

In the previous code snippet, all the **RadioButton** controls display some text via the **Content** property, but this property is of type **object** and can contain any visual element to provide a fully customized visual appearance, like regular buttons.

Remember that **RadioButton** controls are mutually exclusive, which means that selecting one will automatically deselect the other controls. Additionally, all the properties discussed in Table 6 also apply to the **RadioButton**.

## Spinning values with the ButtonSpinner

The **ButtonSpinner** is a control that includes a box where you can show some content or values and two buttons, up and down, that allow for increasing or decreasing the value. The following example shows how to implement a **ButtonSpinner** that allows for increasing or decreasing a numerical value:

```
<ButtonSpinner Margin="10" Name="ButtonSpinner1" AllowSpin="True"
 ButtonSpinnerLocation="Right" Content="1" Spin="ButtonSpinner1_Spin" />
```

**AllowSpin** with **True** enables the buttons, whereas **ButtonSpinnerLocation** can be set with **Right** (default) or **Left** and specifies the position of the up and down buttons.

The **Content** property specifies the value you display. Remember that this is a property of type **object**, so you will need to provide the appropriate conversion in code-behind. For example, supposing you want to allow for increasing or decreasing an integer value based on the previous XAML, you will handle the **Spin** event as follows:

```
private void ButtonSpinner1_Spin(object sender, SpinEventArgs e)
{
    int content = Convert.ToInt32(ButtonSpinner1.Content);
    switch (e.Direction)
    {
        case SpinDirection.Increase:
            content++;
            ButtonSpinner1.Content = content;
            break;
        case SpinDirection.Decrease:
            content--;
            ButtonSpinner1.Content = content;
            break;
    }
}
```

The **SpinEventArgs** class provides the **SpinDirection** property, which you can use to detect the spin direction. Values can be **Increase** or **Decrease**. The previous code snippet converts the original value into an integer, increases or decreases the number, and then reassigns the result to the **Content** property. Figure 17 shows how the **ButtonSpinner** appears.

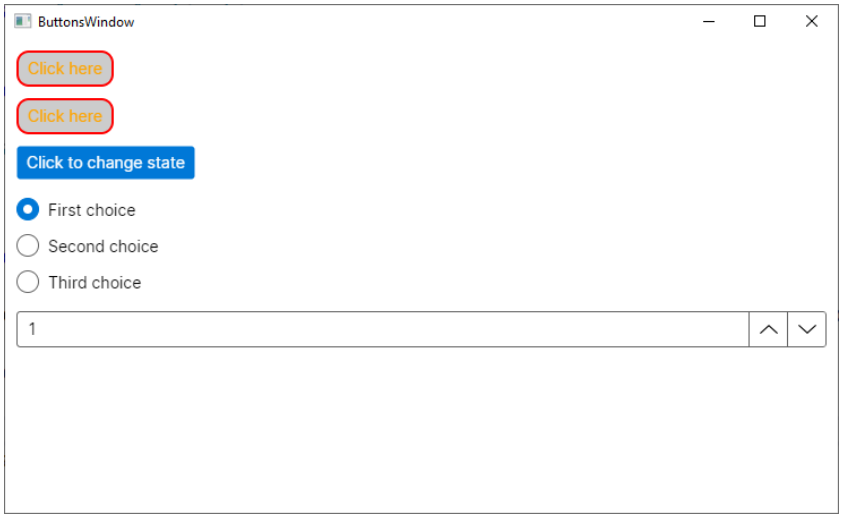



*Figure 17: Different types of buttons in Avalonia UI*

## Working with text

Displaying text and requesting input from the user in the form of text is extremely common in every application. Avalonia UI provides the following controls for displaying and editing text:

- **`TextBlock`**, which allows for displaying read-only text.
- **`TextBox`**, which allows for entering and editing text.
- **`MaskedTextBox`**, which can be used to display or edit text using a mask to distinguish between proper and improper user input formats.
- **`AutoCompleteBox`**, which provides a text box for user input and a dropdown list that contains possible matches based on the input in the text box.

The **`TextBlock`** control displays read-only text, and it exposes some useful properties, as shown in the following XAML:

```
<TextBlock Text="Displaying some text" HorizontalAlignment="Center"
 VerticalAlignment="Center" TextWrapping="Wrap" Foreground="Blue" />
```

**`TextWrapping`** allows you to wrap a long string and can be assigned a value from the **`TextWrapping`** enumeration. For example, **`Wrap`** splits a long string into multiple lines proportionate to the available space. **`WrapWithOverflow`** splits a long string into multiple lines if the line overflows the available block width. If not specified, **`NoWrap`** is the default.

**`HorizontalAlignment`** and **`VerticalAlignment`** specify the horizontal and vertical alignment for the text. The result of the previous snippet is shown in Figure 18 (which also includes other text-related controls).

The **`TextBox`** control allows you to enter unformatted text. Its relevant properties are summarized in Table 7.

*Table 7: TextBox properties*

| Property | Type | Description |
|---|---|---|
| **`Text`** | **`string`** | Gets or sets the input text |
| **`PasswordChar`** | **`char`** | Hides any character in the text box, replacing them with the specified character; useful to implement a password box |
| **`Watermark`** | **`string`** | Specifies a placeholder text |
| **`AcceptsReturn`** | **`bool`** | Gets or sets whether pressing Enter should add a new line |
| **`AcceptsTab`** | **`bool`** | Gets or sets whether pressing Tab has an effect inside the control |

| Property | Type | Description |
|---|---|---|
| **Foreground** | **IBrush** | Gets or sets the foreground brush |
| **TextWrapping** | **TextWrapping** | Determines how long text is sent to a new line; possible values are **Wrap** (splits a long string into multiple lines), **WrapWithOverflow** (splits a long string into multiple lines if the line overflows the available block width), **NoWrap** |

Following is an example:

```
<TextBox Name="TextBox1" Watermark="Enter some text..."
   Foreground="Green" TextInput="TextBox1_TextInput"/>
```

This control also exposes the **TextInput** event which is fired after the control loses focus. You handle it the usual way, as follows:

```
private void TextBox1_TextInput(object sender, TextInputEventArgs e)
{
    string inputText = e.Text;
}
```

The event handler takes an object of type **TextInputEventArgs** as the second parameter, which exposes the **Text** property that stores the content of the text box after it loses focus. The following example demonstrates how to implement a password box:

```
<TextBox RevealPassword="False" PasswordChar="*" Name="TextBox1"/>
```

With **PasswordChar** assigned, every character is replaced with the specified symbol. **RevealPassword** can be assigned with **true** to show the real content of the **TextBox**, whereas when **false**, it keeps it hidden. Figure 18 includes an example of **TextBox** (the second from top).

***Tip: By default, the `TextBox` is displayed with a black border and rounded corners. You can change this by assigning the `BorderBrush`, `BorderThickness`, and `CornerRadius` properties, according to your needs. For example, you could hide the border by simply assigning `0` to `BorderThickness`.***

The **MaskedTextBox** is similar to the **TextBox**, but the input is constrained by a mask that specifies the input format. For example, the following code declares two **MaskedTextBox** controls, where the first one requires a phone number in the international format, and the second one requires the input of an Italian VAT code (made of 11 numbers):

```
<StackPanel>
  <TextBlock Margin="5">International phone number:</TextBlock>
  <MaskedTextBox Margin="5" Mask="(+09) 000 000 0000" />
  <TextBlock Margin="5">Italian VAT number:</TextBlock>
  <MaskedTextBox Margin="5" Mask="IT 000 000 000 00" />
</StackPanel>
```

The constraint format is specified via the **Mask** property. The result of the code is visible in Figure 18, where you can see how the possibility of typing characters is restricted based on the specified constraint. You can retrieve the content of the **MaskedTextBox** by accessing its **Text** property or by handling the **TextInput** event. Table 8 contains the list of supported characters for the mask.

*Table 8: Possible masks*

| Property | Description |
|---|---|
| **0** | Required digit; accepts any single digit between 0 and 9 |
| **9** | Optional digit |
| **#** | Optional digit or space; if blank in the mask, a space will be placed in the **Text** property |
| **L** | Required letter, with input restricted to the ASCII letters a-z and A-Z |
| **?** | Optional letter, with input restricted to the ASCII letters a-z and A-Z |
| **&** | Required character |
| **C** | Optional character |
| **A** | Required alphanumeric value |
| **a** | Optional alphanumeric value |
| **.** | Decimal placeholder shown based on the decimal symbol appropriate to the control's format provider |

| Property | Description |
|---|---|
| , | Thousands placeholder shown based on the decimal symbol appropriate to the control's format provider |
| : | Time separator shown based on the decimal symbol appropriate to the control's format provider |
| / | Date separator shown based on the decimal symbol appropriate to the control's format provider |
| $ | Currency symbol shown based on the decimal symbol appropriate to the control's format provider |
| < | Shifts all following characters to lowercase |
| > | Shifts all following characters to uppercase |
| \| | Disables a previous shift |
| \ | Escapes a mask character, turning it into a literal |

The last control that allows for working with text is the **AutoCompleteBox**. The purpose of this control is to simplify user input by showing suggestions from a list as the user types. The list is prepopulated, so you should use it when you use the **AutoCompleteBox** for a specific topic, and not for generic input. The following XAML shows how to declare an **AutoCompleteBox**:

```
<StackPanel Margin="5">
      <TextBlock>Select your favorite food:</TextBlock>
      <AutoCompleteBox Name="FoodBox" FilterMode="StartsWith" />
</StackPanel>
```

In C# code, for example in the constructor of the window, you can populate the **Items** property as follows:

```
FoodBox.Items = new string[] { "Pizza", "Caesar salad", "Hot dog",
                "Seafood" };
```

The **Items** property could also be dynamically populated via data-binding rather than being populated in C# code with fixed values, but data-binding has not been discussed yet, and will be a topic of Chapter 7, "Resources and Data-Binding."

Figure 18 shows the result of this code, and as you can see, matching suggestions are coming as the user types.

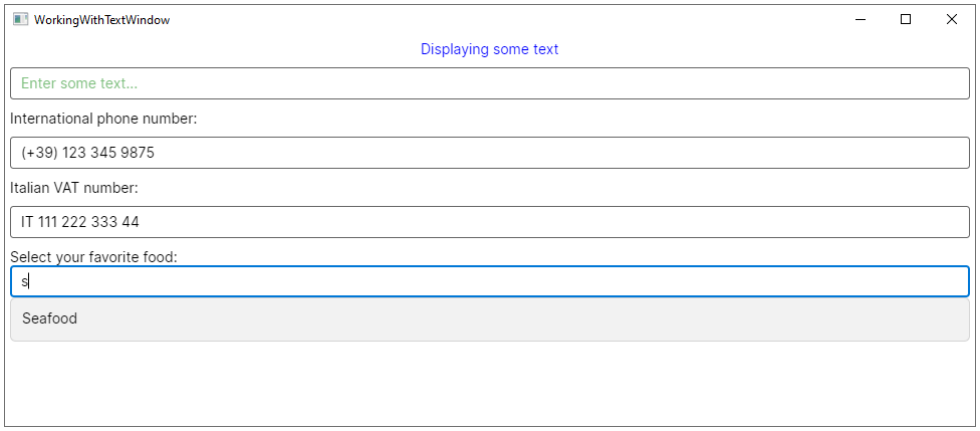



*Figure 18: Controls for working with text*

The way suggestions are provided is determined by the **`FilterMode`** property. Luckily enough, possible value names are self-explanatory and do not require any further explanation. They are **`StartsWith`**, **`StartsWithCaseSensitive`**, **`StartsWithOrdinal`**, **`StartsWithOrdinalCaseSensitive`**, **`Contains`**, **`ContainsCaseSensitive`**, **`ContainsOrdinal`**, **`ContainsOrdinalCaseSensitive`**, **`Equals`**, **`EqualsCaseSensitive`**, **`EqualsOrdinal`**, and **`EqualsOrdinalCaseSensitive`**.

## Working with dates and time

Another common requirement in most applications is working with dates and time: Avalonia UI provides the **`DatePicker`**, **`Calendar`**, **`CalendarDatePicker`**, and **`TimePicker`** controls for that. Both the **`DatePicker`** and the **`Calendar`** allow for date selection, but with a different visual style. The **`DatePicker`** exposes the **`SelectedDate`**, **`MinYear`**, and **`MaxYear`** properties that represent the selected/current date, the minimum date, and the maximum date, respectively, all of type **`DateTime`**. It exposes an event called **`SelectedDateChanged`**, which is raised when the user selects a date. You can handle this to retrieve the value of the **`SelectedDate`** property.

The view can be declared as follows:

```
<DatePicker Name="DatePicker1" MaxYear="12/31/2023" MinYear="01/01/1990"
 SelectedDateChanged="DatePicker1_SelectedDateChanged"/>
```

And then in the code-behind, you can retrieve the selected date like this:

```
private void DatePicker1_SelectedDateChanged(object sender,
    DatePickerSelectedValueChangedEventArgs e)
{
    DateTimeOffset? newDate = e.NewDate;
    DateTimeOffset? oldDate = e.OldDate;
}
```

The **DatePickerSelectedValueChangedEventArgs** object stores the selected date in the **NewDate** property, and the previous date in the **OldDate** property. Both properties are of type **DateTimeOffset?**. Figure 19 shows the **DatePicker** in action.

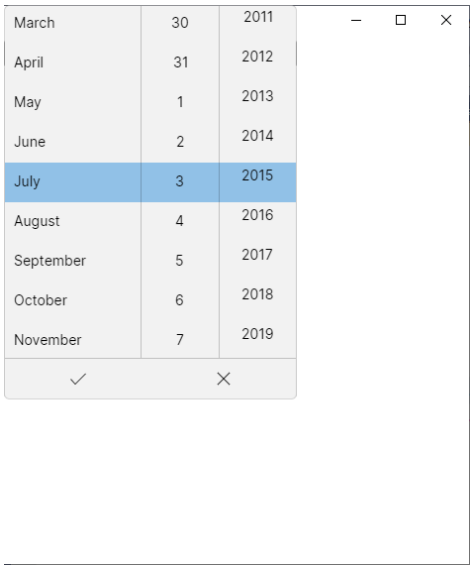



*Figure 19: The DatePicker in action*

The **Calendar** is declared as follows:

```
<Calendar Name="Calendar1" DisplayDate="07/03/2023"
 DisplayDateStart="01/01/2023" DisplayDateEnd="12/31/2023"
 DisplayMode="Month" FirstDayOfWeek="Monday"
 DisplayDateChanged="Calendar1_DisplayDateChanged"/>
```

The **DisplayDate** property sets the default date, whereas **DisplayDateStart** and **DisplayDateEnd** delimit the date range for the calendar. **DisplayMode** allows you to set the view by **Month**, **Year**, and **Decade**.

You can also specify the first day of the week via the **FirstDayOfWeek** property. This can also be assigned in code-behind if you want to determine the user's settings with the following line:

```
Calendar1.FirstDayOfWeek = System.Threading.Thread.
          CurrentThread.CurrentCulture.DateTimeFormat.FirstDayOfWeek;
```

When the user selects a date, the **DisplayDateChanged** event is fired, and can be handled like in the following example:

```
private void Calendar1_DisplayDateChanged(object sender,
    CalendarDateChangedEventArgs e)
{
    DateTime? newDate = e.AddedDate;
    DateTime? oldDate = e.RemovedDate;
}
```

The **AddedDate** and **RemovedDate** property from the **CalendarDateChangedEventArgs** object return the newly selected date and the previously selected date, respectively, as **DateTime?** objects. Figure 20 shows the **Calendar** in action.

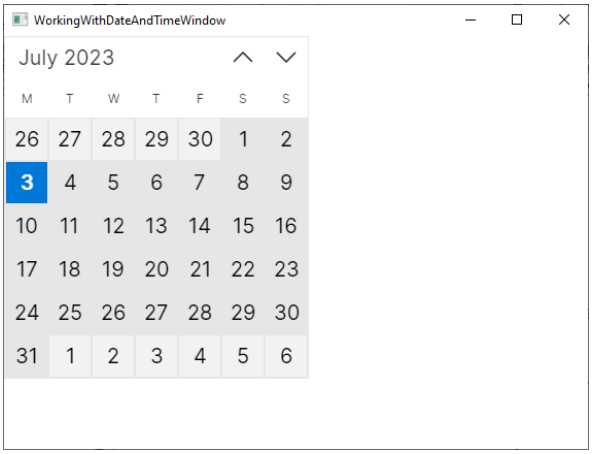



*Figure 20: The Calendar in action*

The third and last control for working with dates is the **CalendarDatePicker**, which has the same properties and events of the **DatePicker**, but it shows a calendar instead of a selector. Figure 21 shows how the **CalendarDatePicker** appears, based on the same property assignments of the previous **DatePicker**.

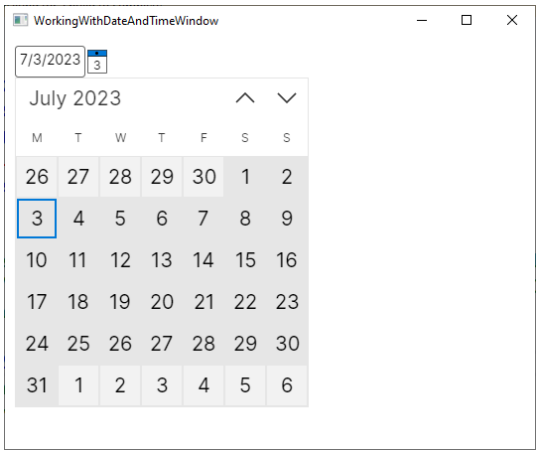



*Figure 21: The CalendarDatePicker in action*

By default, the view is collapsed, and it only shows the date box and the calendar icon. When you click this icon, the calendar view is expanded.

The **TimePicker** allows for selecting a time in the day. The following snippet shows an example:

```
<TimePicker Name="TimePicker1" MinuteIncrement="15" SelectedTime="02:30:00"
 SelectedTimeChanged="TimePicker1_SelectedTimeChanged"/>
```

You can set and read the time for the control via the **SelectedTime** property. The **MinuteIncrement** property allows you to specify the minute increment in the selector. By default, the **TimePicker** shows a 12-hour, AM/PM clock, but you can change this behavior by assigning the following property value:

```
ClockIdentifier="24HourClock"
```

When the user selects a time, the **SelectedTimeChanged** event is fired, and you can handle it like in the following example:

```
private void TimePicker1_SelectedTimeChanged(object sender,
    TimePickerSelectedValueChangedEventArgs e)
{
    TimeSpan? newTime = e.NewTime;
    TimeSpan? oldTime = e.OldTime;
}
```

The **TimePickerSelectedValueChangedEventArgs** exposes the **NewTime** and **OldTime** properties, both of type **TimeSpan?**, which return the newly selected time and the previously selected time, respectively. Figure 21 shows how the **TimePicker** appears.

