Avalonia Book

Contents

1. Welcome to Avalonia and MVVM

Goal - Understand what Avalonia is and why you might choose it. - Learn the simple meanings of C#, XAML, and MVVM. - Get a mental map of how an Avalonia app fits together. - Know where to find things in the source code and samples.

Why this matters - UI development is easier when you understand the main pieces. Avalonia uses C# for logic and XAML for UI markup. MVVM is the pattern that keeps your code clean and testable. Once you know these pieces, the rest of the book will feel natural.

What is Avalonia (in simple words) - Avalonia is a cross-platform UI framework. You write your app once, and run it on Windows, macOS, Linux, Android, iOS, and in the browser (WebAssembly). - It is open source. It looks and feels modern. It has a wide set of controls, a Fluent theme, strong data binding, and great tooling. - You use C# for code and XAML for the UI description. If you know WPF, you will feel at home. If you are new, you will learn with gentle steps.

Platforms you can target - Desktop: Windows, macOS, Linux - Mobile: Android, iOS - Browser: WebAssembly (WASM)

What are C#, XAML, and MVVM - C# (say "see sharp") is the programming language you use for logic: data, commands, navigation, services, tests. - XAML is a simple markup language for UI: you describe windows, pages, controls, layouts, and styles using readable tags. - MVVM is a way to organize code: - Model: your data and core rules. - ViewModel: the "middle" object that exposes properties and commands for the UI. - View: the XAML that shows things on screen and binds to the ViewModel. The View contains no business rules.

How an Avalonia app is shaped - App: a class that sets up your application (themes, resources, startup window). - Views: XAML files that describe what users see (windows, pages, dialogs). - ViewModels: C# classes that provide data and commands to the Views. - Controls: ready-made UI building blocks like Button, TextBox, DataGrid. - Styles and theme: define how the app looks (colors, spacing, typography). - Startup and lifetime: how the app starts and closes on each platform.

A simple mental picture - App starts → sets up theme and services → opens a Window (View) → the View binds to a ViewModel → the ViewModel talks to Models/services → the UI updates automatically through bindings.

Repo and samples in this project - Framework source (read-only for learning): src - Samples (you can run and explore): samples - Docs (build and contributor notes): docs

Look inside (optional, just to get familiar) - Controls live in: src/Avalonia.Controls - Fluent theme: src/Avalonia.Themes.Fluent - Rendering (Skia backend): src/Skia/Avalonia.Skia - ReactiveUI integration: src/Avalonia.ReactiveUI - Browser target: src/Browser - Desktop helpers: src/Avalonia.Desktop

Your first "tour" (no coding yet) 1) Open the samples folder at samples. 2) Skim ControlCatalog projects (Desktop, Android, Browser, iOS) — see samples/ControlCatalog. These show most controls and styles. You don't need to understand the code yet. Just remember: if you wonder "How does Button work?", the Control Catalog shows it in action. 3) Skim samples/BindingDemo and samples/ReactiveUIDemo. These show how data binding and MVVM feel.

MVVM in plain English - MVVM is about separation. The View is just visuals. The ViewModel exposes data (properties) and actions (commands). The Model holds your real data and rules. This separation makes testing easier and code easier to change. - Example idea: A Counter app - ViewModel has a property Count and a command Increment. - View shows Count in a TextBlock and binds a Button to Increment. - When you click the Button, the ViewModel changes Count and the UI updates automatically.

About XAML (don't worry, it's friendly) - XAML uses angle-bracket tags like HTML, but it describes native controls, layout, and styles. - You can nest panels and controls, and use attributes to set properties (like Width, Margin, Text, Items, etc.). - With data binding, you connect XAML properties to ViewModel properties by name.

About data binding (a tiny preview) - Binding is a link between a View property and a ViewModel property. - If the ViewModel changes, the UI updates. If the user changes a control (like typing in a TextBox), the ViewModel can update too (depending on the binding mode). - We will cover binding fully in Chapter 8.

Design and theming - Avalonia ships with a Fluent theme that looks modern. - You can tweak colors, spacing, corner radius, and styles. - You can define reusable resources (colors, brushes, styles) and use them across the app.

Where "startup" happens (a gentle hint only) - An Avalonia app configures itself in a builder (AppBuilder) and chooses a lifetime (desktop with windows, or single-view for mobile). - You'll meet AppBuilder and lifetimes in Chapter 4. For now, just remember: that's where the app decides what it runs on and how it opens windows.

Tooling you'll meet later - DevTools: inspect the visual tree and properties at runtime. - XAML Previewer: see your UI as you type (in supported IDEs). - Headless: run UI logic without a window for special testing scenarios.