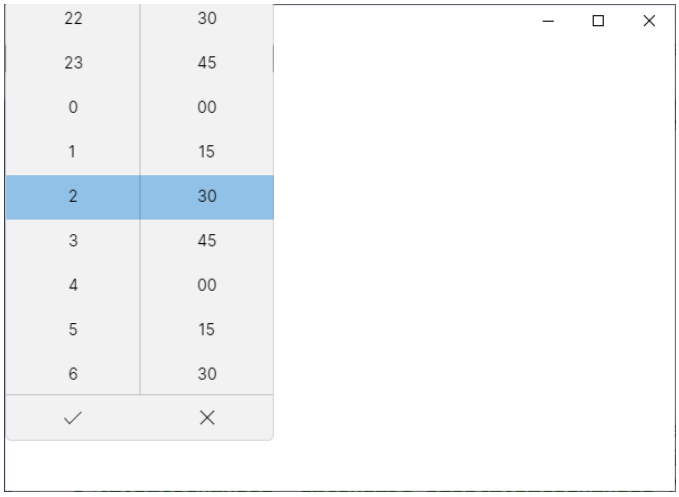



*Figure 22: The TimePicker in action*

***Tip: All four controls described so far also expose a property called `Header`, of type `string`, which you can assign with text that is displayed above the control.***

## Implementing value selection: CheckBox, Slider, ComboBox

Avalonia UI offers a number of controls for user input based on selecting values. The first of them is the **CheckBox**, which exposes the **IsChecked** property, whose value is **true** or **false**. However, if you assign the **IsThreeState** property as **true**, **IsChecked** also supports a null value. The following snippet provides an example:

```
<CheckBox Name="CheckBox1" Content="Do you agree?" IsChecked="true"
 IsThreeState="true"/>
```

Notice that you are not limited to showing a label for the **CheckBox**, because this exposes the **Content** property, making it possible to implement a composite view as the control description. You also access the control's value via its **IsChecked** property. Figure 23 shows what it looks like.

The **Slider** allows the input of a linear value. It exposes the **Value**, **Minimum**, and **Maximum** properties, all of type **double**, which represent the current value, minimum value, and maximum value. It does not have a built-in label, so you can use it together with a **TextBlock** as follows:

```
<StackPanel Margin="10">
    <TextBlock Text="Select your age:" />
    <Slider x:Name="Slider1" Maximum="85" Minimum="13" Value="30"
            PropertyChanged="Slider1_PropertyChanged" TickFrequency="1"
            TickPlacement="TopLeft" Ticks="1"/>
</StackPanel>
```

Surprisingly, no event is fired when the value changes, so the only alternative to handle this is working with the **PropertyChanged** event as follows:

```
private void Slider1_PropertyChanged(object sender,
    AvaloniaPropertyChangedEventArgs e)
{
    if(e.Property.Name == nameof(Slider1.Value))
    {
        // Handle Value here...
    }
}
```

**TickFrequency** is a **double** that defines the interval between ticks on the bar, whereas **TickPlacement** specifies where the tick bar should be placed (**TopLeft**, **BottomRight**, **Outside**, **None**). An example of the slider can be seen in Figure 23.

The next control for value selection is the **ComboBox**. The following is an example of how you can declare it:

```
<ComboBox SelectedIndex="0" IsDropDownOpen="True">
      <ComboBoxItem>Item 1</ComboBoxItem>
      <ComboBoxItem>Item 2</ComboBoxItem>
      <ComboBoxItem>Item 3</ComboBoxItem>
      <ComboBoxItem>Item 4</ComboBoxItem>
</ComboBox>
```

The following is a list of relevant points:

- The **ComboBox** can be populated with items programmatically or via data-binding. In the first case, you add as many **ComboBoxItem** objects as many items you want to display.

In the second case, as you will learn in the next chapter, you assign the **`ItemsSource`** property with a collection.

- The **`SelectedIndex`** property allows you to get and set the current item if the control is populated programmatically; otherwise, you use the **`SelectedItem`** property.
- The **`IsDropDownOpen`** property allows you to automatically expand the view when assigned with **`true`**.
- The **`ComboBoxItem`** object is not limited to displaying strings; it can display complex visual elements through the usual XAML hierarchical structure.

When the selection changes, the **`SelectionChanged`** event is fired. The following code snippet shows a sample handler:

```
private void ComboBox1_SelectionChanged(object sender,
        SelectionChangedEventArgs e)
{
    var selectedItems = e.AddedItems;
}
```

**`AddedItems`** is of type **`IList`**, so you will need to cast the result to the expected type. Figure 23 shows how the **`ComboBox`** appears, along with the controls described previously.

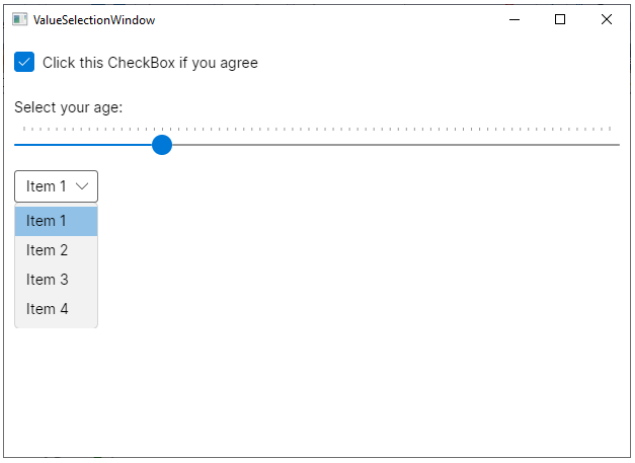



*Figure 23: A summary view of the CheckBox, Slider, and ComboBox controls*

## Working with images

Using images is very common in most applications since they both enrich the look and feel of the user interface and enable apps to support multimedia content. Avalonia UI provides an **`Image`** control you can use to display images from local files and embedded resources. Displaying images is really simple, while understanding how you load and size images is more complex, especially if you have no previous experience with XAML and dynamic user interfaces.

You declare an **`Image`** as follows, obviously replacing the image file name with a name of your choice:

```
<Image Source="avares://Ch5_CommonControls/Assets/yourImage.png"
Stretch="UniformToFill" Width="300"/>
```

The way you specify the image path depends on how and where the image is stored. If it is a resource in the application, you will need to assign the **Build Action** property with **AvaloniaResource** in the **Properties** window. The syntax you use is the following:

```
Source="avares://rootNamespace/folder/filename"
```

**avares://** is a prefix followed by the project's root namespace, a slash with the name of the folder where the image file is, a slash, and the file name. If the file is not a resource and is stored locally on disk, the syntax you use is the following:

```
Source="folder/filename"
```

If you need to display images that are loaded at runtime, you can instantiate a **Bitmap** object and assign the result to the **Source** property as follows:

```
var image = new Image();
var bmp = new Bitmap("folder/imageFileName.jpg");
image.Source = bmp;
// Stack1 is a StackPanel declared in XAML
Stack1.Children.Add(image);
```

The **Stretch** property determines how to size and stretch an image within the bounds it is being displayed in. It requires a value from the **Avalonia.Media.Stretch** enumeration:

- **Fill:** Stretches the image to fill the display area completely and exactly. This may result in the image being distorted.
- **Uniform:** The content is resized to fit the destination dimensions while preserving its native aspect ratio.
- **UniformToFill:** The content is resized to completely fill the destination rectangle while preserving its native aspect ratio.
- **None:** The image is not resized.

You can control the stretch direction by assigning the **StretchDirection** property with **DownOnly** (scales the image downwards), **UpOnly** (scales the image upwards), or **Both** (uses both directions). You can also set the **Width** and **Height** properties to adjust the size of the **Image** control. Figure 24 shows an example.


*Figure 24: Displaying images with the Image view*

Supported image formats are .jpg, .png, .gif, .bmp, and .tif.

## Working with menus

You can create top-level menus with the **Menu** control. This is a common feature for desktop applications. The following XAML code demonstrates how to define a menu with two items and nested commands:

```
<DockPanel VerticalAlignment="Top">
      <Menu DockPanel.Dock="Top" Background="LightGray">
            <MenuItem Header="_File">
                  <MenuItem Header="_Open..." Name="OpenButton"
                   Click="OpenButton_Click">
                        <MenuItem.Icon>
                              <Image
                         Source="avares://Ch5_Controls/Assets/OpenFile.png"/>
                        </MenuItem.Icon>
                  </MenuItem>
                  <Separator/>
                  <MenuItem Header="_Exit"/>
            </MenuItem>
            <MenuItem Header="_Edit">
                  <MenuItem Header="Copy"/>
                  <MenuItem Header="Paste"/>
            </MenuItem>
      </Menu>
</DockPanel>
```

These are the most relevant points about menus:

- A **Menu** is normally included in a **DockPanel** because it arranges child elements most appropriately.
- A **Menu** contains **MenuItem** objects, each representing a real menu.
- The **Header** property of the **MenuItem** shows the name on the bar.
- Each **MenuItem** contains child **MenuItem** objects that point to specific actions.
- A **MenuItem** exposes the **Click** event you can handle to take the appropriate action at button click. It is also possible to bind the **Command** property to an action in a ViewModel, but this will be explained in the next chapter. Look back at the description for the **Button** control to understand how the **Click** event can be handled.
- You can separate menu items using the **Separator** control.
- You can provide an icon by assigning an **Image** control to the **MenuItem.Icon** property. Look back at the **Image** control description about the syntax.

Notice how you can add an underscore (_) before the letter you want to use as a shortcut. For example, pressing **Alt + F** will open the File menu item. In addition, you could assign the **InputGesture** property with a keyboard shortcut as in the following line:

```
<MenuItem Header="Paste" InputGesture="Ctrl+V"/>
```

Also, remember that you have complete control over colors and fonts using the properties discussed in Table 6. Figure 25 shows how the menu appears.


*Figure 25: A menu in Avalonia UI*

***Tip: If you declare a `CheckBox` as the content for the `MenuItem.Icon` property, you will be able to enable or disable that menu item, depending on the `CheckBox` value.***

## Working with flyouts

Flyouts are containers similar to pop-up windows that can be used to display and dismiss arbitrary UI contents. They have become very popular with mobile applications, but they can still be useful in desktop apps as well. They are not used as individual controls; rather, they represent the content of two controls in particular: the **`Button`** and the **`SplitButton`**.

You assign a **`Flyout`** element to the same-named property of the **`Button`** like in the following example:

```
<Button Margin="20" Name="FlyoutButton" Content="Click for privacy policy">
      <Button.Flyout>
            <Flyout Placement="Bottom">
                <TextBlock
                  Text="The application will not collect any personal data"/>
            </Flyout>
      </Button.Flyout>
</Button>
```

With the **`Flyout`** you can show additional UI content. In this case, when the user clicks the button, a pop-up window appears and shows the provided text, as shown in Figure 26.


*Figure 26: A Flyout opened with a Button*

As you would expect, the **`Flyout`** is not limited to displaying text, so you can provide complex visual elements as their content. The flyout position is determined via the **`Placement`** property, of type **`FlyoutPlacementMode`**. Table 9 describes possible values.

*Table 9: Flyout placement modes*

| Value | Description |
|---|---|
| **`Auto`** | Preferred location is determined automatically. |
| **`Bottom`** | Preferred location is below the target element. |
| **`BottomEdgeAlignedLeft`** | Preferred location is below the target element, with the left edge of the flyout aligned with the left edge of the target element. |
| **`BottomEdgeAlignedRight`** | Preferred location is below the target element, with the right edge of the flyout aligned with the right edge of the target element. |
| **`Left`** | Preferred location is to the left of the target element. |
| **`LeftEdgeAlignedBottom`** | Preferred location is to the left of the target element, with the bottom edge of the flyout aligned with the bottom edge of the target element. |

| Value | Description |
|---|---|
| **LeftEdgeAlignedTop** | Preferred location is to the left of the target element, with the top edge of the flyout aligned with the top edge of the target element. |
| **Right** | Preferred location is to the right of the target element. |
| **RightEdgeAlignedBottom** | Preferred location is to the right of the target element, with the bottom edge of the flyout aligned with the bottom edge of the target element. |
| **RightEdgeAlignedTop** | Preferred location is to the right of the target element, with the top edge of the flyout aligned with the top edge of the target element. |
| **Top** | Preferred location is above the target element. |
| **TopEdgeAlignedLeft** | Preferred location is above the target element, with the left edge of the flyout aligned with the left edge of the target element. |
| **TopEdgeAlignedRight** | Preferred location is above the target element, with the right edge of the flyout aligned with the right edge of the target element. |

You can then control the **Flyout** behavior via the **ShowMode** property, of type **FlyoutShowMode**, whose values are listed in Table 10.

*Table 10: Flyout show modes*

| Value | Description |
|---|---|
| **Standard** | The flyout is shown and dismissed on click. |
| **Transient** | The flyout is shown and dismissed when the parent gets focus. |
| **TransientWithDismissOnPointerMoveAway** | The flyout is shown on click but dismissed when the mouse pointer moves away. |

Finally, the **`IsOpen`** property can be used to determine if the pop-up window is open, and to close it programmatically. A more interesting usage of the **`Flyout`** is done with the **`SplitButton`**, as you will learn next.

## The SplitButton

The **`SplitButton`** is a special control made of two parts: the content and a flyout. Figure 27 shows an example that you can take as a reference before writing code.

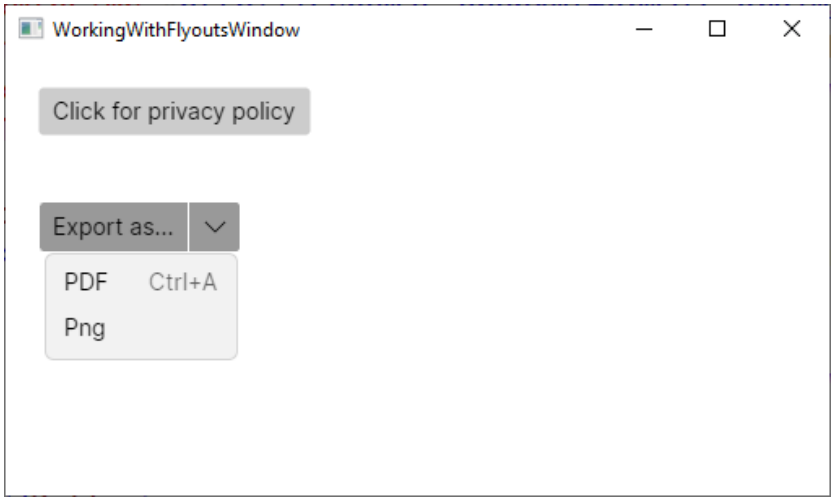



*Figure 27: A SplitButton in action*

As you can see, the **Export as...** button shows a pop-up window to provide further options. Figure 27 shows the result of the following code:

```
<SplitButton Margin="20" Content="Export as..." >
      <SplitButton.Flyout>
            <MenuFlyout Placement="Bottom">
                  <MenuItem Header="PDF"
                              InputGesture="Ctrl+A" />
                  <MenuItem Header="Png" />
            </MenuFlyout>
      </SplitButton.Flyout>
</SplitButton>
```

The **`SplitButton`** supports a flyout called **`MenuFlyout`**, which makes it possible to provide options. The **`Placement`** values are the same as described in Table 9. Options are provided via **`MenuItem`** objects that you saw when working with menus, which means you can handle actions via the **`Click`** event (or by binding the **`Command`** property if you work with data-binding). This is a very interesting control that you can use in a variety of scenarios that simplifies the way you can provide interaction over options.

## The SplitView

The **SplitView** allows for the implementation of a master-detail view with a side pane and some content. Figure 28 shows an example, with considerations following shortly.

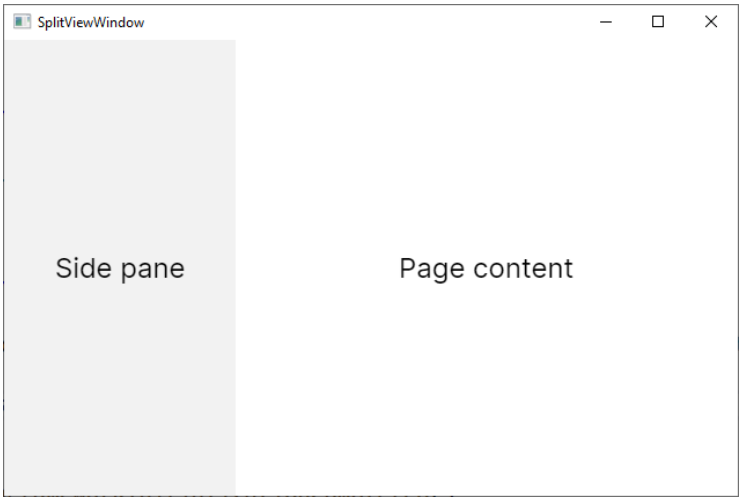



*Figure 28: Organizing a window with the SplitView*

The result in Figure 28 is obtained via the following XAML code:

```
<SplitView IsPaneOpen="True" DisplayMode="Inline" OpenPaneLength="200">
      <SplitView.Pane>
            <TextBlock Text="Side pane" FontSize="24"
                       VerticalAlignment="Center"
                       HorizontalAlignment="Center"/>
      </SplitView.Pane>

      <Grid>
            <TextBlock Text="Page content" FontSize="24"
                       VerticalAlignment="Center"
                       HorizontalAlignment="Center"/>
      </Grid>
</SplitView>
```

The side pane is defined by implementing visual elements as the content of the **SplitView.Pane** property. The **OpenPaneLength** property specifies the width of the pane, and it will keep it fixed, while the rest of the window is rearranged dynamically. The **DisplayMode** property, of type **SplitViewDisplayMode**, supports values described in Table 11.

*Table 11: Pane display modes*

| Value | Description |
|---|---|
| **CompactInline** | The pane is displayed next to the content. When collapsed, the pane is still visible according to **CompactPaneLength**. The pane does not automatically collapse when tapped outside. |
| **CompactOverlay** | The pane is displayed above the content. When collapsed, the pane is still visible according to **CompactPaneLength** and collapses when the user taps outside of it. |
| **Inline** | The pane is displayed next to the content and does not collapse automatically when the user taps outside of it. |
| **Overlay** | The pane is displayed above the content and collapses when a user taps outside of it. |

All the visual elements you add to the **SplitView** pane or content behave exactly as they would outside this container. This can be a convenient UI implementation for cross-platform projects or mobile apps.

## Organizing views with the Expander

The Expander is a control that can be collapsed or expanded. When expanded, it shows additional visual elements. The following code provides an example:

```
<Expander Header="Expand for full view">
      <StackPanel>
          <TextBlock Text="You have expanded the view to see this image:"/>
          <Image Width="200"
           Source="avares://Ch5_Controls/Assets/xamarin-forms-succinctly.png"
                    Stretch="Fill"/>
      </StackPanel>
</Expander>
```

The **Header** property can be a string or another visual element. In this case, you use the extended **Expander.Header** form. It also exposes the **IsExpanded** property, of type **bool**, that gets or sets whether the control is expanded. Figure 29 shows an example.