Check yourself - Can you explain in one sentence what Avalonia is? - Can you name the three MVVM parts and what each one does? - Do you know the difference between C# and XAML in an Avalonia app? - Can you point to where controls and themes live in the repo?

Quick glossary - App: the application entry point that configures theme, resources, and startup. - View: the UI (XAML) that users see, such as a Window or a Page. - ViewModel: the C# class the View binds to (data + commands, no UI code). - Model: your domain data and rules. - Binding: the connection between a View property and a ViewModel property. - Command: an action you call from UI (e.g., when a Button is clicked).

Extra practice - Explore the ControlCatalog in samples/ControlCatalog (pick the Desktop one if unsure). Open it in your IDE, build, and run. Click around to see many controls and styles. - Open samples/BindingDemo. Look for a binding in XAML and try to guess which ViewModel property it uses. You don't need to change anything yet.

What's next - Next: Chapter 2

2. Set up tools and build your first project

Goal - Install the tools you need. - Create a new Avalonia app from a template. - Build and run it on your machine. - Understand the key files that were generated.

What you need (pick one IDE) - Visual Studio (Windows) with ".NET desktop development" workload - JetBrains Rider (Windows/macOS/Linux) - Visual Studio Code (Windows/macOS/Linux) + C# extension

Also needed - .NET SDK installed (use the latest stable SDK). If the "dotnet" command works in your terminal, you're good.

Install Avalonia project templates - This gives you "dotnet new" templates for Avalonia projects.

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

Create your first app - We'll use the basic desktop app template.

```
# Create a new folder with a project inside
dotnet new avalonia.app -o HelloAvalonia

# Go into the project folder
cd HelloAvalonia
```

Build and run - These commands work on Windows, macOS, and Linux.

```
# Restore packages and build
dotnet build

# Run the app
dotnet run
```

You should see a window open with a simple UI. Close it when you are done.

Open the project in your IDE - Open the HelloAvalonia folder in your IDE. - You can also press the "Run" button in the IDE instead of using the terminal.

A quick tour of the files - HelloAvalonia.csproj: the project file. It lists NuGet packages and target frameworks. - Program.cs: the entry point. It configures Avalonia and starts the app. - App.axaml and App.axaml.cs: application resources and startup setup. - MainWindow.axaml and MainWindow.axaml.cs: your first window (the main View) and its code-behind.

Peek inside Program.cs (what it does) - It creates an AppBuilder, sets up the platform and renderer, and starts a desktop lifetime with a main window. - We will learn AppBuilder and lifetimes in Chapter 4. For now, just know: this is where the app starts.

Make a tiny change (to see it's real) - Open MainWindow.axaml. - Change the Title attribute and add a TextBlock inside the layout.

Example

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="HelloAvalonia.MainWindow"
        Title="Hello Avalonia!">
  <StackPanel Margin="16">
    <TextBlock Text="It works!" FontSize="24"/>
    <Button Content="Click me" Margin="0,12,0,0"/>
  </StackPanel>
</Window>
```

Run again

```
dotnet run
```

- You should see the new title and the text/button you added.

Common template choices (for later) - avalonia.app: minimal app, code-behind pattern (what we used). - avalonia.mvvm: MVVM-friendly skeleton without ReactiveUI. - avalonia.reactiveui: MVVM with ReactiveUI helpers.

Troubleshooting - “dotnet” not found: install the .NET SDK and restart your terminal/IDE. - Restore/build errors: run “dotnet restore” then “dotnet build” to see details. - Window doesn’t show: ensure you ran from inside the project folder; check the terminal output for errors.

Check yourself - Can you create a new Avalonia app with one command? - Do you know what Program.cs and App.axaml are responsible for? - Can you change a window title in XAML and see the change when you run?

Extra practice - Add another TextBlock and change its FontSize and Foreground. - Add a second Button. Try changing Margin and Padding to see layout effects.

What’s next - Next: Chapter 3

3. Your first UI: layouts, controls, and XAML basics

Goal - Build your first real window using common controls and two core layout panels. - Learn the XAML you'll write most often (attributes, nesting, simple resources). - Run, resize, and understand how layout adapts.

What you'll build - A window with a title, some text, a button that updates text, and a simple form laid out with Grid. - You'll see how StackPanel and Grid work together and how controls size themselves.

Prerequisites - You've completed Chapter 2 and can create and run a new Avalonia app.

Step-by-step 1) Create a new app - In a terminal: dotnet new avalonia.app -o HelloLayouts - Open the project in your IDE, then run it (dotnet run) to verify it starts.

2) Replace MainWindow content with basic UI

- Open MainWindow.axaml and replace the inner content of with this:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="HelloLayouts.MainWindow"
        Width="500" Height="360"
        Title="Hello, Avalonia!">
  <StackPanel Margin="16" Spacing="12">
    <TextBlock Classes="h1" Text="Your first UI"/>

    <TextBlock x:Name="CounterText" Text="You clicked 0 times."/>
    <Button x:Name="CounterButton"
            Width="140"
            Content="Click me"
            Click="CounterButton_OnClick"/>

    <Border Background="{DynamicResource ThemeAccentBrush}"
            CornerRadius="6" Padding="12">
      <TextBlock Foreground="White" Text="Inside a Border"/>
    </Border>

    <Grid ColumnDefinitions="Auto,*" RowDefinitions="Auto,Auto" Margin="0,8,0,0">
      <TextBlock Text="Name:" Margin="0,0,8,8"/>
      <TextBox Grid.Column="1" Width="240"/>

      <TextBlock Grid.Row="1" Text="Email:" Margin="0,0,8,0"/>
      <TextBox Grid.Row="1" Grid.Column="1" Width="240"/>
    </Grid>
  </StackPanel>
</Window>
```