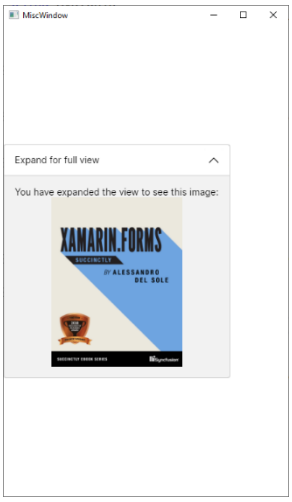



*Figure 29: Organizing the UI with the Expander*

The **Expander** is a convenient choice when you have several pieces of user interface that do not need to be displayed all at the same time.

## Showing progress with the ProgressBar

The **ProgressBar** is another popular control that allows you to display the progress of a long-running operation. You declare it as follows:

```
<ProgressBar Minimum="0" Maximum="100" Foreground="LightGray"
             Value="70" ShowProgressText="True"/>
```

You specify the **Minimum** and **Maximum** values, and the **Value** property will likely be updated at runtime by your C# logic. If you do not specify a **Foreground** color, a shade of blue will be used by default.

If you set the **ShowProgressText** property as **True**, the **ProgressBar** will show the value with a percentage sign (%). This behavior cannot be changed, but at least it is not mandatory, and you could implement a different visualization option via a **TextBlock**. Figure 30 shows what the **ProgressBar** looks like.

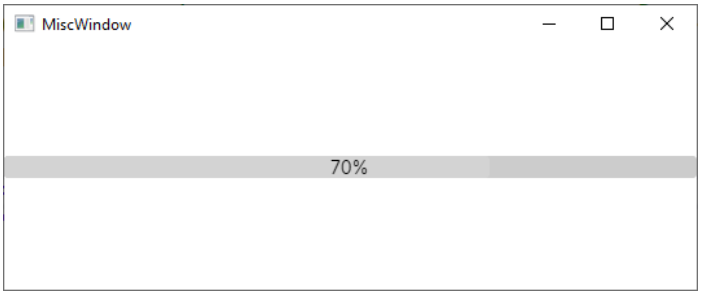



*Figure 30: Implementing a ProgressBar*

The **LightGray** color has been used here to better render this page.

## Chapter summary

This chapter introduced the concept of the control in Avalonia UI, the building blocks for any user interface. You have seen the different types of buttons, the different controls you use to display and edit text, how you can manipulate dates and time, how you can implement value selection, and how to work with images and menus.

Now you have all you need to build high-quality user interfaces with panels and controls, but there is more that you need to know to really enrich your applications. In the next chapter, you will make an important step forward, understanding how to consume styles and resources, and how to display data sets via data binding.

# Chapter 6  Resources and Data Binding

XAML is a very powerful declarative language, and it shows all of its power with two particular scenarios: working with resources and working with data binding. If you have existing experience with platforms like WPF, .NET MAUI, and UWP, you will be familiar with the concepts described in this chapter, though with some differences that are highlighted where appropriate. If this is your first time, you will immediately appreciate how XAML simplifies difficult things in both scenarios.

## Working with resources

Generally speaking, in XAML-based platforms such as WPF, UWP, .NET MAUI, and Avalonia UI, resources are reusable pieces of information that you can apply to visual elements in the user interface. Typical XAML resources are styles, control templates, object references, and data templates. Avalonia UI supports styles, control templates, and data templates, which we'll cover in this chapter.

***Tip: Resources in XAML are very different from resources in platforms such as Windows Forms, where you typically use .resx files to embed strings, images, icons, or files. My suggestion is that you should not make any comparison between XAML resources and other .NET resources.***

### Declaring resources

Every **`Window`** object and layout exposes a property called **`Resources`**, a collection of XAML resources that you can populate with one or more supported resources, such as reusable references and data templates:

```
<Window.Resources>
      <SolidColorBrush Color="Red" x:Key="RedResx"/>
</Window.Resources>
```

In this case, a solid color is declared as a resource and can be referenced across the whole window. An identifier for the resource must be provided via the **`x:Key`** literal. A resource can then be consumed as follows:

```
<TextBlock Foreground="{StaticResource RedResx}"
           Text="Hi from Avalonia UI!"/>
```

You point to a resource with one of these two options:

- **`StaticResource`**: This assigns the resource as a one-time approach. If the value of the resource changes at runtime, the target is not updated.
- **`DynamicResource`**: If the value of the resource is updated at runtime, the target is automatically updated.

Resources have scope. This implies that resources you add to the page level are available to the whole page, whereas resources you add to the layout level are only available to the current layout, like in the following snippet:

```
<StackPanel.Resources>
    <!-- Resources are available only to this panel, not outside -->
</StackPanel.Resources>
```

Sometimes you might want to make resources available to the entire application. In this case, you can take advantage of the **App.axaml** file as follows:

```
<Application.Resources>
    <!-- Resources are available across the app -->
</Application.Resources>
```

Resources you put inside this resource dictionary will be visible to any window, layout, or view in the application. It is also possible to organize resources within *resource dictionaries*. These are .axaml files that only contain resources. To create one, right-click the project name in Solution Explorer, select **Add** > **New Item**. In the **Add New Item** dialog, select the **Resource Dictionary (Avalonia)** item, as shown in Figure 31.

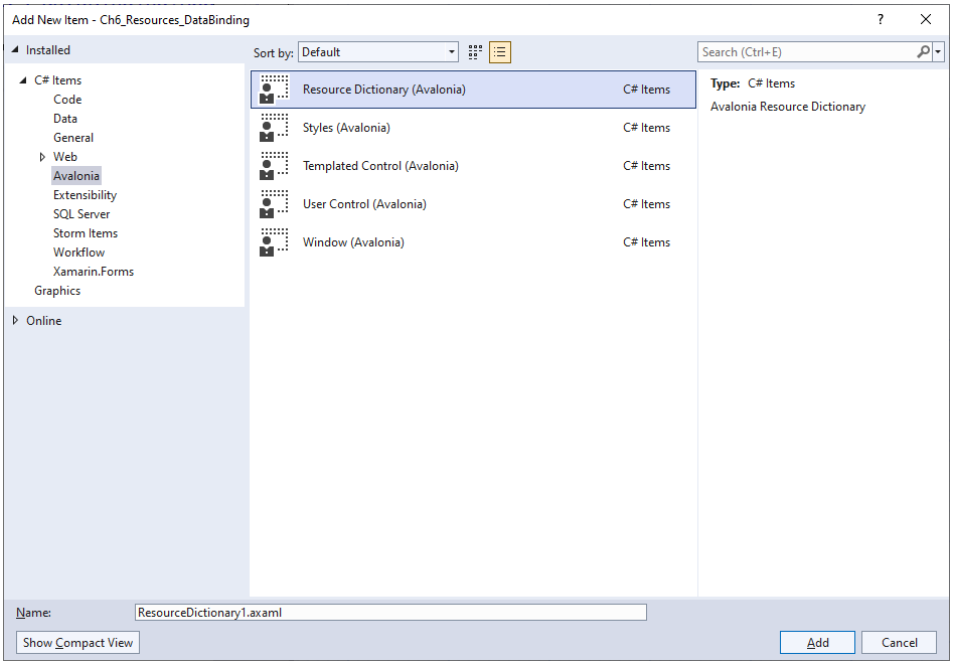



*Figure 31: Creating a resource dictionary*

Provide a name and click **Add**. The XAML for the new resource dictionary looks like the following:

```
<ResourceDictionary xmlns="https://github.com/avaloniaui"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <!-- Add Resources Here -->
</ResourceDictionary>
```

You will add resources inside the dictionary, replacing the comment; however, resource dictionaries cannot be consumed directly so they must be referenced first. To accomplish this, you use a *merged dictionary* as follows:

```
<Window.Resources>
      <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                  <ResourceInclude Source="/ResourceDictionary1.axaml"/>
            </ResourceDictionary.MergedDictionaries>
      </ResourceDictionary>
</Window.Resources>
```

You point to a resource dictionary specifying one or more **`ResourceInclude`** objects as a child of the **`ResourceDictionary.MergedDictionaries`** property. You assign the relative path of the dictionary to the **`Source`** property. In this way, resources stored in the dictionary can be consumed from within the visual element where they are referred from (or at the application level, if you add the merged dictionary to the App.axaml file).

Now that you know where resources are declared and their scope, it is time to see how resources work, starting with styles. Other resources, such as data templates, will be discussed later in this chapter.

## Introducing styles

When designing your user interface, in some situations, you might have multiple controls of the same type and, for each of them, you might need to assign the same properties with the same values. For example, you might have two buttons with the same width and height, or two or more labels with the same width, height, and font settings. In such situations, instead of assigning the same properties many times, you can take advantage of styles. A style allows you to assign a set of properties to views of the same type.

***Note: Styles in Avalonia UI work differently from WPF, where styles are resources that target a specific control as a whole. In Avalonia UI, styles are a separate feature with more flexibility, with an approach that recalls the Cascading Style Sheet (CSS) implementation. As you will see shortly, in Avalonia UI it is possible to reference specific parts of a control with a syntax that recalls HTML.***

### Implementing styles

In Avalonia UI, styles are added to the **`Styles`** element of a visual element and have the same scope as regular resources. They work like in the following example:

```
<Window.Styles>
```

```
      <Style Selector="TextBlock.textblockwelcome">
            <Setter Property="Foreground" Value="Green" />
            <Setter Property="FontSize" Value="24" />
      </Style>
</Window.Styles>
<TextBlock Classes="textblockwelcome" Text="Hi from Avalonia UI!"    />
```

The first thing you notice is the **Classes** property of the target control, in this case, a **TextBlock**. Like for HTML, this identifies the target of a style with a custom identifier. The **Style** element targets the specified element via its **Selector** property, whose value is the name of the control followed by a dot and the value of the **Classes** property. Property values are assigned with **Setter** elements, whose **Property** property represents the target property name, and whose **Value** represents the property value.

Another difference with WPF is that you do not need to explicitly reference the style, since this is applied by the selector. Selectors also allow you to target specific behaviors of a control, also referred to as *pseudo-classes*. The following declaration applies the style only when the mouse pointer hovers over the **TextBlock**:

```
      <Style Selector="TextBlock.textblockwelcome:pointerover">
            <Setter Property="Foreground" Value="Green" />
            <Setter Property="FontSize" Value="24" />
      </Style>
```

Another common target is **focus**, which applies the style only when the target gets focus. The full list of selectors and pseudo-classes is available in the documentation.

## Organizing styles

Like it is for regular resources, styles can be organized inside dedicated files. Avalonia UI provides the **Styles (Avalonia)** item template that you can see in Figure 31. With a separate file, you could rewrite the previous style as follows:

```
<Styles xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
      <Style Selector="TextBlock.textblockwelcome:pointerover">
            <Setter Property="Foreground" Value="Green" />
            <Setter Property="FontSize" Value="24" />
      </Style>
</Styles>
```

You could then reference the styles files in a window (or other container) with the following syntax:

```
<Window.Styles>
    <StyleInclude Source="/Styles1.axaml" />
</Window.Styles>
```

The **Source** is represented by the relative path of the .axaml file on disk.

### Using built-in themes

Themes are a special type of style that refer to the visual appearance of controls based on a specific style set. If you look at the App.axaml file, you will find the following definition:

```
<Application xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             x:Class="Ch6_Resources_DataBinding.App">
    <Application.Styles>
        <FluentTheme Mode="Light"/>
    </Application.Styles>
</Application>
```

The **`FluentTheme`** built-in style applies a light theme at the application level. You can change it to **`Dark`** and see a result similar to Figure 32, depending on what controls you add to the UI.

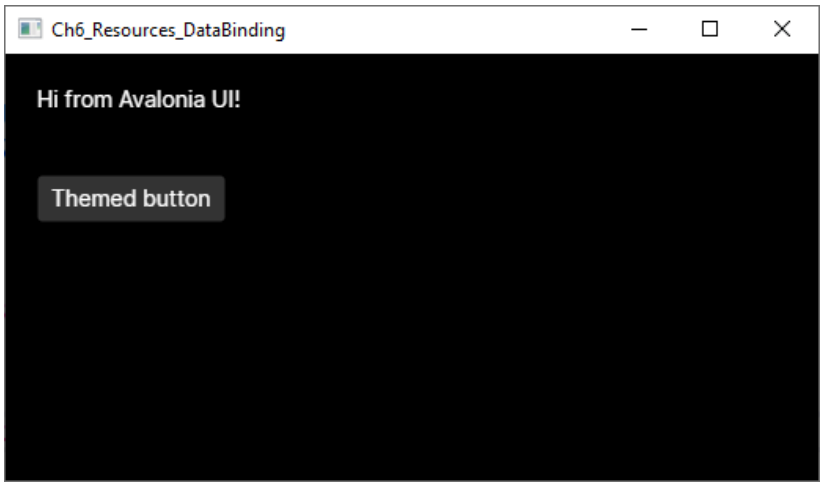



*Figure 32: Applying a dark theme*

It is also possible to define custom themes, but this is not covered here. You can refer to the official documentation page for more information on custom themes.

## Working with data binding

Data binding is a built-in mechanism that allows visual elements to communicate with data so that the user interface is automatically updated when data changes, and vice versa. Data binding is available in all the most important development platforms, and Avalonia UI is no exception. In fact, its data-binding engine relies on the power of XAML, and the way it works is similar in all the XAML-based platforms.

Avalonia UI supports binding objects, collections, and visual elements to other visual elements. This chapter describes the first two scenarios. Because data binding is a very complex topic, the best way to start is with an example. Suppose you want to bind an instance of the following **Person** class to the user interface, so that a communication flow is established between the object and views:

```
public class Person
{
    public string FullName { get; set; }
    public DateTime DateOfBirth { get; set; }
    public string Address { get; set; }
}
```

In the user interface, you will want to allow the user to enter their full name, date of birth, and address via a **TextBox**, a **DatePicker**, and another **TextBox**, respectively. In XAML, this can be accomplished with the code shown in Code Listing 10.

*Code Listing 10*

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
        x:Class="Ch6_Resources_DataBinding.SimpleBindingWindow"
        Title="SimpleBindingWindow">
       <StackPanel>
              <TextBlock Text="Name:" />
              <TextBox Text="{Binding FullName}"/>
              <TextBlock Text="Date of birth:"/>
              <DatePicker
               SelectedDate="{Binding DateOfBirth, Mode=TwoWay}"/>
              <TextBlock Text="Address:"/>
              <TextBox Text="{Binding Address}"/>

       </StackPanel>
</Window>
```

As you can see, the **Text** property for **TextBox** controls and the **SelectedDate** property of the **DatePicker** have a markup expression as their value. Such an expression is made up of the **Binding** literal followed by the property you want to bind from the data object. Actually, the expanded form of this syntax could be **{Binding Path=PropertyName}**, but **Path** can be omitted. Data binding can be of five types:

- **TwoWay:** Views can read and write data.
- **OneWay:** Views can only read data.
- **OneWayToSource:** Views can only write data.
- **OneTime:** Views can read data only once.
- **Default**: Avalonia UI resolves the appropriate mode automatically, based on the view.

**TwoWay** and **OneWay** are the most-used modes, and in most cases, you do not need to specify the mode explicitly because Avalonia UI automatically resolves the appropriate mode based on the control. For example, binding in the **TextBox** control is **TwoWay** because this kind of view can be used to read and write data, whereas binding in the **TextBlock** control is **OneWay** because this view can only read data.

Controls properties that are bound to an object's properties are known as *styled properties* (or dependency properties, if you come from the WPF or UWP world). More details about styled properties will come shortly; for now, what you need to know is that styled properties will automatically update the value of the bound object's property and automatically refresh their value in the user interface if the object is updated. However, this automatic refresh is possible only if the data-bound object implements the **INotifyPropertyChanged** interface, which allows an object to send change notifications. As a consequence, you must extend the **Person** class definition, as shown in Code Listing 11.

*Code Listing 11*

```
public class Person : INotifyPropertyChanged
{
    private string fullName;
    public string FullName
    {
        get
        {
            return fullName;
        }
        set
        {
            fullName = value;
            OnPropertyChanged();
        }
    }

    private DateTime dateOfBirth;

    public DateTime DateOfBirth
    {
        get
        {
            return dateOfBirth;
        }
        set
        {
            dateOfBirth = value;
            OnPropertyChanged();
        }
    }
    private string address;
    public string Address
```

```
        {
            get
            {
                return address;
            }
            set
            {
                address = value;
                OnPropertyChanged();
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;

        private void OnPropertyChanged([CallerMemberName]
                string propertyName = null)
        {
            PropertyChanged?.Invoke(this,
                new PropertyChangedEventArgs(propertyName));
        }
    }
```

By implementing **INotifyPropertyChanged**, property setters can raise a change notification via the **PropertyChanged** event. Bound views will be notified of any changes and will refresh their contents.

***Tip: With the `CallerMemberName` attribute, the compiler automatically resolves the name of the caller member. This avoids the need to pass the property name in each setter and helps keep code much cleaner.***

The next step is binding an instance of the **Person** class to the user interface. This can be accomplished with the following lines of code, normally placed inside the window's code-behind file:

```
Person person = new Person();
this.DataContext = person;
```

Windows and panels expose the **DataContext** property, of type **object**, which represents the data source for the window or panel. Child controls that are data bound to an object's properties will search for an instance of the object in the **DataContext** property value and bind to properties from this instance. In this case, the **TextBox** and the **DatePicker** will search for an object instance inside **DataContext** and bind to properties from that instance. Remember that XAML is case-sensitive, so binding to **FullName** is different from binding to **Fullname**. The runtime will throw an exception if you try to bind to a property that does not exist or has a different name.

If you now try to run the application, not only will data binding work, but the user interface will also be automatically updated if the data source changes. You may think of binding views to a single object instance, like in the previous example, as binding to a row in a database table.

## Introducing styled properties

Styled properties are the equivalent of WPF's dependency properties and, put succinctly, they represent the target of a data-binding expression. Styled properties are only exposed by visual elements, and their value is automatically updated as the source value changes. As a consequence, the user interface is automatically updated as the styled property value changes.

In Avalonia UI, you will implement custom styled properties only when defining your own control (as demonstrated in Chapter 7, "Creating Custom Controls"). For now, it is important to understand their behavior. If you look at Code Listing 10, the **`Text`** property of the **`TextBox`** and the **`SelectedDate`** property of the **`DatePicker`** are both styled properties. They are the target of the data-binding expression, and the corresponding visual element is updated as their value changes.

For a better understanding, consider the following code that shows the implementation for the **`Button.IsPressed`** styled property:

```
public static readonly StyledProperty<bool> IsPressedProperty =
    AvaloniaProperty.Register<Button, bool>(nameof(IsPressed));

public bool IsPressed
{
    get => GetValue(IsPressedProperty);
    private set => SetValue(IsPressedProperty, value);
}
```

The first step is declaring a static read-only field of type **`StyledProperty<T>`**, where **`T`** is the data type for the property. The name of the field ends, by convention, with the **`Property`** literal. Its value is an instance of the **`StyledProperty`** object created by the **`AvaloniaProperty.Register`** method (**`AvaloniaProperty`** is the base class for styled properties). **`Register`** requires two type parameters: the target control and the property type. The method argument is instead a string representing the name of a regular CLR property defined in the following example. The **`IsPressed`** property is a normal property that returns the value of the styled property via the **`GetValue`** method and stores the value via the **`SetValue`** method. Some styled properties, like the **`TextBox.Text`**, also define the default binding direction.

***Note: Compared to WPF, Avalonia UI provides a richer property system made of the styled properties and the so-called direct properties. For the purposes of this***

***book, knowing the styled properties is enough. For more information about the direct properties, you can refer to the documentation.***

## Working with collections

Though working with a single object instance is a common scenario, another very common situation is working with collections that you display as lists in the user interface. Avalonia UI supports data-binding over collections via the **`ObservableCollection<T>`** object. This collection works exactly like the **`List<T>`** but also raises a change notification when items are added to or removed from the collection. Collections are very useful, for example, when you want to represent rows in a database table.

For example, suppose you have the following collection of **`Person`** objects:

```
Person person1 = new Person { FullName = "Alessandro",
    DateOfBirth = new DateTime(1977,5,10) };
Person person2 = new Person { FullName = "James",
    DateOfBirth = new DateTime(1980, 1, 1) };
Person person3 = new Person { FullName = "Graham",
    DateOfBirth = new DateTime(1982, 12, 31) };
var people = new ObservableCollection<Person>() { person1, person2,
 person3 };
this.DataContext = people;
```

The code assigns the collection to the **`DataContext`** property of the root container (and later you will work with data and logic separation when talking about MVVM). At this point, you need a visual element capable of displaying the content of this collection. This is where the **`ListBox`** control comes in. The **`ListBox`** can receive the data source from either the **`DataContext`** of its container or by assigning its **`Items`** property, and any object that implements the **`IEnumerable`** interface can be used with the **`ListBox`**. You will typically assign **`Items`** directly if the data source for the **`ListBox`** is not the same data source as for the other views in the page.

***Tip: All the controls that allow for displaying data and that expose the `Items` property are also referred to as Item Controls.***

The problem to solve with the **`ListBox`** is that it does not know how to present objects in a list. For example, think of the **`People`** collection that contains instances of the **`Person`** class. Each instance exposes the **`FullName`**, **`DateOfBirth`**, and **`Address`** properties, but the **`ListBox`** does not know how to present these properties, so it is your job to explain to it how. This is accomplished with the *data templates*. A data template is a static set of views that are bound to properties in the object. It instructs the **`ListBox`** on how to present items.

For example, if you only had to display and edit the **`FullName`** property, you could write the following data template:

```
<Grid>
   <ListBox x:Name="PeopleList" Items="{Binding}">
         <ListBox.ItemTemplate>
               <DataTemplate>
                     <StackPanel Orientation="Horizontal">
                           <TextBlock Text="Full name: "/>
                           <TextBlock Text="{Binding FullName}"/>
                     </StackPanel>
               </DataTemplate>
         </ListBox.ItemTemplate>
   </ListBox>
</Grid>
```

***Tip: The `DataTemplate` definition is always defined inside the `ListBox.ItemTemplate` element.***

As a general rule, if the data source is assigned to the **`DataContext`** property, the **`Items`** must be set with the **`{Binding}`** value. This means your data source is the same as that of your parent. With this code, the **`ListBox`** will display all the items in the bound collection, showing two **`TextBlock`** controls for each item. However, each **`Person`** also exposes a property of type **`DateTime`**, so you need to extend the **`DataTemplate`**, as shown in Code Listing12.

*Code Listing 12*

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
        x:Class="Ch6_Resources_DataBinding.ListBoxWindow"
        Title="ListBoxWindow">
        <Grid>
               <ListBox x:Name="PeopleList" Items="{Binding}">
                      <ListBox.ItemTemplate>
                             <DataTemplate>
                                    <StackPanel>
                             <StackPanel Orientation="Horizontal">

        <TextBlock Text="Full name: "/>

        <TextBlock Text="{Binding FullName}"/>
                                              </StackPanel>

        <StackPanel Orientation="Horizontal" Margin="0,5,0,0">

        <TextBlock Text="Date of birth: "/>

        <CalendarDatePicker SelectedDate="{Binding DateOfBirth}"/>
```

```
                                        </StackPanel>
                                    </StackPanel>
                                </DataTemplate>
                            </ListBox.ItemTemplate>
                        </ListBox>
                </Grid>
</Window>
```

As you can see, data templates allow you to implement complex visual elements and display whatever kind of information you need.

Figure 33 shows the result for the code described in this section. Notice that the **`ListBox`** includes built-in scrolling capability and must never be enclosed within a **`ScrollViewer`**.

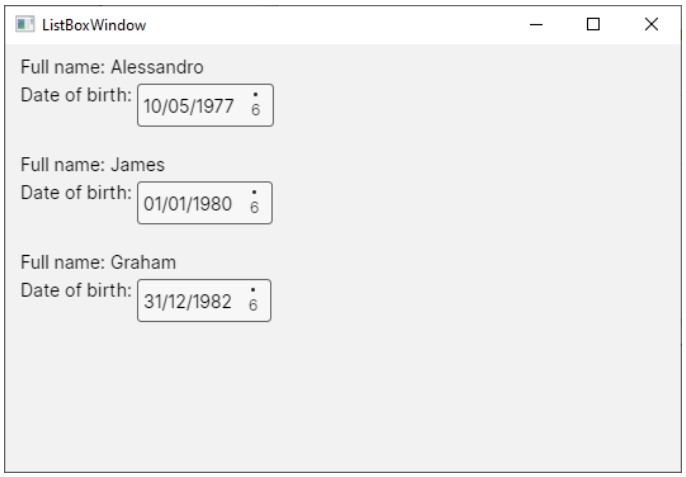



*Figure 33: A data-bound ListBox*

A data template can be placed inside the window or app resources so that it becomes reusable. Then, you assign the **`ItemTemplate`** property in the **`ListBox`** definition with the **`StaticResource`** expression, as shown in Code Listing 20.

*Code Listing 20*

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
        x:Class="Ch6_Resources_DataBinding.ListBoxWindow"
        Title="ListBoxWindow">
       <Window.Resources>
              <DataTemplate x:Key="PersonDataTemplate">
                     <StackPanel>
                            <StackPanel Orientation="Horizontal">
```

```
                                    <TextBlock Text="Full name: "/>

        <TextBlock Text="{Binding FullName}"/>
                                </StackPanel>

        <StackPanel Orientation="Horizontal" Margin="0,5,0,0">
                                    <TextBlock Text="Date of birth: "/>

        <CalendarDatePicker SelectedDate="{Binding DateOfBirth, Mode=TwoWay
}"/>
                                </StackPanel>
                        </StackPanel>
                </DataTemplate>
        </Window.Resources>
        <Grid>
                <ListBox x:Name="PeopleList" Items="{Binding}"

         ItemTemplate="{StaticResource PersonDataTemplate}"/>
        </Grid>
</Window>
```

Separating data templates from the data controls is very common; it allows for reusability and helps keep the code cleaner. The **ListBox** also provides the following important properties:

- **SelectedItem:** Represents the currently selected item in the control.
- **SelectionMode:** Specifies how items can be selected according to the following values: **Single**, **Multiple**, **Toggle**, **AlwaysSelected**.

In order to handle selection, the **ListBox** must subscribe to the **SelectionChanged** event that can be handled as follows:

```
private void PeopleList_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    // Get the current item
    var currentItem = PeopleList.SelectedItem as Person;

    // Get the first selected person from the collection
    var selectedPerson = e.AddedItems[0] as Person;
}
```

**SelectedItem** is of type **object** and must be converted into an instance of the bound type. The **AddedItems** property of the **SelectionChangedEventArgs** object returns a collection of currently selected items (when **SelectionMode** is **Multiple**), whereas **RemovedItems** returns a collection of items that were selected before the current selection.

## Binding lists to a ComboBox

In the previous chapter, you saw how to implement a **ComboBox** to display lists of options supplied in code, but actually, the **ComboBox** is an items control and exposes the **Items** property. For example, you could bind the collection of **Person** objects shown previously as follows:

```
<ComboBox Items="{Binding}">
      <ComboBox.ItemTemplate>
            <DataTemplate>
                  <StackPanel>
                        <TextBlock Text="{Binding FullName}"/>
                  </StackPanel>
            </DataTemplate>
      </ComboBox.ItemTemplate>
</ComboBox>
```

Figure 34 shows how the data-bound **ComboBox** appears.


*Figure 34: A data-bound ComboBox*

Like the **ListBox**, the **ComboBox** also exposes the **SelectedItem** property and the **SelectionChanged** event that you handle in the same ways you previously saw.

## Introducing value converters

In Chapter 3, you learned about type converters, which automatically convert a type into another one that is supported by a property binding. For example, if you bind an integer value to the **Text** property of a **TextBox** control, such an integer is converted into a **string** by a XAML type converter. However, there are situations in which you might want to bind objects that XAML type converters cannot automatically convert into the type a control expects.

For example, you might want to bind a **Color** value to a **Label**'s **Text** property, which is not possible out of the box. In these cases, you can create *value converters*. A value converter is a class that implements the **IValueConverter** interface and must expose the **Convert** and **ConvertBack** methods. **Convert** translates the original type into a type that the view can receive, while **ConvertBack** does the opposite.

For a better understanding, suppose you want to show or hide a **TextBlock** depending on the **Yes** or **No** choice made via the following **ComboBox**:

```
<ComboBox Name="ComboBox1">
      <ComboBoxItem>Yes</ComboBoxItem>
      <ComboBoxItem>No</ComboBoxItem>
</ComboBox>
```

The choice made here is stored by the **Content** property of each **ComboBoxItem**. This needs to be converted into a **bool** that can be assigned to the **IsVisible** property of a **TextBlock**. To accomplish this, you could implement the converter shown in Code Listing 21.

*Code Listing 21*

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

        var item = value as ComboBoxItem;
        string actualValue = item.Content.ToString();
        switch(actualValue)
        {
            case "Yes":
                return true;
            default:
                return false;
        }
    }

    public object? ConvertBack(object? value, Type targetType,
           object? parameter, CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

Both methods always receive the data to convert as object instances, and then you need to cast the object into a specialized type for manipulation. In this case, **Convert** receives a **ComboBoxItem** and converts the value of its **Content** property into a **string**. Then, depending on the content (**Yes** or **No**), it returns the appropriate **bool** value. The value converter must then be declared in the resources of the XAML file where you wish to use it (or in App.axaml). To do so, you first declare an XML namespace that points to the assembly that defines the converter:

```
xmlns:local="using:Ch6_Resources_DataBinding"
```

Then, you add the converter to the resources:

```
<Window.Resources>
      <local:YesNoToBoolConverter x:Key="BoolConverter"/>
</Window.Resources>
```

As with any other resource, you assign an identifier via the **`x:Key`** attribute. Finally, you consume the converter in the following data-binding expression:

```
<TextBlock Text="Visibility test"
           IsVisible="{Binding ElementName=ComboBox1, Path=SelectedItem,
           Converter={StaticResource BoolConverter}}"/>
```

As a general rule, you assign a converter via the **`Converter`** expression pointing to the converter identifier via **`StaticResource`**. In this particular case, the **`TextBlock`** needs to read the value of the currently selected item in the **`ComboBox`**, so it is bound to the **`ComboBox`** via **`ElementName`** and the **`Path`** expression points to the **`SelectedItem`** property. If you ran this code, you would see that the **`TextBlock`** visibility is determined by what you choose in the **`ComboBox`**.

## Displaying lists with the DataGrid

Previously, you have seen how to display lists of data with the **`ListBox`**, which provides great flexibility when it comes to creating complex data templates. However, sometimes you might need to display data in a tabular form. In this case, you can use a convenient control called **`DataGrid`**. This comes with a separate NuGet package called Avalonia.Controls.DataGrid. Figure 35 shows how it appears in the NuGet Package Manager.

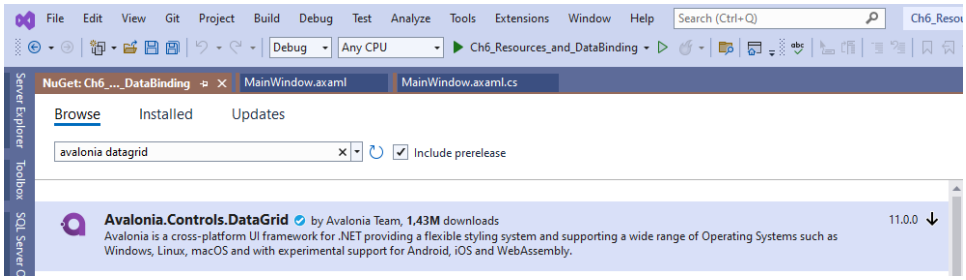



*Figure 35: Installing the DataGrid package*

Make sure that the package version matches the Avalonia UI version you selected during the project creation. Once installed, you need to add the following line to the **`Styles`** node of the App.axaml file:

```
<StyleInclude
Source="avares://Avalonia.Controls.DataGrid/Themes/Fluent.xaml"/>
```

Now, suppose you want to display the previously discussed **`ObservableCollection<Person>`** as tabular data. The easiest way to accomplish this, after assigning the collection to the window's **`DataContext`** (as you have already done several times), is by declaring a **`DataGrid`** as follows:

```
<DataGrid Name="DataGrid1" ClipboardCopyMode="IncludeHeader"
```

```
        Items="{Binding}" AutoGenerateColumns="True"
        GridLinesVisibility="All"/>
```

If you assign **True** to the **AutoGenerateColumns** property, the control will generate the columns for you. With the **ClipboardCopyMode**, you can specify if copying to the clipboard will include the header (**IncludeHeader**) or not (**ExcludeHeader**). You can also control the grid line visibility, such as all visible, only vertical, or only horizontal.

If you run the code, you will notice that the **DataGrid** generates editable text columns for all your properties by basically invoking a **.ToString()** on each. However, if you have specific data types like the **DateTime** of the sample code, a better approach is to implement specialized columns. What you need to do is assign **AutoGenerateColumns** with **False** and leverage the following objects:

- **DataGridTextColumn**: This allows for displaying and editing text values.
- **DataGridCheckBoxColumn**: This allows for displaying and editing bool values via a **CheckBox** control.
- **DataGridTemplateColumn**: This allows for the implementation of custom visualization for any other data type.

For example, if you look at Code Listing 22, you can see how to show text, but also how to provide a date selection for the sample data model you created previously.

*Code Listing 22*

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
        x:Class="Ch6_Resources_and_DataBinding.DataGridWindow"
        Title="DataGridWindow">
    <Grid>
        <DataGrid Name="DataGrid1" ClipboardCopyMode="IncludeHeader"
                  Items="{Binding}" AutoGenerateColumns="False"
                  AlternatingRowBackground="LightGreen"

          CanUserResizeColumns="True" CanUserReorderColumns="True"
                 GridLinesVisibility="All">
           <DataGrid.Columns>
             <DataGridTextColumn Header="First Name"
              Binding="{Binding FullName}"/>
             <DataGridTemplateColumn Header="Date of birth">
                <DataGridTemplateColumn.CellTemplate>
                   <DataTemplate>
                      <CalendarDatePicker
                       SelectedDate="{Binding DateOfBirth}" />
                   </DataTemplate>
```

```
                </DataGridTemplateColumn.CellTemplate>
            </DataGridTemplateColumn>
        </DataGrid.Columns>
    </DataGrid>
</Grid>
</Window>
```

All the columns share the **Header** property, which contains the text for the column header.

> ***Tip: The*** *HeaderTemplate* ***property can be used instead of*** *Header* ***if you want to use a custom visual element as the column header. The definition is added inside a*** *DataTemplate* ***element.***

The **DataGridTextColumn** and **DataGridCheckBoxColumn** objects also expose the **Binding** property, which is used to bind the backing data. For custom columns, in the **DataGridTemplateColumn** object, you need to populate the **CellTemplate** property with a **DataTemplate** that contains the visual elements you want to implement. In the current example, a **CalendarDatePicker** is used, and its **SelectedDate** property is bound to the **DateOfBirth** property of the **Person** class. The result of Code Listing 22 is shown in Figure 36.

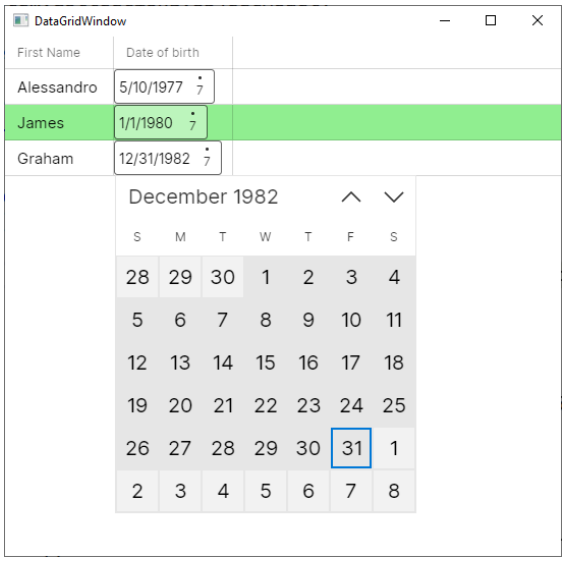



*Figure 36: Generating custom columns with the DataGrid*

Notice how the **CanUserResizeColumns** and **CanUserReorderColumns** properties can be used to improve the user interaction. You could also use the **IsReadOnly** property to implement read-only columns. Like for the **ListBox**, you can set the **SelectionMode** property to determine how item selection should work, and you can handle the **SelectionChanged** event in the same way.

In addition, you can control what happens inside cells by subscribing some events exposed by the **DataGrid**. For example, you can handle the **CellEditEnding** and **CellEditEnded** events, whose names are self-explanatory, as follows:

```
private void DataGrid1_CellEditEnding(object sender,
        DataGridCellEditEndingEventArgs e)
{
    if (e.EditAction == DataGridEditAction.Commit)
    {
        // Further data validation here...
    }
}

private void DataGrid1_CellEditEnded(object sender,
        DataGridCellEditEndedEventArgs e)
{
    if(e.EditAction == DataGridEditAction.Commit)
    {
        // Some post-save actions here...
    }
}
```

Both the **DataGridCellEditEndingEventArgs** and the **DataGridCellEditEndedEventArgs** objects expose the **EditAction** property, whose value can be **Commit** or **Cancel**, so you can understand what action was requested by the user. In the case of **CellEditEnding**, which happens before finalizing the edit action, you can also programmatically cancel the operation by setting **e.Cancel = true;**. The full list of events can be found in the official documentation. For now, you have everything you need to start working with the **DataGrid**.

## Scrolling lists with the Carousel

The **Carousel** is a control that allows for scrolling lists. Typically, a **Carousel** is used to scroll lists horizontally, but it also supports vertical orientation. This control can also be populated manually, like the **ComboBox**, so it is not limited to data-binding scenarios, but the dynamic assignment of items is certainly more common.

Unlike what you would expect from similar controls offered by other platforms, the scrolling actions are not automatic, so you need to implement your own click handling. The code shown in Code Listing 23 provides an example of implementing a **Carousel** to scroll the previously defined list of people horizontally.