- Save. Your previewer (if enabled in your IDE) should refresh. Otherwise, run the app to see the layout.

3) Wire up a simple event in code-behind

- Open MainWindow.axaml.cs and add this method and field:

```
using Avalonia.Controls;       // for TextBlock, Button
using Avalonia.Interactivity;  // for RoutedEventArgs

private int _count;
private TextBlock? _counterText;
```

```
public MainWindow()
{
    InitializeComponent();
    _counterText = this.FindControl<TextBlock>("CounterText");
}

private void CounterButton_OnClick(object? sender, RoutedEventArgs e)
{
    _count++;
    if (_counterText is not null)
        _counterText.Text = $"You clicked {_count} times.";
}
```

- Build and run. Click the button—your text updates. This is a tiny taste of events; MVVM and bindings come later.

4) XAML basics you just used

- Nesting: Panels (like StackPanel and Grid) contain other controls.
- Attributes: Properties like Margin, Spacing, Width are set as attributes.
- Attached properties: Grid.Row and Grid.Column are attached properties that apply to children inside a Grid.
- Resources: {DynamicResource ThemeAccentBrush} pulls a color from the current theme.

5) Layout in plain words

- StackPanel lays out children in a single line (vertical by default) and gives each its desired size.
- Grid gives you rows and columns. Use Auto for "size to content," * for "take the rest," and numbers like 2* for proportional sizing.
- Most controls size to their content by default. Add Margin for space around, and Padding for space inside containers.

6) Run and resize

- Resize the window. Notice TextBox stretches in the Grid's second column while labels stay Auto-sized in the first column.

Check yourself - Can you add another row to the Grid for a "Phone" field? - Can you put the button above the Border by moving it earlier in the StackPanel? - Can you make the button stretch horizontally (set HorizontalAlignment="Stretch")? - Do Tab key presses move focus between fields in the expected order?

Look under the hood (optional) - Controls live here: Avalonia.Controls - Themes (Fluent): Avalonia.Themes.Fluent - Explore the ControlCatalog sample: samples/ControlCatalog

Extra practice - Add a DockPanel with a top bar (a TextBlock) and the rest filled with your StackPanel using DockPanel.Dock. - Replace StackPanel with a Grid-only layout: two rows (title on top, content below), and columns inside the content row. - Try WrapPanel for a row of buttons that wrap on small widths.

`If the Previewer isn't available in your IDE, just build and run often. Fast feedback is what matters.`

What's next - Next: Chapter 4

4. Application startup: AppBuilder and lifetimes

Goal - Understand how an Avalonia app starts and what AppBuilder configures. - Learn the difference between desktop (multi-window) and single-view lifetimes. - Hook up MainWindow for desktop and MainView for single-view.

Why this matters - Startup and lifetimes decide how your app is created and how windows/views are shown. - Getting this right early saves confusion when you target desktop, mobile, and browser later.

Prerequisites - You can create and run a new Avalonia app (from Chapter 2).

Mental model - AppBuilder configures Avalonia (platform backend, renderer, logging, optional libraries like ReactiveUI). - A lifetime drives the app's main loop and surface(s): - ClassicDesktopStyleApplicationLifetime = desktop windowed apps (Windows/macOS/Linux). - SingleViewApplicationLifetime = single-view apps (Android/iOS/Browser, often one root view).

Step-by-step 1) Inspect Program.cs - Open Program.cs in a new app and you'll usually see:

```
using Avalonia;
using System;

class Program
{
    public static void Main(string[] args) => BuildAvaloniaApp()
        .StartWithClassicDesktopLifetime(args); // Desktop lifetime entry

    public static AppBuilder BuildAvaloniaApp()
        => AppBuilder.Configure<App>()
            .UsePlatformDetect()   // Pick platform backends (Win32, macOS, X11, Android, iOS, Browser)
            .UseSkia()             // Use Skia renderer (CPU/GPU)
            .LogToTrace();
}
```

- UsePlatformDetect chooses the right platform backends at runtime.
- UseSkia selects Skia as the rendering engine.
- StartWithClassicDesktopLifetime wires the desktop-specific lifetime and passes command-line args.