*Code Listing 23*

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
        x:Class="Ch6_Resources_and_DataBinding.CarouselWindow"
        Title="CarouselWindow">
       <Grid>
```

```
            <Grid.ColumnDefinitions>
                <ColumnDefinition  Width="Auto"/>
                <ColumnDefinition />
                <ColumnDefinition Width="Auto"/>
            </Grid.ColumnDefinitions>
            <Carousel Name="Carousel1" Items="{Binding}" Grid.Column="1"
             HorizontalAlignment="Center" VerticalAlignment="Center">
                <Carousel.PageTransition>
                   <CompositePageTransition>
                        <PageSlide Duration="0:00:01.000"
                         Orientation="Horizontal" />
                   </CompositePageTransition>
                </Carousel.PageTransition>
                <Carousel.ItemTemplate>
                   <DataTemplate>
                        <StackPanel>
                           <TextBlock Text="{Binding FullName}"
                            FontSize="18" FontWeight="Bold" />
                           <TextBlock Text="{Binding DateOfBirth}"
                            Foreground="Gray" />
                           <TextBlock Text="{Binding Address}"
                            FontSize="14" />
                        </StackPanel>
                     </DataTemplate>
                </Carousel.ItemTemplate>
            </Carousel>
            <Button Margin="10" Name="PreviousButton"
             Click="PreviousButton_Click">&lt;</Button>
            <Button Margin="10" Name="NextButton"
             Click="NextButton_Click" Grid.Column="2">&gt;</Button>
        </Grid>
</Window>
```

First, you specify the page transition via the **CompositePageTransition** object, assigned with a **PageSlide** object that represents the duration and direction of the transition. Then, you specify an **ItemTemplate**, as you have done previously with the **ListBox** and **DataGrid**. Notice how the two **Button** controls are implemented to provide sliding actions. Their event handlers look like the following:

```
private void PreviousButton_Click(object sender, RoutedEventArgs e)
{
    Carousel1.Previous();
}

private void NextButton_Click(object sender, RoutedEventArgs e)
{
    Carousel1.Next();
}
```

The **Previous** and **Next** methods allow for sliding the **Carousel** content. Figure 37 shows the result.

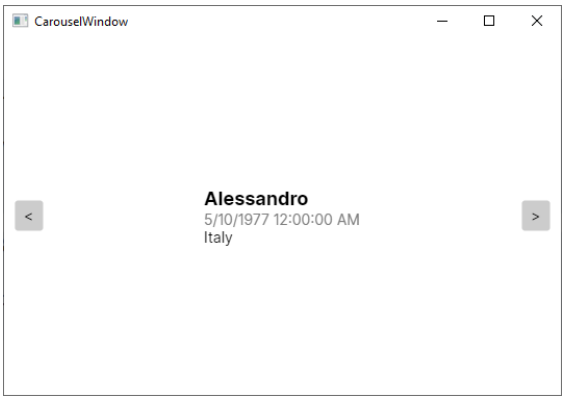



*Figure 37: Sliding items with a Carousel*

The current example is totally based on strings, but you can obviously add images and complex visual elements that are more suitable to be displayed with a **Carousel**.

## Items repeater controls

The Avalonia UI codebase provides two controls called **ItemsControl** and **ItemsRepeater**. Their purpose is to display lists of items in a stack. The **ItemsControl** has no template and no interaction, whereas the **ItemsRepeater** does. For this reason, an example will be provided for the **ItemsRepeater**. For more on the **ItemsControl**, you can refer to the official documentation.

Suppose you want to display a list of **Person** objects as defined previously. You could implement an **ItemsRepeater**, as shown in Code Listing 24.

*Code Listing 24*

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
        x:Class="Ch6_Resources_and_DataBinding.ItemsRepeaterWindow"
        Title="ItemsRepeaterWindow">
       <Grid>
          <ItemsRepeater Items="{Binding}">
             <ItemsRepeater.ItemTemplate>
                <DataTemplate>
                  <Border BorderBrush="Blue" Margin="5"
                          CornerRadius="10" BorderThickness="2">
                      <StackPanel Margin="5">
                          <TextBlock Text="{Binding FullName}"/>
```

```
                        <CalendarDatePicker
                         SelectedDate="{Binding DateOfBirth,
                                        Mode=TwoWay}"/>
                    </StackPanel>
                </Border>
            </DataTemplate>
        </ItemsRepeater.ItemTemplate>
    </ItemsRepeater>
</Grid>
</Window>
```

In the code-behind file, assign the data source as you have done for the previous controls. As the name implies, the **`Border`** draws a border around its child element by specifying the brush (**`BorderBrush`**), the thickness (**`BorderThickness`**), and the (optional) rounding for corners (**`CornerRadius`**). As you can see in Figure 38, the **`ItemsRepeater`** renders items in a vertical stack.

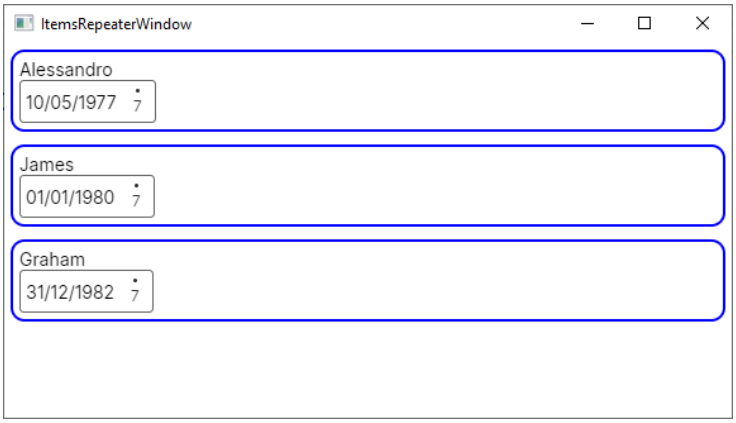



*Figure 38: Displaying items with the ItemsRepeater*

If you want to implement horizontal rendering, you need to assign the **`Layout`** property as follows:

```
<ItemsRepeater.Layout>
  <StackLayout Spacing="40"
      Orientation="Horizontal" />
</ItemsRepeater.Layout>
```

Here you also meet the **`StackLayout`** for the first time. This visual element renders items horizontally, but only as the child element of the **`Layout`** property from item controls.

***Note: The ItemsRepeater control is a porting from the UWP counterpart and has been discussed here for the sake of consistency. However, you might prefer a ListBox, which certainly provides more flexibility.***

## Introducing the Model-View-ViewModel pattern

Model-View-ViewModel (MVVM) is an architectural pattern used in XAML-based platforms that allows for clean separation between the data (model), the logic (ViewModel), and the user interface (view). With MVVM, windows and user controls only contain code related to the user interface. They strongly rely on data binding, and most of the work is done in the ViewModel. MVVM can be quite complex if you have never seen it before, so I will try to simplify the explanations as much as possible, but you should use Avalonia's MVVM documentation as a reference.

Let's start with a simple example and a fresh Avalonia UI solution. Imagine you want to work with a list of **`Person`** objects. This is your model, and you can reuse the **`Person`** class from Code Listing 11. Add a new folder called **Model** to your project and add a new **Person.cs** class file to this folder, pasting in the code of the **`Person`** class. Next, add a new folder called **ViewModel** to the project and a new class file called **PersonViewModel.cs**.

Before writing the code for it, let's summarize some important considerations:

- The ViewModel contains the business logic, acts like a bridge between the model and the view, and exposes properties to which the view can bind.
- Among such properties, one will certainly be a collection of **`Person`** objects.
- In the ViewModel, you can load data, filter data, execute save operations, and query data.

Loading, filtering, saving, and querying data are examples of actions a ViewModel can execute against data. In a classic development approach, you would handle **`Click`** events on **`Button`** controls and write the code that executes an action. However, in MVVM, windows and controls should only work with code related to the user interface, not code that executes actions against data. In MVVM, ViewModels expose public methods that will be bound to the **`Command`** property of a control, such as **`Button`** or **`MenuItem`**, and that will execute the action. In the UI, you bind a view to a command in the ViewModel. This way, the action is executed in the ViewModel instead of in the view's code behind. Code Listing 25 shows the **`PersonViewModel`** class definition.

*Code Listing 25*

```
public class PersonViewModel : INotifyPropertyChanged
{
    public ObservableCollection<Person> People { get; set; }

    private Person _selectedPerson;
    public Person SelectedPerson
    {
```

```
        get
        {
            return _selectedPerson;
        }
        set
        {
            _selectedPerson = value;
            OnPropertyChanged();
        }
    }

    public void AddPerson()
    {
        People.Add(new Person());
    }

    public bool CanAddPerson()
    {
        return true;
    }


    public event PropertyChangedEventHandler PropertyChanged;
    private void OnPropertyChanged([CallerMemberName] string propertyName
                                   = null)
    {
        PropertyChanged?.Invoke(this,
            new PropertyChangedEventArgs(propertyName));
    }
    private void LoadSampleData()
    {
        People = new ObservableCollection<Person>();

        // sample data
        Person person1 =
            new Person
            {
                FullName = "Alessandro",
                Address = "Italy",
                DateOfBirth = new DateTime(1977, 5, 10)
            };
        Person person2 =
            new Person
            {
                FullName = "Robert",
                Address = "United States",
                DateOfBirth = new DateTime(1960, 2, 1)
            };
        Person person3 =
```

```
            new Person
            {
                FullName = "Niklas",
                Address = "Germany",
                DateOfBirth = new DateTime(1980, 4, 2)
            };

        People.Add(person1);
        People.Add(person2);
        People.Add(person3);
    }

    public PersonViewModel()
    {
        LoadSampleData();
    }

}
```

The **People** and **SelectedPerson** properties expose a collection of **Person** objects and a single **Person**, respectively, and the latter will be bound to the **SelectedItem** property of a **DataGrid**, as you will see shortly. Notice how:

- The **AddPerson** method represents the action that makes it possible to add a new object to the collection and that will be bound to the **Command** property of a **Button**.
- The **CanAddPerson** method returns a **bool** value that the data-binding engine checks for, in order to understand if it is possible to execute an action.

The **LoadSampleData** method generates some sample data that will be the source for a **DataGrid**. Now it is time to write the XAML code for the user interface. Code Listing 26 shows how to use a **DataGrid** and bind a **Button** to commands.

*Code Listing 26*

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
        x:Class="Ch6_MVVM.MainWindow"
        Title="Ch6_MVVM">
       <Grid>
         <Grid.RowDefinitions>
              <RowDefinition Height="Auto"/>
              <RowDefinition />
          </Grid.RowDefinitions>
          <StackPanel Orientation="Horizontal">
```

```
            <Button Name="AddButton" Command="{Binding AddPerson}"
             Content="Add person"/>
          </StackPanel>
          <DataGrid Grid.Row="1" Name="DataGrid1"
           ClipboardCopyMode="IncludeHeader"
           Items="{Binding People}" AutoGenerateColumns="False"
           SelectedItem="{Binding SelectedPerson, Mode=TwoWay}"
          AlternatingRowBackground="LightGreen"
          CanUserResizeColumns="True" CanUserReorderColumns="True"
          GridLinesVisibility="All">
              <DataGrid.Columns>
                  <DataGridTextColumn Header="First Name"
                    Binding="{Binding FullName}"/>
                 <DataGridTemplateColumn Header="Date of birth">
                   <DataGridTemplateColumn.CellTemplate>
                      <DataTemplate>
                        <CalendarDatePicker
                          SelectedDate="{Binding DateOfBirth}" />
                      </DataTemplate>
                   </DataGridTemplateColumn.CellTemplate>
                 </DataGridTemplateColumn>
                 <DataGridTextColumn Header="Address"
                       Binding="{Binding Address}"/>
              </DataGrid.Columns>
        </DataGrid>
    </Grid>
</Window>
```

Notice how:

- The **`DataGrid.Items`** property is bound to the **`People`** collection in the ViewModel.
- The **`DataGrid.SelectedItem`** property is bound to the **`SelectedPerson`** property in the ViewModel.
- The **`Button`** is bound to the **`AddPerson`** method in the ViewModel via its **`Command`** property.

The final step is to create an instance of the ViewModel and assign it to the **`DataContext`** of the page, which you can do in the page code-behind, as demonstrated in Code Listing 27.

*Code Listing 27*

```
namespace Ch6_MVVM
{
    public partial class MainWindow : Window
    {
        // Not using a field here because properties
        // are optimized for data-binding.
        private PersonViewModel ViewModel { get; set; }
```

```
        public MainWindow()
        {
            InitializeComponent();

            this.ViewModel = new PersonViewModel();
            this.DataContext = this.ViewModel;
        }
    }
}
```

If you run the application now (see Figure 39), you will see the list of **`Person`** objects and be able to use the button. However, the real benefit is that the whole logic is in the ViewModel. With this approach, if you change the logic in the properties or in the commands, you will not need to change the page code.

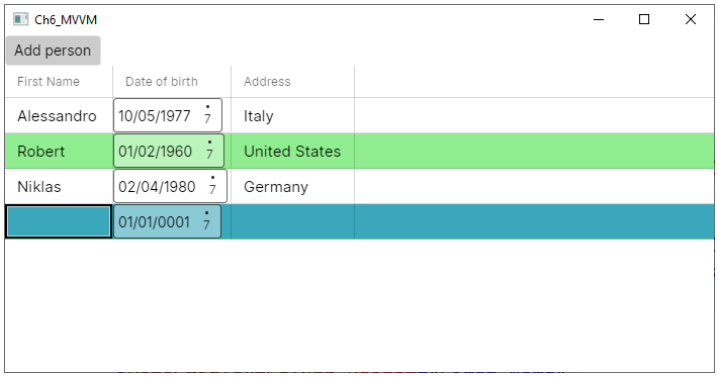



*Figure 39: Showing a list of people and adding a new person with MVVM*

MVVM is very powerful, but real-world implementations can be very complex. For example, if you want to navigate to another page and you have commands, the ViewModel should contain code related to the user interface (launching a window) that does not adhere to the principles of MVVM. You will get some more information about this in Chapter 8, “Working with Pages and Dialogs.”

# Chapter summary

XAML plays a fundamental role in Avalonia UI and allows for defining reusable resources and for data-binding scenarios. Resources are reusable styles, data templates, and references to objects you declare in XAML. In particular, styles allow you to set the visual appearance of controls with a CSS-based approach. XAML also includes a powerful data-binding engine that allows you to quickly bind objects to visual elements in a two-way communication flow.

In this chapter, you have seen how to bind a single object to individual visual elements and collections of objects to the **`ListBox`**, **`DataGrid`**, **`Carousel`**, and **`ItemsRepeater`**. You have seen how to define data templates so that item controls can have knowledge of how items must be presented, and you have learned about value converters, special objects that come in to help when you want to bind objects of a type that is different from the type a view supports.

In the second part of the chapter, you walked through an introduction to the Model-View-ViewModel pattern, focusing on separating the logic from the UI and understanding new objects and concepts such as command binding.

In the next chapter, you will learn how to build your own controls and how to redefine the user interface for existing controls.

# Chapter 7  Creating Custom Controls

Like any development platform, Avalonia UI also allows you to create reusable controls. Similarly to WPF, you can build user controls, which represent the aggregation of several visual elements into one control, and custom controls via templates. The latter allows for completely redefining the look and feel of an existing control, while keeping the existing behavior. Both types are discussed in this chapter.

## Creating user controls

A user control is a reusable component that aggregates primitive or complex elements. The following example shows how to create a user control that will allow users to choose a file and display the file name in a text box. When you have an Avalonia UI solution ready, you can select **Project** > **Add New Item**, and in the **Add New Item** window, select the template called **User Control (Avalonia)** (see Figure 40). Assign **FileBrowserUserControl.axaml** as the file name, then click **Add**.

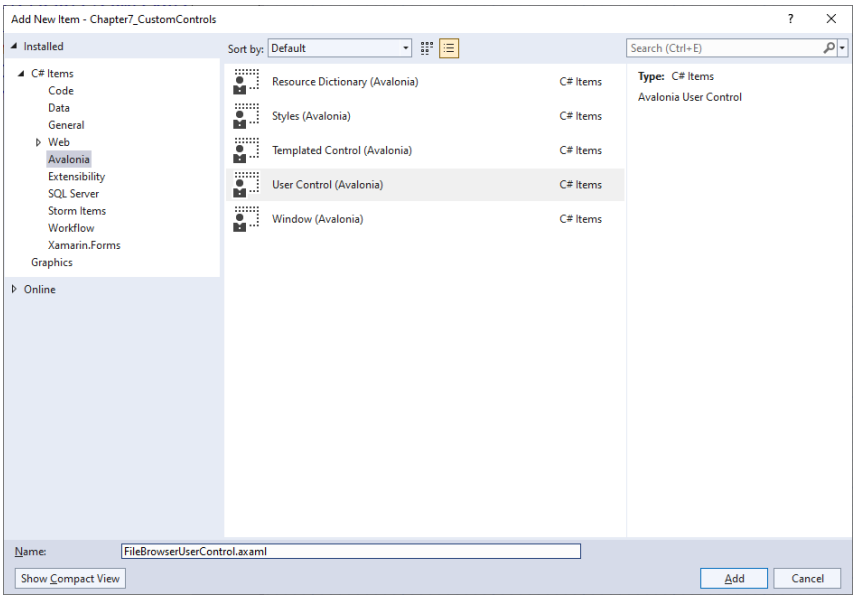



*Figure 40: Adding a new user control*

When the XAML code editor appears, you will notice that the root element in this case is **`UserControl`**, whose behavior is quite similar to **`Window`**, so all the techniques you learn in this e-book are still relevant. Now, suppose you want to create a control that allows the user to select a file and store the filename in a property. Designing the control can be done quite simply, as demonstrated in Code Listing 28.

*Code Listing 28*

```
<UserControl xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
             mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
             x:Class="Chapter7_CustomControls.FileBrowserUserControl">
        <StackPanel Orientation="Horizontal">
                <TextBlock Margin="3" Text="File name:" Height="30"/>
                <TextBox Text="{Binding RelativeSource={RelativeSource
                        FindAncestor, AncestorType={x:Type UserControl}},
                        Path=FileName}" Width="200"
                        Name="FileNameTextBox" Margin="3" Height="30"/>

                 <Button Margin="3" Name="BrowseButton"
                         Click="BrowseButton_Click"
                         Content="Browse" Height="30"/>
        </StackPanel>
</UserControl>
```

The implementation is quite simple: a **`TextBlock`**, a **`TextBox`**, and a **`Button`** are used. The important note is about the **`TextBox`**, whose **`Text`** property is bound to a property called **`FileName`**, which is exposed by the control itself (**`RelativeSource FindAncestor`**), and which must now be defined as a styled property. Code Listing 29 demonstrates this.

*Code Listing 29*

```
public partial class FileBrowserUserControl : UserControl
{
    public FileBrowserUserControl()
    {
        InitializeComponent();
    }

    private static StyledProperty<string> FileNameProperty =
        AvaloniaProperty.Register<FileBrowserUserControl,
        string>(nameof(FileName));

    public string FileName
    {
        get { return Convert.ToString(GetValue(FileNameProperty)); }
        set
        {
            SetValue(FileNameProperty, value);
            RaiseEvent(new RoutedEventArgs(fileNameChangedEvent));
        }
    }
```