2) Wire windows/views in App.axaml.cs

- Open App.axaml.cs and find OnFrameworkInitializationCompleted. Make sure it handles both lifetimes:

```
using Avalonia;
using Avalonia.Controls.ApplicationLifetimes;

public partial class App : Application
{
    public override void OnFrameworkInitializationCompleted()
    {
        if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
        {
            desktop.MainWindow = new MainWindow(); // Your primary window for desktop apps
        }
        else if (ApplicationLifetime is ISingleViewApplicationLifetime singleView)
        {
            singleView.MainView = new MainView(); // Your root view for mobile/browser
        }

        base.OnFrameworkInitializationCompleted();
```

```
    }
}
```

- Desktop lifetime expects a window; single-view expects a single root control.
- You can share most app code and choose at runtime based on the actual lifetime.

3) Try single-view with a basic MainView

- Add a view (MainView.axaml) with a simple layout and set singleView.MainView = new MainView().
- Running on desktop still uses the desktop lifetime (MainWindow). On mobile/browser targets, the single-view branch is used.

4) Options and add-ons

- Logging: .LogToTrace() is helpful while developing.
- ReactiveUI: add .UseReactiveUI() when you adopt ReactiveUI in later chapters.
- Renderer and platform options can be customized later (e.g., Skia options), but defaults work well to start.

Check yourself - Can you explain what BuildAvaloniaApp() returns and where it's used? - Where would you put code that must run before showing the first window/view? - What changes between desktop and single-view in OnFrameworkInitializationCompleted?

Look under the hood (optional) - AppBuilder desktop helpers: Avalonia.Desktop/AppBuilderDesktopExtensions.cs - Classic desktop lifetime: Avalonia.Controls/ApplicationLifetimes/ClassicDesktopStyleApplicationLifetime.cs - Single-view lifetime interface: Avalonia.Controls/ApplicationLifetimes/ISingleViewApplicationLifetime.cs

Extra practice - Add a command-line flag (e.g., –no-main) and skip creating MainWindow when present. - For single-view, replace MainView with a view containing a TabControl and confirm it appears on mobile/browser. - Log which lifetime branch executed at startup and verify on different targets.

`If your app doesn't start on desktop, confirm you're calling StartWithClassicDesktopLifetime(args) in Ma`

What's next - Next: Chapter 5

5. Layout system without mystery

Goal - Understand how Avalonia's layout works in plain words (measure and arrange). - Get comfortable with the core panels: StackPanel, Grid, DockPanel, WrapPanel. - Learn practical sizing: star sizing, Auto, alignment, Margin vs Padding, Min/Max.

Why this matters - Layout is the backbone of every UI; once clear, everything else is simpler. - A small set of panels covers most real apps — you'll combine them confidently.

Prerequisites - You can run a basic Avalonia app and edit MainWindow.axaml (Chapters 2–3).

Mental model (no jargon) - Parents lay out children. First, measure: the parent asks each child, "how big would you like to be within this space?" Then, arrange: the parent gives each child a final rectangle to live in. - Content sizes to its content by default. Alignment controls how leftover space is used. - Grid gives structure; StackPanel flows; DockPanel pins to edges; WrapPanel flows and wraps.

Step-by-step layout tour 1) Start a fresh page - Open MainWindow.axaml and replace the inner content of with:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="LayoutPlayground.MainWindow"
        Width="800" Height="520"
        Title="Layout Playground">
  <Grid ColumnDefinitions="*,*" RowDefinitions="Auto,*" Padding="16" RowSpacing="12" ColumnSpacing="12"
    <TextBlock Grid.ColumnSpan="2" Classes="h1" Text="Layout system without mystery"/>

    <!-- Left: StackPanel + DockPanel samples -->
    <StackPanel Grid.Row="1" Spacing="8">
      <TextBlock Classes="h2" Text="StackPanel"/>
      <Border BorderBrush="#CCC" BorderThickness="1" Padding="8">
        <StackPanel Spacing="6">
          <Button Content="Top"/>
          <Button Content="Middle"/>
          <Button Content="Bottom"/>
          <!-- Stretch the next one across -->
          <Button Content="Stretch me" HorizontalAlignment="Stretch"/>
        </StackPanel>
      </Border>