```
    private static readonly RoutedEvent fileNameChangedEvent =
            RoutedEvent.Register<FileBrowserUserControl,
            RoutedEventArgs>(nameof(FileNameChanged),
            RoutingStrategies.Bubble);

    public event EventHandler<RoutedEventArgs> FileNameChanged
    {
        add { AddHandler(fileNameChangedEvent, value); }
        remove { RemoveHandler(fileNameChangedEvent, value); }
    }


    private async void BrowseButton_Click(object sender, RoutedEventArgs e)
    {
        OpenFileDialog openDialog = new OpenFileDialog();
        openDialog.Filters.Add(new FileDialogFilter { Extensions = new
            System.Collections.Generic.List<string> { "*.*" },
            Name="All files" });
        Window parentWindow = (Window)((StackPanel)Parent).Parent;
        var result = await openDialog.ShowAsync(parentWindow);
        if(result != null)
            FileName = result.FirstOrDefault();
    }
}
```

The following is a list of relevant points for Code Listing 29:

- The **FileName** property is implemented as a styled property with the syntax you learned in the previous chapter. The addition is that the set method also raises a new, custom event.
- The **FileNameChanged** event is a custom routed event, defined with the **Bubble** strategy. The **RoutedEvent.Register** method creates an instance of the **RoutedEvent** class that is registered at runtime, and the **FileNameChanged** event, of type **EventHandler<RoutedEventArgs>**, registers the handlers to subscribe to and unsubscribe from the event.
- With an approach very close to WPF, the control opens a file dialog via the **OpenFileDialog** object. File filters are represented by the **FileDialogFilter**. This exposes the **Extensions** property, a **List<string>** that represents the file extensions, and a **Name** property that represents a friendly name for the filter.
- The **ShowAsync** method of the **OpenFileDialog** requires specifying a parent window as the argument, so this is retrieved by casting to **Window** the parent of the parent **StackPanel**.

At this point, you can consume the control from a regular window. To accomplish this, you first add an XML namespace that points to the assembly where the control is defined, like in the following line, where the assembly name is taken from the companion source code:

**xmlns:local="using:Chapter7_CustomControls"**

Then you can add the control to the UI, for example, by adding the following code as the child for the main **`Window`**:

```
<StackPanel Spacing="20">
      <local:FileBrowserUserControl Name="FileBrowserControl1"/>

      <TextBlock Text="{Binding ElementName=FileBrowserControl1,
       Path=FileName}"/>
</StackPanel>
```

If you run the sample code, you will see the user control in the window, and you will be able to select a file from disk. The selected filename will be visible in the **`TextBlock`** via data binding. Figure 41 shows the result.

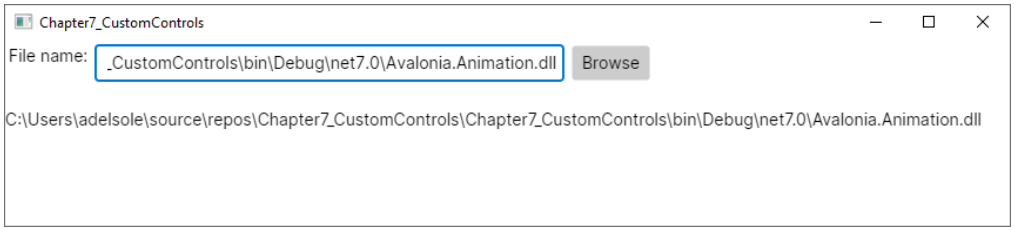



*Figure 41: The new user control in action*

Like for WPF, Avalonia UI is not limited to aggregating existing elements into one control. It also allows for redefining the look and feel of existing controls, as explained in the next section.

# Creating templated controls

Exactly like in WPF, in Avalonia UI primitive controls are *lookless*. This means that their appearance is separate from their behavior. In other words, controls have a set of properties that make them operational, while what the user sees on the screen is represented by a *control template*, which is a special graphic style.

All Avalonia UI controls have a default control template, which is what you see when you declare them in XAML code. The control template can be completely redefined inside styles, giving the control a totally different look while keeping the behavior unchanged. For example, the appearance of a button can be redefined using the geometric shape of the ellipse, but while you will see an ellipse on the screen, it is a full-fledged button with the possibility of clicking or reacting to hovering at runtime. The example you will see in this chapter is exactly about redefining the control template of a **`Button`** by using an **`Ellipse`**.

Technically speaking, when you redefine the control template of an existing control or create a new customizable control from scratch, you work with *templated controls*. Creating a templated control is simplified by a specific Avalonia template for Visual Studio. If you right-click the project name in Solution Explorer and then select **Add New Item**, you will see an item template called **Templated Control (Avalonia)**, as shown in Figure 42.

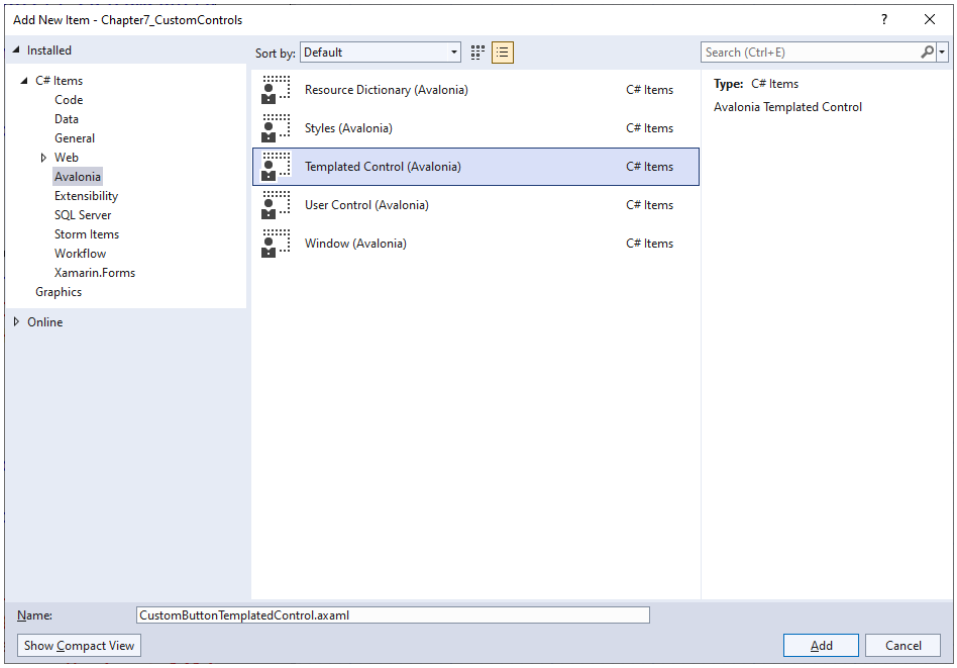



*Figure 42: Creating a templated control*

Name the new file **CustomButtonTemplatedControl.axaml** and then click **Add**. The auto-generated XAML code is related to a brand new control, but understanding how restyling an existing control works makes it easier to develop a new one. Having said that, remove all the XAML code inside the **`Styles`** node. Now that you are ready to redefine the control template of the **`Button`** control via an **`Ellipse`**, start adding the following lines:

```
<Design.PreviewWith>
   <Button Classes="custombutton" />
</Design.PreviewWith>
```

With these lines, you are instructing the Visual Studio designer to show a preview of the templated control as if it were referenced by a **`Window`** or user control. The necessity of assigning the **`Classes`** property will be discussed shortly. As mentioned previously, redefining the control template is nothing but assigning the **`Template`** property of a control with an object called **`ControlTemplate`**, whose content is the different visual element you want to use. This is demonstrated by the following code:

```
<Style Selector="Button.custombutton">
   <Setter Property="Template">
      <ControlTemplate>
         <Grid>
            <Ellipse Fill="Red" Width="{TemplateBinding Width}"
                     Height="{TemplateBinding Height}"/>
            <ContentPresenter Content="{TemplateBinding Content}"
             VerticalAlignment="Center" HorizontalAlignment="Center"/>
         </Grid>
      </ControlTemplate>
   </Setter>
```

```
</Style>
```

You already saw the **Style.Selector** property in the previous chapter, and here it is targeting a class called **custombutton**. This implies that the control template will be applied only to those **Button** controls that assign the **Classes** property with the **customcontrol** value, whereas all the others will be ignored.

You assign the **Template** property of the **Style** with a **ControlTemplate** child element. This is where you actually provide a different look to the control, and you can add a complex visual element. In the current example, a **Grid** is the root layout containing an **Ellipse** and a **ContentPresenter**. The **Ellipse** exposes the **Fill** property, which represents the shape's color.

Both the **Width** and **Height** properties invoke the **TemplateBinding** expression. This allows for the same property value to be applied to the declaring control. In other words, the control template should never have hard-coded values; instead, it should take property values from the control that is being redefined and declared in XAML.

The **ContentPresenter** represents the **Content** property of the **Button** and other content controls. It is fundamental to add a **ContentPresenter** when you restyle a content control; otherwise, you would not be able to display any content.

Before extending the template with additional states, it is a good idea to show how you consume the templated control. Assuming you want to use it inside a **Window**, you first need to reference the .axaml file that defines the control as follows (adjust the path for the .axaml file according to your folder structure):

```
<Window.Styles>
   <StyleInclude Source="avares://CustomButtonTemplatedControl.axaml"/>
</Window.Styles>
```

Now, exactly how you have done with styles back in Chapter 6, you can declare a **Button** as follows:

```
<Button Classes="custombutton" Width="150"
            Height="150" Content="A custom button"/>
```

As you assign the **Classes** property with the value targeted by the templated control's **Selector**, the new control template will be applied, and it will look as it does in Figure 43.


*Figure 43: Templated control in action*

As you can see, your **Button** looks like an **Ellipse**, but it still works like a **Button** with its properties and events. Restyling a **Button** is also a good learning opportunity, because it has different states, like normal, pressed, and hovered. You can provide a different control template depending on the pseudo-classes, like in the following code:

```
<Style Selector="Button.custombutton:pointerover">
  <Setter Property="Template">
      <ControlTemplate>
         <Grid>
            <Ellipse Fill="LightGreen" Width="{TemplateBinding Width}"
             Height="{TemplateBinding Height}"/>
            <ContentPresenter Content="{TemplateBinding Content}"
             VerticalAlignment="Center" HorizontalAlignment="Center"/>
         </Grid>
      </ControlTemplate>
  </Setter>
</Style>
<Style Selector="Button.custombutton:pressed">
  <Setter Property="Template">
      <ControlTemplate>
         <Grid>
            <Ellipse Fill="Yellow" Width="{TemplateBinding Width}"
             Height="{TemplateBinding Height}"/>
            <ContentPresenter Content="{TemplateBinding Content}"
             VerticalAlignment="Center" HorizontalAlignment="Center"/>
         </Grid>
      </ControlTemplate>
  </Setter>
</Style>
```

With this code, the **`Ellipse`** will be filled with the **`LightGreen`** color when you hover over the button, whereas it will be filled with **`Yellow`** when you keep the **`Button`** pressed. In terms of behavior, you can extend a templated control with the following:

- **Styled properties:** You have seen how to implement styled properties in the previous section, and they are still the way you implement bindable properties with templated controls.
- **Custom routed events:** These have also been discussed in the previous section, and they are the way you implement events that need to be raised across the visual tree.

As you can see, this example has provided a very basic control template, but you can actually implement complex visual elements to completely redefine the look and feel of controls. The documentation does not include a complete tutorial about creating new templated controls from scratch, but there are community resources available, so you can watch this YouTube video about this topic.

 ***Note: Templated controls are very powerful but should be used carefully. The end user expects simplicity and an easy connection between the control and the action it is intended for.***

## Chapter summary

Creating reusable controls is crucial in every development platform, and Avalonia UI is no exception. There are many similarities with the WPF approach, due to the fact that you can create user controls, which are the result of the aggregation of multiple controls into one, and templated controls, which are primitive, lookless controls where you completely redefine the look and feel while keeping the behavior unchanged.

For both types, you can implement styled properties that represent the target of a data-binding expression and new routed events. It was important to discuss controls at this particular point, especially user controls, because they are at the core of the topic discussed in the next chapter: navigation between pages.

# Chapter 8 Working with Windows, Pages, and Dialogs

Applications normally work on multiple pages. In the case of desktop applications, you usually implement multiple windows. In the case of web and mobile applications, you typically implement pages. There are different ways to implement navigation between windows or pages in Avalonia UI, depending on the application type. This chapter will first provide an overview of multi-window applications for the desktop and then explain one possible solution to implement navigation for cross-platform projects.

## Multiple windows on desktop applications

You can have multiple **`Window`** objects in your application and access them programmatically. For example, suppose you have a new Avalonia UI desktop solution and want to open a secondary window from the main one. If you right-click the project name in Solution Explorer and then select **Add New Item**, you can add a new window by selecting the **Window (Avalonia)** item template (see Figure 44).

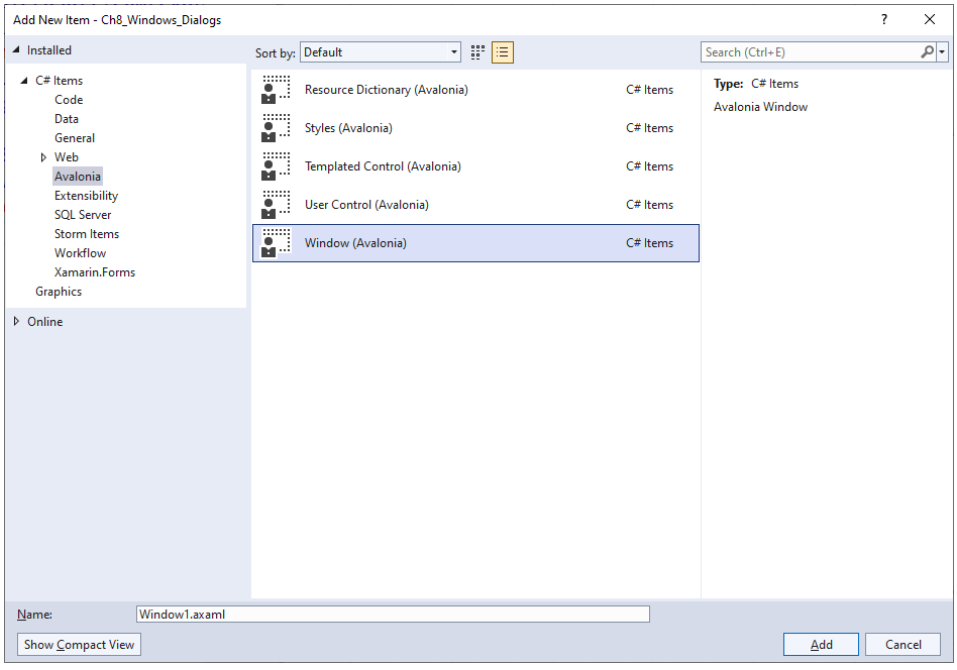



*Figure 44: Adding a new window*

Call the file **SecondaryWindow.axaml** and then click **Add**. In the XAML code for the MainWindow.axaml file, add the following simple code that will allow for opening a new window with a button click:

```
<StackPanel>
```

```
      <Button Name="OpenWindowButton"
                  Content="Open new window"
                  Margin="10" Click="OpenWindowButton_Click"/>
</StackPanel>
```

The **Click** event handler contains the code that opens a new window, as follows:

```
private void OpenWindowButton_Click(object sender, RoutedEventArgs e)
{
    var newWindow = new SecondaryWindow();
    newWindow.Show();
}
```

This code creates an instance of the specified window and opens it. When you invoke **Show**, both the main and secondary windows are enabled, so you can use both (or all the windows that you have opened). In some cases, you might want the secondary window to be opened as a modal dialog, for example, to force the user to complete an action without being able of interacting with other pieces of the user interface.

In this case, you can invoke the following method:

```
newWindow.ShowDialog(this);
```

The argument for **ShowDialog** is an instance of the owner **Window**. In both cases, you can control the position and state of the window. You can assign the **WindowStartupLocation** property with one of the following self-explanatory values: **CenterOwner**, **CenterScreen**, or **Manual**. The state of the window can be assigned via the **WindowState** property with self-explanatory values like: **FullScreen**, **Maximized**, **Minimized**, and **Normal**.

You can also subscribe for a window's events as you would normally do in C# code; for example, you could subscribe to the **Window.Closed** event if you need to understand when the secondary window has been closed:

```
private void OpenWindowButton_Click(object sender, RoutedEventArgs e)
{
    var newWindow = new SecondaryWindow();
    newWindow.Closed += NewWindow_Closed;
    newWindow.Show();
}

private void NewWindow_Closed(object? sender, System.EventArgs e)
{
    // Secondary window has been closed
}
```

You have a lot of flexibility when working with windows, but you have alternatives such as tabs.

## Implementing tabs

As an alternative to multiple windows, you can organize the user interface with tabs. This approach is very common, and it is certainly more suitable for cross-platform projects that target different devices and systems. In Avalonia UI, you can use the **`TabControl`** visual element to organize the UI with tabs. The following XAML demonstrates how to implement a **`TabControl`**:

```
<TabControl>
      <TabItem Header="First tab">
            <TextBlock Text="Your content here.."/>
      </TabItem>
      <TabItem Header="Second tab">

      </TabItem>
      <TabItem Header="Third tab">

      </TabItem>
</TabControl>
```

A **`TabControl`** is basically a container for **`TabItem`** elements, each representing a tab. Every **`TabItem`** has a **`Header`** and content. Because they are content controls, their content can be any visual element. In addition, they can be customized by assigning the properties listed in Table 6. If you add the code above to the XAML of the secondary window created before, you will get the result shown in Figure 45.

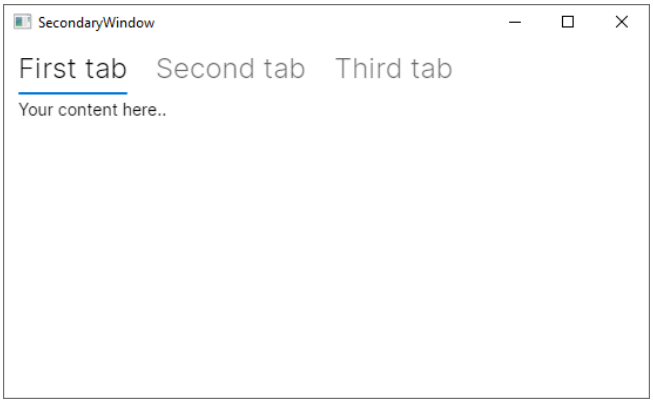



*Figure 45: Organizing the UI with tabs*

***Tip: You can also customize the header and support data binding by assigning the `HeaderTemplate` property with a `DataTemplate` object populated with a visual element***

***(for example, a panel with multiple child views) instead of assigning the `Header` property with a string.***

There is also another control, called **`TabStrip`**, whose goal is to display a list of tabs as a menu. It is declared like the **`TabControl`** and still contains **`TabItem`** objects, but unlike the **`TabControl`**, the content inside **`TabItem`** objects is ignored. However, since tab items should act like menu items, you can handle the **`Tapped`** event, which is raised when the tab is touched (on devices that support touch gestures) or clicked. You would declare it as follows:

```
<TabStrip>
      <TabItem Header="First tab" Tapped="TabItem_Tapped" />
</TabStrip>
```

The layout of a **`TabStrip`** is the same as a **`TabControl`**, but without any content for **`TabItem`** objects. This is immediately visible in the Visual Studio designer. The **`Tapped`** event can be handled as follows:

```
private void TabItem_Tapped(object sender, TappedEventArgs e)
{

}
```

Therefore, you need to handle this event to support interaction over tabs.

# Implementing navigation between pages

When you create an Avalonia UI cross-platform solution that targets desktop systems, the web browser, and mobile devices, you cannot use windows only. In other development platforms, you would have a navigation framework that allows you to implement and browse pages. Though Avalonia UI does not implement a navigation framework per se, it provides the **Avalonia ReactiveUI** library. This library is available as a NuGet package and provides an additional layer to the Avalonia UI codebase, providing everything you need to implement navigation. It is based on the MVVM pattern and on logic that probably requires some time to learn if you are not familiar with more advanced MVVM concepts, but it exposes objects that can be used beyond the need to navigate between pages.

The cross-platform solution template you saw in Chapter 2, in the section called “Creating cross-platform solutions,” is based on the ReactiveUI library. It already includes most of the things that you can use for a quicker navigation startup. However, a better idea is to work with the official ToDo app sample because the documentation has a dedicated explanation page about this sample app. This includes some views and navigation objects that are more useful for learning purposes.

Now, let’s download the source code or clone the repository, and open the solution in Visual Studio. When you’re ready, run the project. You will see a main page that shows a list of items and a button that allows you to add a new item to the list. When you click this button, you will be able to add a new item via a new page that gives you the option to confirm or cancel the operation. Figure 46 shows the two pages in the application.

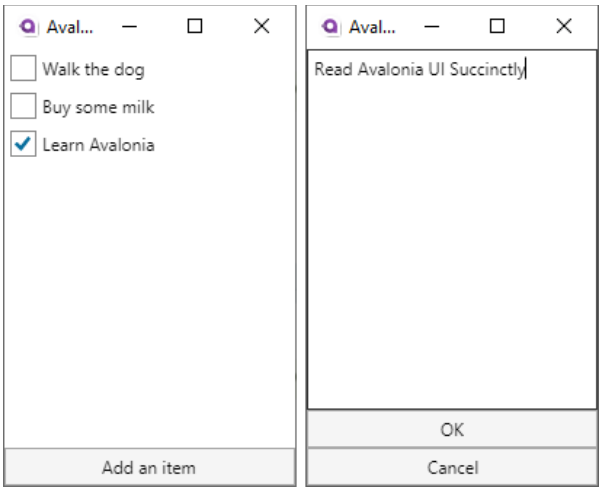



*Figure 46: The ToDo app in action*

Pages are actually represented by **`UserControl`** items, and each defines the entire visual tree of a page. The way the application navigates between user controls and the way these exchange data between one another is based on data binding, commands, and the MVVM pattern via objects that are exposed by the ReactiveUI library. Having said this, it is a good idea to dissect how the flow works from a different point of view, starting from the main window and the way it consumes user controls.

## The main window and the main ViewModel

The definition of the main window is extremely simple:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="Todo.Views.MainWindow"
        Icon="/Assets/avalonia-logo.ico"
        Width="200" Height="300" Title="Avalonia Todo"
        Content="{Binding Content}">
</Window>
```

The purpose of the main window is to act as a container of pages, rather than an individual page itself. As you will discover shortly, pages are built as user controls that are rendered as the window's child visual element. The content of the window is represented by the **`Content`** property, which is bound to the same-named property of a ViewModel class called **`MainWindowViewModel`**. Code Listing 30 shows the content of the ViewModel.

*Code Listing 30*

```
class MainWindowViewModel : ViewModelBase
{
    ViewModelBase content;
```

```
    public MainWindowViewModel(Database db)
    {
        Content = List = new TodoListViewModel(db.GetItems());
    }

    public ViewModelBase Content
    {
        get => content;
        private set => this.RaiseAndSetIfChanged(ref content, value);
    }

    public TodoListViewModel List { get; }

    public void AddItem()
    {
        var vm = new AddItemViewModel();

        Observable.Merge(
            vm.Ok,
            vm.Cancel.Select(_ => (TodoItem)null))
            .Take(1)
            .Subscribe(model =>
            {
                if (model != null)
                {
                    List.Items.Add(model);
                }

                Content = List;
            });

        Content = vm;
    }
}
```

Before discussing the details of the **ViewModel** class, the first thing to say is that it derives from another class called **ViewModelBase**, whose definition is the following:

```
public class ViewModelBase : ReactiveObject
{
}
```

The goal of this class is to offer a base with common objects to specialized ViewModels. In this particular example, there is no additional member, but you would ideally add any method or property that derived ViewModels could benefit from here. The key point is that this class inherits from **ReactiveObject**, a class exposed by the ReactiveUI library that already implements change notification, so you do not need to implement the **INotifyPropertyChanged** interface manually.

The following is a list of key points in the **MainWIndowViewModel** class:

- When the **Content** property is assigned, the setter invokes a method called **RaiseAndSetIfChanged**, which assigns the property value and raises a change notification. This is inherited from **ReactiveObject** and avoids the need to implement the **INotifyPropertyChanged** interface.
- The constructor assigns an instance of the **ToDoListViewModel** class, via a property called **List**, to the **Content** property. This ViewModel is discussed in the next section, but the important thing to notice here is that it represents a page called **ToDoListView**, that ReactiveUI implements navigation between ViewModels, and that the **ViewLocator** class (discussed shortly) renders the related view.
- Still on the constructor, it works with a class called **Database**. The purpose of this class is just to create sample data, specifically a list of **TodoItem** objects. These are made of two simple properties: **Description** of type **string**, and **IsChecked** of type **bool**. It is not a real database implementation, only in-memory data, but it is enough to understand how and where you could handle data, like a SQLite database. Code Listing 31 contains the definition for both classes.
- The **AddItem** method represents a command that will be bound to a button in the **ToDoListView** page. It creates an instance of the **AddNewItemViewModel** class (discussed later), and merges (**Observable.Merge**) a newly added **TodoItem** object into the current **List**. This is done by combining a method called **Ok**, which confirms the addition from the secondary page, and subscribing for the **!= null** condition over the item. If the condition is instead **== null**, the addition is canceled (**Cancel** method).

*Code Listing 31*

```
namespace Todo.Services
{
    public class Database
    {
        public IEnumerable<TodoItem> GetItems() => new[]
        {
            new TodoItem { Description = "Walk the dog" },
            new TodoItem { Description = "Buy some milk" },
            new TodoItem { Description = "Learn Avalonia",
                IsChecked = true },
        };
    }
}

namespace Todo.Models
{
    public class TodoItem
    {
        public string Description { get; set; }
        public bool IsChecked { get; set; }
    }
}
```

The **MainWindowViewModel** represents a connection between the other ViewModels. In the next section, you will see how the **ToDoListViewModel** class is implemented and how a page is built on top of it.

## Working with pages and ViewModels

With ReactiveUI, pages are represented by user controls, and each page has a backing ViewModel. Unlike other platforms, where you navigate by invoking another page, with ReactiveUI you invoke the ViewModel for the page you intend to display. The library then knows how to connect the ViewModel to the appropriate page and what to render. In the next paragraphs, this concept will be clarified.

If you look back at Code Listing 30, you can see how the content for the main **Window** is an instance of the **TodoListViewModel** class, created by the **MainWindowViewModel**. This means that the **TodoListViewModel** must be coupled to a user control that becomes the first page for the application. The code for this ViewModel is very simple:

```
public class TodoListViewModel : ViewModelBase
{
    public TodoListViewModel(IEnumerable<TodoItem> items)
    {
        Items = new ObservableCollection<TodoItem>(items);
    }

    public ObservableCollection<TodoItem> Items { get; }
}
```

The relevant point of this class is the **Items** property, of type **ObservableCollection<TodoItem>**, which contains the list of items that will be displayed in the main list (see the left side of Figure 46). The user control that consumes this ViewModel is defined via the XAML code shown in Code Listing 32.

*Code Listing 32*

```
<UserControl xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
             mc:Ignorable="d" d:DesignWidth="200" d:DesignHeight="300"
             x:Class="Todo.Views.TodoListView">
  <DockPanel>
    <Button DockPanel.Dock="Bottom"
            Command="{Binding $parent[Window].DataContext.AddItem}">
      Add an item
    </Button>
    <ItemsControl Items="{Binding Items}">
      <ItemsControl.ItemTemplate>
```

```
        <DataTemplate>
          <CheckBox Margin="4"
                    IsChecked="{Binding IsChecked}"
                    Content="{Binding Description}"/>
        </DataTemplate>
      </ItemsControl.ItemTemplate>
    </ItemsControl>
  </DockPanel>
</UserControl>
```

There are two relevant points in Code Listing 32:

- The button that allows adding a new item is bound to a command method defined in the **`MainWindowViewModel`**. The binding expression has a particular syntax: **`$parent[Window]`** points to the parent container for the user control, which is in fact a **`Window`**; **`DataContext`** is the property to which the **`MainWindowViewModel`** instance is bound; and **`AddItem`** is the actual invocation to the command method that adds a new **`TodoItem`** object to the list.
- The **`ItemsControl`** shows the list of items by binding the **`Items`** property of the ViewModel to the **`Items`** property of the control. A simple **`DataTemplate`** is defined to display the status and description of each object.

The code-behind file for the user control is very simple and is shown in Code Listing 33.

*Code Listing 33*

```
using Avalonia.Controls;
using Avalonia.Markup.Xaml;

namespace Todo.Views
{
    public class TodoListView : UserControl
    {
        public TodoListView()
        {
            this.InitializeComponent();
        }

        private void InitializeComponent()
        {
            AvaloniaXamlLoader.Load(this);
        }
    }
}
```

The only relevant point here is the invocation of the **Load** method from the **AvaloniaXamlLoader** class. This helper object provides members whose goal is to simplify how the runtime works with XAML. The **Load** method simply loads some XAML into an Avalonia UI component.

***Tip: The invocation to `AvaloniaXamlLoader.Load` is also visible in the MainWindow.axaml.cs file with exactly the same purpose.***

When the user clicks the button to add a new item, the **AddItem** method from the **MainWindowViewModel** class is invoked. If you look back at its definition in Code Listing 30, you can see how an instance of the **AddItemViewModel** class is created. As you can imagine, this is coupled with a user control called **AddItemView**. For a better understanding, the best approach is to start from the definition of the user control, shown in Code Listing 34.

*Code Listing 34*

```
<UserControl xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
             mc:Ignorable="d" d:DesignWidth="200" d:DesignHeight="300"
             x:Class="Todo.Views.AddItemView">
  <DockPanel>
    <Button DockPanel.Dock="Bottom" Command="{Binding Cancel}">Cancel
    </Button>
    <Button DockPanel.Dock="Bottom" Command="{Binding Ok}">OK</Button>
    <TextBox AcceptsReturn="False"
             Text="{Binding Description}"
             Watermark="Enter your TODO"/>
  </DockPanel>
</UserControl>
```

The code is quite simple to understand due to the knowledge that you have gained so far. The relevant point is that the two buttons are bound to two commands, **Cancel** and **Ok**, whereas the **TextBox** is bound to the **Description** property of a new **TodoItem** object assigned in C# code. Both commands are defined in the **AddItemViewModel** class, whose code is shown in Code Listing 35.

*Code Listing 35*

```
using System.Reactive;
using ReactiveUI;
using Todo.Models;

namespace Todo.ViewModels
{
    class AddItemViewModel : ViewModelBase
```

```
    {
        string description;

        public AddItemViewModel()
        {
            var okEnabled = this.WhenAnyValue(
                x => x.Description,
                x => !string.IsNullOrWhiteSpace(x));

            Ok = ReactiveCommand.Create(
                () => new TodoItem { Description = Description },
                okEnabled);
            Cancel = ReactiveCommand.Create(() => { });
        }

        public string Description
        {
            get => description;
            set => this.RaiseAndSetIfChanged(ref description, value);
        }

        public ReactiveCommand<Unit, TodoItem> Ok { get; }
        public ReactiveCommand<Unit, Unit> Cancel { get; }
    }
}
```

This class introduces new concepts of the ReactiveUI library, and it provides a good opportunity to understand more about its power. Starting from the bottom, you can see the definition of two objects of type **`ReactiveCommand`**. This type encapsulates an interaction between the user and a ReactiveUI interface, and when using this library, it can be used instead of command methods.

Both the **`Ok`** and **`Cancel`** commands are exposed as read-only properties of type **`ReactiveCommand`**, and they share the first type parameter, which is of type **`System.Reactive.Unit`**. This is used to specify that a method or property can return only one value. Then, in the case of **`Ok`**, the second type parameter is an object of type **`TodoItem`**, which represents the item that is committed to the data source when the OK button is pressed. In the case of **`Cancel`**, the second type parameter is another **`Unit`**, which simply returns no value. Both commands are objects that need to be instantiated before they can be used.

In Code Listing 35, you can see how the **`Create`** method of the **`ReactiveCommand`** class generates the command instances. In the case of **`Ok`**, a new **`TodoItem`** is created, and its **`Description`** property is assigned with the content that the user enters through the user interface.

It is also possible to specify some validation rules before the new item is added to the data source. In this case, the validation is performed via the **`WhenAnyValue`** method (included in the ReactiveUI library). Each method argument is a delegate provided and described in the following order:

1. The property whose value needs to be evaluated (**`x.Description`** in this case)
2. The condition that needs to be satisfied (**`!string.IsNullOrWhiteSpace(x)`** in this case)

When the conditions are satisfied (**`Description`** is not null or white space), an object of type **`IObservable<bool>`** is returned, and the **`ReactiveCommand.Create`** method uses this to determine whether the command can be executed or not, and, consequently, whether the bound control should be enabled. The **`Cancel`** command can always be executed, so it does not need validation rules and is simply instantiated via **`ReactiveCommand.Create`**.

So far, you have defined some user controls for pages and their operational ViewModels, but there is still the most important missing piece, something that connects ViewModels between one another and that allows for navigation between pages. This is discussed in the next section.

## Resolving pages from ViewModels: The ViewLocator class

The ReactiveUI library implements a class called **`ViewLocator`**, which is responsible for displaying views based on their backing ViewModels. Code Listing 36 shows its definition. Notice that comments have been added for you, so they are not available in the solution you download from GitHub.

*Code Listing 36*

```
public class ViewLocator : IDataTemplate
{

    public bool SupportsRecycling => false;

    //Create an instance of a user control
    public IControl Build(object data)
    {
        //Retrieve a user control name starting from its backing ViewModel
        var name = data.GetType().FullName.Replace("ViewModel", "View");

        //Retrieve the actual type for the control
        var type = Type.GetType(name);

        if (type != null)
        {
            //Return an instance of a user control at runtime
            return (Control)Activator.CreateInstance(type);
        }
        else
        {
            //No backing ViewModel found
            return new TextBlock { Text = "Not Found: " + name };
        }
    }
```

```
    //Return true if the data object derives from ViewModelBase
    public bool Match(object data)
    {
        return data is ViewModelBase;
    }
}
```

When a project is based on the ReactiveUI library, the runtime knows that the **`ViewLocator`** class must be instantiated, and it delegates to this class the responsibility of matching user controls with their ViewModels. The **`Build`** method retrieves the name of the view based on the name of the ViewModel, so you now better understand the reason for the naming convention introduced at the beginning of this topic. If a user control can be resolved based on the name, an instance of such a control is created and returned to the consumer (the main window).

In summary, differently from other cross-platform development environments where you have a navigation framework, with Avalonia UI you:

- Leverage the power of the ReactiveUI library to implement a view locator, commands, and objects that implement change notification.
- Create a user control for each page you want to display.
- Create a ViewModel for each user control and implement all the operational code.

You are certainly free to explore different options to implement your own navigation system, but if you plan to work cross-platform, you will still need to work with user controls within a container window.

## Chapter summary

Applications are normally built on top of multiple windows for desktop apps, and on multiple pages for mobile apps. When you work on the desktop version, you can simply create **`Window`** objects and show them as floating or modal dialogs. If you prefer to just have one window (or page), you can organize the user interface with tabs via the **`TabControl`** visual element. When you work with mobile apps, or more generally with cross-platform projects, you might want to work with pages.

Avalonia UI does not have a built-in navigation framework, but it provides the ReactiveUI library, which simplifies the way you create pages by connecting view models to user controls. So far, you have seen most of what Avalonia UI offers in terms of user interface, but there is actually more. With graphics and animations, you will be able to enrich the look and feel of your user interface. These are discussed in the next chapter.

# Chapter 9 Brushes, Graphics, and Animations

At its core, Avalonia UI is a development platform that enhances the user interface of your applications. To do so, it is not limited to its control library. There are additional elements of the user interface that you can leverage to create powerful visual experiences for your users. This chapter describes three more elements: brushes, 2D graphics, and animations, completing the knowledge you need to have to be productive with Avalonia UI.

## Working with brushes

In your developer life, you likely had to set the background color of a window, a control, or an interface element hundreds of times, or simply had to change the foreground color of portions of text, using solid colors or pictures, for example, to create interesting texture-like effects. Avalonia UI is also a special platform from this point of view, because with very few simple lines of code, it allows you to apply fill effects using solid colors and gradients, pictures, and even other interface elements. This is possible thanks to objects called *brushes* that inherit from the **`Brush`** class. The available brushes are summarized in Table 12.

*Table 12: Available brushes*

| Value | Description |
|---|---|
| **`SolidColorBrush`** | Allows for filling an object with a solid color |
| **`LinearGradientBrush`** | Allows for filling an object with a gradient of colors with a linear direction |
| **`RadialGradientBrush`** | Allows for filling an object with a circular gradient of colors |
| **`ImageBrush`** | Allows for filling an object with an image |
| **`VisualBrush`** | Allows for filling an object with the content of another visual element |

You typically use brush objects to assign the **`Foreground`** and **`Background`** properties of controls and panels, but also to fill in geometric shapes. For a better understanding, first consider Figure 47, where:

- The root panel's background has been filled with a **`SolidColorBrush`**.

- The **Grid** at the upper-left corner has been filled with a **LinearGradientBrush**.
- The **Grid** at the upper-right corner has been filled with a **RadialGradientBrush**.
- The **Grid** at the bottom-left corner has been filled with an **ImageBrush**.
- The bottom-right corner contains a **StackPanel** with a **Button** whose background has been filled with a **VisualBrush**, which points to another **StackPanel** that contains a **TextBlock** and a **TextBox**.