      <TextBlock Classes="h2" Text="DockPanel"/>
      <Border BorderBrush="#CCC" BorderThickness="1" Padding="8">
        <DockPanel LastChildFill="True">
          <TextBlock DockPanel.Dock="Top" Text="Top bar"/>
          <TextBlock DockPanel.Dock="Left" Text="Left" Margin="0,4,8,0"/>
          <Border Background="#F0F6FF" CornerRadius="4" Padding="8">
            <TextBlock Text="The last child fills the remaining space"/>
          </Border>
        </DockPanel>
      </Border>
    </StackPanel>

    <!-- Right: Grid + WrapPanel samples -->
    <StackPanel Grid.Column="1" Grid.Row="1" Spacing="8">
      <TextBlock Classes="h2" Text="Grid"/>
      <Border BorderBrush="#CCC" BorderThickness="1" Padding="8">
        <!-- 2 columns: label column Auto sizes to content, value column takes the rest (*) -->
```

```
        <Grid ColumnDefinitions="Auto,*" RowDefinitions="Auto,Auto,Auto" ColumnSpacing="8" RowSpacing="8
          <TextBlock Text="Name:"/>
          <TextBox Grid.Column="1"/>

          <TextBlock Grid.Row="1" Text="Email:"/>
          <TextBox Grid.Row="1" Grid.Column="1"/>

          <!-- Proportional star sizing example: make a tall row using 2* vs * -->
          <TextBlock Grid.Row="2" Text="About:" VerticalAlignment="Top"/>
          <TextBox Grid.Row="2" Grid.Column="1" Height="80" AcceptsReturn="True"/>
        </Grid>
      </Border>

      <TextBlock Classes="h2" Text="WrapPanel"/>
      <Border BorderBrush="#CCC" BorderThickness="1" Padding="8">
        <WrapPanel ItemWidth="100" ItemHeight="32" HorizontalAlignment="Left">
          <Button Content="One"/>
          <Button Content="Two"/>
          <Button Content="Three"/>
          <Button Content="Four"/>
          <Button Content="Five"/>
          <Button Content="Six"/>
        </WrapPanel>
      </Border>
    </StackPanel>
  </Grid>
</Window>
```

- Run and resize the window. Watch StackPanel keep items in a column, DockPanel pin edges and let the last child fill, Grid align labels and stretch inputs, and WrapPanel wrap buttons to new rows when needed.

2) Alignment and sizing toolkit

- Margin vs Padding: Margin adds space outside a control; Padding adds space inside containers like Border.
- HorizontalAlignment/VerticalAlignment: Start, Center, End, Stretch. Most inputs stretch in Grid's star column.
- Width/Height vs Min/Max: Prefer Min/Max to keep flexibility; fixed sizes can fight responsiveness.
- In Grid, Auto means "size to content." * means "take remaining space." 2* means "take twice the share."

3) Common patterns

- Forms: Grid with Auto label column and * input column; use RowSpacing/ColumnSpacing for clean gaps.
- Toolbars: DockPanel with Top bar and LastChildFill content; or a Grid with star rows.
- Responsive groups: WrapPanel for chips/tags/buttons that naturally reflow.

Check yourself - Can you swap the left/right columns by changing Grid.Column on the StackPanels? - Can you add a third Grid column for an "Edit" button and keep inputs stretching? - Can you make the DockPanel's Left area wider by wrapping it in a Border with Width set? - Can you make WrapPanel items equal width using ItemWidth, and let text wrap inside a Button?

Look under the hood (optional) - Controls and panels live here: Avalonia.Controls - Explore layout samples in ControlCatalog: samples/ControlCatalog - Styling and theme resources used by controls: Avalonia.Themes.Fluent

Extra practice - Rebuild the entire right-hand column using only Grid (no nested StackPanels). - Create a two-pane layout: DockPanel with a fixed left navigation (Width=220) and content filling the rest. - Add MinWidth to text inputs to keep them readable when the window gets small.

```
When layouts get tricky, add Borders with different background colors briefly to see the rectangles eac
```

What's next - Next: Chapter 6

6. Controls tour you'll actually use

Goal - Get comfortable with everyday controls: TextBlock/TextBox, Button, CheckBox/RadioButton, ComboBox, ListBox, Slider, ProgressBar, DatePicker, and more. - Learn simple properties that make them useful right away. - Explore ControlCatalog to see each control in action.

Why this matters - You'll use these controls constantly; knowing their basics speeds up every screen you build.

Prerequisites - You've built small UIs from Chapters 3–5 and can run your app.

Quick tour by example 1) Text and inputs

```
<StackPanel Spacing="8">
  <TextBlock Classes="h1" Text="Controls tour"/>
  <TextBlock Text="TextBlock shows read-only text"/>
  <TextBox Watermark="Type here..."/>
  <PasswordBox PasswordChar="•"/>
</StackPanel>
```

- TextBlock is for read-only text; TextBox for text input. Watermark shows a hint when empty.
- PasswordBox masks input. Use bindings later for MVVM.

2) Buttons and toggles