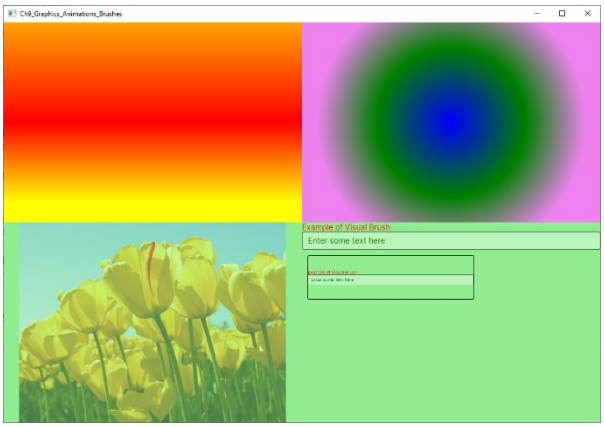



*Figure 47: Applying brushes*

The following paragraphs discuss the various brushes in more detail.

## Working with solid colors

You paint visual elements with a solid color via the **SolidColorBrush** object. The following code shows an example:

```
<Grid>
   <Grid.Background>
      <SolidColorBrush Color="LightGreen"/>
   </Grid.Background>
</Grid>
```

You assign the **Color** property a color. You can implicitly specify a solid color by assigning the color to the property of interest, like this:

```
<Grid Background="LightGreen"/>
```

Put succinctly, when you assign control properties like **Foreground** or **Background** with a color, you are actually assigning a **SolidColorBrush**.

***Tip: All the brushes discussed in this section can also be reusable resources. You can define them in the scope you need, and then you assign the x:Key identifier so they can be consumed multiple times.***

## Applying gradients

In Avalonia UI, you have the option to use two different gradient brushes: **`LinearGradientBrush`** and **`RadialGradientBrush`**. The first object applies a linear gradient, whereas the second object applies a circular gradient. Based on Figure 47, the following code applies the linear gradient:

```
<Grid>
      <Grid.Background>
            <LinearGradientBrush StartPoint="0%,0%" EndPoint="0%,100%">
                  <GradientStop Offset="0" Color="Orange"/>
                  <GradientStop Offset="0.5" Color="Red"/>
                  <GradientStop Offset="0.9" Color="Yellow"/>
            </LinearGradientBrush>
      </Grid.Background>
</Grid>
```

Each color in the gradient is represented by a **`GradientStop`** object, and its **`Color`** property is assigned with the color of interest, while the **`Offset`** property represents the position of the color within the gradient, with a value between 0 and 1. The **`StartPoint`** and **`EndPoint`** properties of the **`LinearGradientBrush`** determine the direction of the gradient through coordinates of the type x, y where both x and y are percentage values between 0 and 100.

If you have worked with linear gradients in WPF, you might remember that the value for **`StartPoint`** and **`EndPoint`** is a double between 0.0 and 1.0. In Avalonia, these values must be specified in percentage, and you must enter the **`%`** symbol. The **`RadialGradientBrush`** shares most of the properties, but it does not allow for specifying a start point and endpoint due to its circular nature:

```
<Grid Grid.Column="1">
      <Grid.Background>
            <RadialGradientBrush>
                  <GradientStop Offset="0" Color="Blue"/>
                  <GradientStop Offset="0.5" Color="Green"/>
                  <GradientStop Offset="0.9" Color="Violet"/>
            </RadialGradientBrush>
      </Grid.Background>
</Grid>
```

Gradients are certainly powerful, but there is more, as discussed in the next paragraphs.

## Filling objects with images

You can use an image to fill an object by using the **ImageBrush** class as follows:

```
<Grid Grid.Row="1">
   <Grid.Background>
        <ImageBrush Opacity="0.5" Source="/tulips.jpg" />
    </Grid.Background>
</Grid>
```

You specify the image file via the **Source** property, and the transparency via the **Opacity** property, of type **double**, whose value is between 0 and 1. This can be particularly interesting if you want to create a texture effect, for example, over the **Foreground** property of a **TextBlock** control.

## Dynamic painting with visual elements

An interesting option in Avalonia UI (like in WPF) is the possibility of filling an object with another visual element. Consider the following code:

```
<StackPanel Grid.Row="1" Grid.Column="1">
      <StackPanel Name="Stack1">
            <TextBlock Text="Example of Visual Brush" Foreground="Red" />
            <TextBox Text="Enter some text here" Foreground="Green" />
      </StackPanel>

      <StackPanel VerticalAlignment="Center">
            <Button BorderBrush="Black"
                    Margin="10,10,10,10" Height="80" Width="300">
                  <Button.Background>
                        <VisualBrush Visual="{Binding ElementName=Stack1}"/>
                  </Button.Background>
            </Button>
      </StackPanel>
</StackPanel>
```

The **Background** property of the **Button** is assigned by a **VisualBrush** object. This allows for filling the background (or other property of type **Brush**) with another visual element. In this case, the background of the button is a **StackPanel** with all its content. The content is specified via the **Visual** property, assigned with a binding expression that requires specifying the source's name (**Binding ElementName**).

# Introducing shapes

Avalonia UI provides several geometric shapes that you can use to create the following 2D drawings:

- **`Rectangle`**, which allows for drawing rectangles and squares.
- **`Ellipse`**, which allows for drawing ellipses and circles.
- **`Line`**, which allows for drawing an individual line.
- **`Polyline`**, which allows for drawing lines connected to one another.
- **`Polygon`**, which allows for drawing a variety of polygons.
- **`Path`**, which allows for creating complex drawings via XAML points.

All these shapes inherit from the **`Shape`** base class. They also share the properties described in Table 13.

*Table 13: Shape properties*

| Property | Type | Description |
|---|---|---|
| **`Fill`** | **`Brush`** | The brush used to fill the shape's interior |
| **`Stroke`** | **`Brush`** | The brush used for the shape's outline |
| **`StrokeThickness`** | **`double`** | The width of the shape's outline with a default of 0 |
| **`StrokeDashArray`** | **`AvaloniaList<double>`** | A collection of values that indicate the pattern of dashes and gaps used to outline a shape |
| **`StrokeDashOffset`** | **`double`** | The distance between dashes |

In the following paragraphs, you will see how to draw geometric shapes with hints about their usage in applications.

***Tip: You are not limited to using shapes only to create drawings. Being visual elements, they can also be assigned to the `Content` property of content controls.***

## Drawing rectangles

The first shape we will discuss is the **`Rectangle`**. The following example helps understanding how you declare the shape, and how you can use brushes:

```
<Rectangle Stroke="Green" StrokeThickness="4" StrokeDashArray="1,1"
 StrokeDashOffset="6" Width="250" Height="100"
 Margin="0,50,0,0">
      <Rectangle.Fill>
            <LinearGradientBrush>
                  <GradientStop Color="LightBlue" Offset="0"/>
```

```
                <GradientStop Color="Blue" Offset="0.5"/>
                <GradientStop Color="Violet" Offset="1"/>
            </LinearGradientBrush>
        </Rectangle.Fill>
</Rectangle>
```

The resulting shape is the first one from the top of Figure 48.

## Drawing ellipses and circles

The second shape is the **Ellipse**, which you use to draw ellipses and circles. The following code provides an example where the ellipse is filled in violet and the outline is green, with a thickness of **3**:

```
<Ellipse Fill="Violet" Stroke="Green" StrokeThickness="3"
 Width="250" Height="100" Margin="0,50,0,0"/>
```

The resulting shape is the second one from the top in Figure 48.

## Drawing lines

The next shape is the **Line**. Let's start with some code:

```
<Line StartPoint="10,0" EndPoint="230,0" StrokeLineCap="Round"
 Stroke="Violet" StrokeThickness="12" Margin="0,50,0,0" />
```

**StartPoint** and **EndPoint** represent the starting and ending points of the line. The value on the left of the comma represents the position on the X axis, whereas the value on the right of the comma represents the position on the Y axis.

The **StrokeLineCap** property, of type **PenCap**, describes a shape at the end of a line and can be assigned with **Flat** (default), **Square**, or **Round**. **Flat** basically draws no shape, **Square** draws a rectangle with the same height and thickness of the line, and **Round** draws a semicircle with the diameter equal to the line's thickness. The resulting shape is the third from the top shown in Figure 48.

## Drawing polygons

Polygons are complex shapes, and Avalonia UI provides the **Polygon** class to draw these geometries. The following example draws a triangle filled in violet and with a green, dashed outline:

```
<Polygon Points="50,20 80,60 20,60" Fill="Violet" Stroke="Green"
 StrokeThickness="4" StrokeDashArray="1,1" StrokeDashOffset="6"/>
```

The resulting shape is the fourth from the top in Figure 48. The key property in the **Polygon** shape is **Points**, a collection of **Point** objects that each represent the coordinates of a specific delimiter in the polygon. Because there's basically no limit to the collection of points, you can create very complex polygons.

## Drawing complex shapes with the Polyline

The **Polyline** is a particular shape that allows for drawing a series of straight lines connected to one another, but where the last line does not connect with the first point of the shape. Consider the following example:

```
<Polyline Margin="0,50,0,0" Points="0 48, 0 144, 96 150, 100 0, 192 0, 192
 96, 50 96, 48 192, 150 200 144 48"
 Fill="Violet" Stroke="DarkGreen" StrokeThickness="3"/>
```

Similarly to the **Polygon**, the **Polyline** exposes a property called **Points**, a collection of **Point** objects that each represent the coordinates of a point. The resulting shape is the fifth from the top in Figure 48.

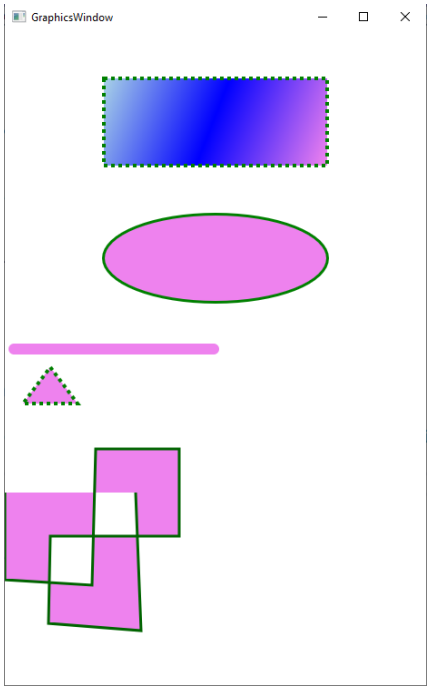



*Figure 48: Drawing shapes*

## Hints about geometries and paths

The new drawing possibilities are not limited to the basic shapes described in the previous sections. Avalonia UI also offers powerful 2D drawings with *geometries* and provides the **`Path`** and **`PathIcon`** classes that allow drawing curves and complex shapes using geometries. Both are very complex and long topics that cannot fit inside a book of the *Succinctly* series. For this reason, I recommend taking a look at the official documentation page.

# Adding animations

In Avalonia UI, there are two types of animations: keyframe animations and easing animations. Despite similar names, they work differently from WPF. This section explains keyframe animations in detail and provides hints about the easing functions.

## Keyframe animations

Keyframe animations work with styles and selectors, and their goal is to animate property values of an object. Consider Code Listing 37, where a keyframe animation is applied to an **`Ellipse`**.

*Code Listing 37*

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        mc:Ignorable="d" d:DesignWidth="800" d:DesignHeight="450"
        x:Class="Ch9_Graphics_Animations_Brushes.AnimationsWindow"
        Title="AnimationsWindow">
       <Window.Styles>
          <Style Selector="Ellipse.blue">
            <Setter Property="Height" Value="100"/>
            <Setter Property="Width" Value="100"/>
            <Setter Property="Fill" Value="Blue"/>
            <Style.Animations>
              <Animation Duration="0:0:15">
                 <KeyFrame Cue="0%">
                      <Setter Property="Opacity" Value="0.0"/>
                      <Setter Property="Height" Value="0"/>
                 </KeyFrame>
                 <KeyFrame Cue="100%">
                      <Setter Property="Opacity" Value="1.0"/>
                      <Setter Property="Height" Value="100"/>
                  </KeyFrame>
              </Animation>
             </Style.Animations>
```

```
            </Style>
        </Window.Styles>

        <Ellipse Classes="blue"/>
</Window>
```

The style is applied to the **Ellipse** via selectors and the syntax you learned in Chapter 6, but it adds the **Animations** node where you add an **Animation** object. The **Duration** property specifies the animation duration with the *hh:mm:ss* format. **KeyFrame** objects represent individual moments in the animation, and their **Cue** property determines, in percentage, what needs to happen in a specific moment of the animation progress.

In the current example, the first **KeyFrame.Cue** assigns the **Opacity** property of the **Ellipse** a value of **0.0**, which means full transparency, and the **Height** property a value of **0** at the beginning of the animation. In contrast, the second assignment assigns the **Opacity** a value of **1.0**, which means no transparency, and the **Height** property a value of **100** when the animation is 100% completed.

It is not possible to render an animation on a printed figure, so you can try running Code Listing 37 and see the resulting animation on your machine. You can further control the animation via the properties listed in Table 14.

*Table 14: Animation properties*

| Property | Description |
|---|---|
| **Delay** | Adds a delay in the animation rendering. |
| **IterationCount** | The number of iterations for the animation. Supported values are **0**, an integer representing the number, or **INFINITE**. |
| **PlaybackDirection** | Determines how the animation should be played. Possible values are **Normal**, **Reverse** (in reverse direction, from 100% to 0%), **Alternate** (played forward first and then backward), and **AlternateReverse** (played backward first and then forward). |

These properties also apply to easing functions, as discussed in the next section.

## Hints about easing functions

Easing functions apply mathematical formulas to animations. The following code provides an example based on the **QuarticEaseIn** function, and it is applied to a **TextBlock** control that you can declare in your XAML code, assigning the **Classes** property with **red**:

```
<Style Selector="TextBlock.red">
      <Setter Property="Foreground" Value="Red"/>
      <Style.Animations>
            <Animation Duration="0:0:15"
                        Easing="QuarticEaseIn">
                  <KeyFrame Cue="0%">
                        <Setter Property="Margin" Value="0"/>
                  </KeyFrame>
                  <KeyFrame Cue="100%">
                        <Setter Property="Margin" Value="200,0,0,0"/>
                  </KeyFrame>
            </Animation>
      </Style.Animations>
</Style>
```

The list of easing functions is very long, and it is not possible to include them all here, so in this case, it is recommended to have a look at the documentation.

## Introducing transformations

In Avalonia UI, objects called *render transforms*, commonly referred to as *transformations*, allow you to apply changes to the graphic layout of the visual elements, keeping their behavior unchanged. These transformations allow you to tilt, rotate, resize, and translate the elements of the graphic interface, so that you can obtain extremely interesting graphic effects. Transformations make sense with animations, and the following ones are available:

- **`SkewTransform`**, which creates a three-dimensional effect by tilting an object according to the X and Y coordinates and the specified **`AngleX`** and **`AngleY`** properties.
- **`RotateTransform`**, which allows you to rotate a control by *n* degrees, assigning the **`Angle`** property.
- **`ScaleTransform`**, which allows you to resize a control to the values provided via the **`ScaleX`** and **`ScaleY`** properties.
- **`TranslateTransform`**, which allows you to move a control to the specified **`X`** and **`Y`** coordinates.
- **`MatrixTransform`**, which allows you to define custom transformations.

For example, the following transformation will rotate, via an animation, the target **`TextBlock`** object, changing the angle's value from 0 to 270:

```
<Style Selector="TextBlock.red">
      <Setter Property="Foreground" Value="Red"/>
      <Style.Animations>
            <Animation Duration="0:0:15">
                  <KeyFrame Cue="0%">
                        <Setter Property="RotateTransform.Angle" Value="0"/>
                  </KeyFrame>
                  <KeyFrame Cue="100%">
                        <Setter Property="RotateTransform.Angle"
```

```
                              Value="270"/>
                </KeyFrame>
            </Animation>
      </Style.Animations>
</Style>
```

In terms of syntax, you specify the transformation name followed by the property name (in the previous code, **`RotateTransform.Angle`**). Trying all the other transformations is left to you as an exercise. It is worth mentioning that you can combine multiple transformations into one animation, so you can really create complex and interesting effects.

## Introducing transitions

While animations change property values based on the specified conditions, *transitions* listen to changes over the specified properties, and when their value changes, a transition effect is applied to that property. Transitions work at the **`Setter`** level, as you can see in the following code:

```
<Style Selector="TextBlock.red">
      <Setter Property="Foreground" Value="Red"/>
      <Setter Property="Transitions">
            <Transitions>
                  <ThicknessTransition Property="Margin" Duration="0:0:2"/>
            </Transitions>
      </Setter>
</Style>
```

When the **`Margin`** property of the target control changes, a transition of two seconds is applied. In this case, you use the **`ThicknessTransition`** object, which targets properties of type **`Thickness`**. Table 15 summarizes the available transitions.

*Table 15: Available transitions*

| | |
|---|---|
| **`BoxShadowsTransition`** | For **`BoxShadows`** properties |
| **`BrushTransition`** | For properties of type **`IBrush`** |
| **`ColorTransition`** | For properties of type **`Color`** |
| **`CornerRadiusTransition`** | For **`CornerRadius`** properties |
| **`DoubleTransitions`** | For properties of type **`double`** |

| | |
|---|---|
| **`FloatTransitions`** | For properties of type **`float`** |
| **`IntegerTransitions`** | For properties of type **`int`** |
| **`PointTransition`** | For properties of type **`Point`** |
| **`SizeTransition`** | For properties of type **`Size`** |
| **`ThicknessTransition`** | For properties of type **`Thickness`** |
| **`TransformOperationsTransition`** | For properties of type **`ITransform`** |
| **`VectorTransition`** | For properties of type **`Vector`** |

You can combine multiple transitions into one style, so you have great control over the effects that you can get.

## Chapter summary

In this chapter, you have seen another important part of the Avalonia UI framework, related to 2D graphics. You have seen how to fill objects with brushes, how to create drawings using shapes, and how they can also be assigned to the **`Content`** property of content controls for a richer user interface.

Finally, you have seen how to implement animations and transformations, leveraging the power of the Avalonia UI graphic engine based on Skia. You have now gained all the necessary knowledge to build applications with Avalonia UI, leveraging all the framework and tooling productivity.

# Conclusion

Avalonia UI offers everything you need to create powerful, native applications that target desktop and mobile systems, plus the web browser. The purpose of this book is to provide guidance on how to get the most out of Avalonia UI to implement common requirements in modern apps, from user experience requirements to runtime requirements.

You have seen how to set up your development machine to work with Microsoft Visual Studio 2022, and how to organize windows with panels. You have seen how to use the Avalonia UI library to satisfy basic requirements in a cross-platform approach, such as using built-in controls and implementing (or restyling) new controls. You learned how to leverage one of the most powerful features of the Avalonia UI framework, data binding, to display and update data. You have seen how to enrich the user interface with brushes, geometries, animations, and transitions. You have also seen how to implement navigation between pages, something you need to do especially with mobile apps, and how to implement complex architectures based on the MVVM pattern.

Because sharing knowledge is the key to technical growth for every developer, I am happy to receive feedback on the book and about the techniques described here. If reading these chapters makes you think about more efficient solutions that you want to share, that would be beneficial to the entire developer community.