```
<StackPanel Orientation="Horizontal" Spacing="8">
  <Button Content="Primary"/>
  <ToggleButton Content="Toggle me"/>
  <CheckBox Content="I agree"/>
  <StackPanel DataContext="{x:Static Enum:MyEnum}">
    <!-- RadioButtons are typically grouped by container and bound to a value -->
    <RadioButton Content="Option A" GroupName="Choice"/>
    <RadioButton Content="Option B" GroupName="Choice"/>
  </StackPanel>
</StackPanel>
```

- ToggleButton stays pressed when toggled. CheckBox is on/off. RadioButtons are mutually exclusive per GroupName.

3) Choices and lists

```
<StackPanel Spacing="8">
  <ComboBox PlaceholderText="Pick one">
    <ComboBoxItem Content="Red"/>
    <ComboBoxItem Content="Green"/>
    <ComboBoxItem Content="Blue"/>
  </ComboBox>

  <ListBox Height="120">
    <ListBoxItem Content="Item 1"/>
    <ListBoxItem Content="Item 2"/>
    <ListBoxItem Content="Item 3"/>
  </ListBox>
</StackPanel>
```

- ComboBox renders a dropdown; ListBox renders a vertical list. Later you'll use ItemsSource and DataTemplates for real data.

4) Sliders, progress, and pickers

```
<StackPanel Spacing="8" Orientation="Horizontal" VerticalAlignment="Center">
  <TextBlock Text="Volume" Margin="0,0,8,0"/>
  <Slider Width="180" Minimum="0" Maximum="100" Value="50"/>
  <ProgressBar Width="160" IsIndeterminate="True"/>
</StackPanel>

<DatePicker SelectedDate="2025-01-01"/>
```

- Slider is great for numeric ranges; ProgressBar shows progress or an indeterminate animation; DatePicker picks dates.

5) Menus, tooltips, and context menus

```
<DockPanel>
  <Menu DockPanel.Dock="Top">
    <MenuItem Header="File">
      <MenuItem Header="New"/>
      <MenuItem Header="Open"/>
    </MenuItem>
    <MenuItem Header="Help">
      <MenuItem Header="About"/>
    </MenuItem>
  </Menu>

  <Button Content="Right-click me" HorizontalAlignment="Center" VerticalAlignment="Center">
    <Button.ContextMenu>
      <ContextMenu>
        <MenuItem Header="Copy"/>
        <MenuItem Header="Paste"/>
      </ContextMenu>
    </Button.ContextMenu>
    <ToolTip.Tip>
      <ToolTip Content="I am a tooltip"/>
    </ToolTip.Tip>
  </Button>
</DockPanel>
```

- Menu sits at the top; ContextMenu opens on right-click; ToolTip appears on hover.

ControlCatalog tour - Run the ControlCatalog sample to explore each control's options and templates: samples/ControlCatalog - Look at XAML and toggle options to see how properties change behavior.

Check yourself - Can you replace the ListBox with a ListBox bound to a simple list of strings (you'll need ItemsSource and DataTemplate)? - Can you add hotkeys to menu items (e.g., Header="_File" for mnemonic, InputGesture) and verify they work? - Can you wire a Button Click handler to show a dialog (simple MessageBox from your IDE or custom Window)?

Look under the hood (optional) - Controls live here: Avalonia.Controls - Fluent theme styles: Avalonia.Themes.Fluent - Input events and commands appear in later chapters; peek at RoutedCommand support in ControlCatalog.

Extra practice - Build a small "settings" page with TextBox, CheckBox, ComboBox, and a Save button. - Add a Slider that updates a TextBlock as you move it. - Give the Button a ContextMenu and a ToolTip with helpful hints.

`Don't memorize every property - learn patterns. ControlCatalog is your friend when you need to explore.`

What's next - Next: Chapter 7

7. Fluent theming and styles made simple

Goal - Understand Avalonia's Fluent theme, light/dark variants, resources, and styles. - Learn how to create global and local styles, use StaticResource vs DynamicResource, and switch themes at runtime.

What you'll build - A small app that: - Uses Fluent theme. - Defines shared colors/brushes in resources. - Styles buttons globally and locally (implicit and keyed styles). - Adds a simple theme switch (Light/Dark) at runtime.

Prerequisites - You can run a basic Avalonia app (Ch. 2–4). - You are comfortable editing App.axaml and a Window/UserControl (Ch. 3–6).

1) Meet FluentTheme and theme variants

- Avalonia ships with FluentTheme. Add it to App.axaml if your template didn't already:

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

- RequestedThemeVariant can be Light or Dark on Application, a Window, or any ThemeVariantScope. Start with Light.
- FluentTheme picks resources (brushes, etc.) appropriate for the active theme variant.

2) Resources: colors, brushes, and where to put them

- Resources are key/value objects you can reference from XAML. Put app-wide resources in App.axaml:

```
<Application ...>
  <Application.Resources>
    <SolidColorBrush x:Key="AccentBrush" Color="#2563EB"/>
    <SolidColorBrush x:Key="AccentBrushHover" Color="#1D4ED8"/>
    <Thickness x:Key="ControlCornerRadius">6</Thickness>
  </Application.Resources>
  <Application.Styles>
    <FluentTheme />
  </Application.Styles>
</Application>
```

- Referencing resources:
  - StaticResource resolves once at load time (faster): Background="{StaticResource AccentBrush}"
  - DynamicResource updates if the resource changes at runtime: Background="{DynamicResource AccentBrush}"
- Resource lookup walks upward: control → parent → Window → Application. App resources are global.

3) Global styles (implicit) vs local styles (keyed)

- A style sets properties for a target control. Put global styles in Application.Styles:

```
<Application.Styles>
  <FluentTheme />
  <!-- Implicit (applies to all Buttons) -->
  <Style Selector="Button">
    <Setter Property="CornerRadius" Value="{StaticResource ControlCornerRadius}"/>
    <Setter Property="Padding" Value="12,8"/>
  </Style>
```

```
  <!-- Hover visual (pseudo-class) -->
  <Style Selector="Button:pointerover">
    <Setter Property="Background" Value="{DynamicResource AccentBrushHover}"/>
  </Style>
</Application.Styles>
```

- A local, keyed style only applies when you opt in:

```
<StackPanel>
  <StackPanel.Resources>
    <Style x:Key="PrimaryButtonStyle" Selector="Button">
      <Setter Property="Background" Value="{DynamicResource AccentBrush}"/>
      <Setter Property="Foreground" Value="White"/>
      <Setter Property="FontWeight" Value="SemiBold"/>
    </Style>
  </StackPanel.Resources>

  <Button Content="OK" Classes="primary" Style="{StaticResource PrimaryButtonStyle}"/>
  <Button Content="Cancel"/>
</StackPanel>
```

- Tip: You can combine implicit styles (for consistent baselines) with keyed styles (for special cases).

4) Selectors and pseudo-classes you'll actually use

- Selector="Button" targets all Buttons.
- You can target by name (#[Name]), class (.danger), and state pseudo-classes like :pointerover, :pressed, :disabled, :focus.
- Example:

```
<Application.Styles>
  <Style Selector="Button.danger">
    <Setter Property="Background" Value="#B91C1C"/>
  </Style>
  <Style Selector="Button.danger:pointerover">
    <Setter Property="Background" Value="#991B1B"/>
  </Style>
</Application.Styles>
```

5) Switching Light/Dark at runtime

- You can switch theme variants in code behind. For example, add a ToggleSwitch to your MainView and handle its change:

```
<ToggleSwitch x:Name="ThemeSwitch" Content="Dark mode"/>
```

```
using Avalonia;
using Avalonia.Styling; // ThemeVariant

private void ThemeSwitch_PropertyChanged(object? sender, AvaloniaPropertyChangedEventArgs e)
{
    if (e.Property == ToggleSwitch.IsCheckedProperty)
    {
        var dark = ThemeSwitch.IsChecked == true;
        Application.Current!.RequestedThemeVariant = dark ? ThemeVariant.Dark : ThemeVariant.Light;
    }
}
```

- Hook this handler once in your view's constructor (after InitializeComponent) and subscribe to ThemeSwitch.PropertyChanged.
- Because you used DynamicResource for color brushes, your UI reacts to theme-driven resource changes automatically.
- Scope theme changes: set RequestedThemeVariant on a specific Window or container (ThemeVariantScope) to localize the effect.

6) Organizing styles and resources into files

- Keep App.axaml readable by moving big sections into separate XAML files and merge them:

```
<Application ...>
  <Application.Resources>
    <ResourceInclude Source="avares://MyApp/Styles/Colors.axaml"/>
  </Application.Resources>
  <Application.Styles>
    <FluentTheme />
    <StyleInclude Source="avares://MyApp/Styles/Controls.axaml"/>
  </Application.Styles>
</Application>
```

- Use ResourceInclude for resources and StyleInclude for styles. Each file should have a root ResourceDictionary or Styles element accordingly.

7) StaticResource vs DynamicResource in practice

- Use StaticResource for values that won't change (e.g., Thickness, FontSize constants).
- Use DynamicResource when the value should update at runtime (e.g., theme-dependent brushes, app accent color).

Check yourself - Can you explain the difference between implicit and keyed styles? - Where does Avalonia look for resources when resolving a key? - Which binding updates at runtime: StaticResource or DynamicResource? - How do you switch theme variant from code?

Look under the hood (repo reading list) - Styles and selectors: src/Avalonia.Styling - FluentTheme implementation and resources: src/Avalonia.Themes.Fluent - ThemeVariant enum and theme scoping: src/Avalonia.Styling

Extra practice - Create a secondary accent (e.g., SuccessBrush) and use it for positive actions. - Add a named class (e.g., .danger) and adjust background/foreground/hover/pressed styles. - Split your styles/resources into separate axaml files and include them from App.axaml. - Try setting RequestedThemeVariant on just one panel to create a light "sheet" in a dark window.

What's next - Next: Chapter 8

8. Data binding basics you'll use every day

Goal - Understand DataContext and how bindings connect your UI to a view model. - Use binding modes (OneWay, TwoWay, OneTime) and element-to-element bindings. - Bind lists with ItemsControl/ListBox, track SelectedItem, and display with DataTemplates. - Create and use a simple value converter; add a minimal validation pattern.

What you'll build - A small view with text fields, a live FullName, a ListBox of people, and a simple converter.

Prerequisites - You can run an Avalonia app and edit XAML/code-behind (Ch. 2–7). - Basic C# (properties, classes, events) and INotifyPropertyChanged familiarity.

1) Set the DataContext

- The DataContext is the source object for most bindings in a view. Set it in code-behind or XAML.
- In XAML (recommended for samples), you can create a view model instance:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:vm="clr-namespace:MyApp.ViewModels"
        x:Class="MyApp.MainWindow">
  <Window.DataContext>
    <vm:MainViewModel />
  </Window.DataContext>

  <!-- content here -->
</Window>
```

- Or in code-behind after InitializeComponent:

```
public MainWindow()
{
    InitializeComponent();
    DataContext = new MainViewModel();
}
```

2) Bind text: OneWay vs TwoWay vs OneTime

- Create a simple view model implementing INotifyPropertyChanged:

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

public class MainViewModel : INotifyPropertyChanged
{
    private string? _firstName;
    private string? _lastName;
    private int _age;

    public string? FirstName
    {
        get => _firstName;
        set { if (_firstName != value) { _firstName = value; OnPropertyChanged(); OnPropertyChanged(nam
    }

    public string? LastName
    {
        get => _lastName;
        set { if (_lastName != value) { _lastName = value; OnPropertyChanged(); OnPropertyChanged(nameo
```

```
    }

    public int Age
    {
        get => _age;
        set { if (_age != value) { _age = value; OnPropertyChanged(); } }
    }

    public string FullName => ($"{FirstName} {LastName}").Trim();

    public event PropertyChangedEventHandler? PropertyChanged;
    protected void OnPropertyChanged([CallerMemberName] string? name = null)
        => PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
}
```

- In XAML, TwoWay binding lets TextBox push changes back to the view model as you type:

```
<StackPanel Spacing="8">
  <TextBox Watermark="First name" Text="{Binding FirstName, Mode=TwoWay}"/>
  <TextBox Watermark="Last name"  Text="{Binding LastName, Mode=TwoWay}"/>

  <!-- OneWay: computed value from VM to UI -->
  <TextBlock Text="{Binding FullName, Mode=OneWay}"/>

  <!-- Age as number; binding still TwoWay -->
  <NumericUpDown Value="{Binding Age, Mode=TwoWay}" Minimum="0" Maximum="130"/>
</StackPanel>
```

- OneTime binds once at load; useful for constants. OneWay updates from VM to UI. TwoWay updates both ways.

3) Bind to another control (element-to-element)

- Sometimes the source isn't the DataContext but another control:

```
<StackPanel Spacing="8">
  <Slider x:Name="S" Minimum="0" Maximum="100"/>
  <ProgressBar Minimum="0" Maximum="100" Value="{Binding #S.Value}"/>
</StackPanel>
```

- The #S syntax binds to the named element S's Value.

4) Bind lists with ItemsControl and ListBox

- Add a People collection and SelectedPerson to the view model:

```
using System.Collections.ObjectModel;

public class Person { public string Name { get; set; } = ""; public int Age { get; set; } }

public class MainViewModel : INotifyPropertyChanged
{
    // ... previous properties ...
    public ObservableCollection<Person> People { get; } = new()
    {
        new Person { Name = "Ada", Age = 28 },
        new Person { Name = "Linus", Age = 32 },
        new Person { Name = "Grace", Age = 45 },
    };
```

```
    private Person? _selectedPerson;
    public Person? SelectedPerson
    {
        get => _selectedPerson;
        set { if (_selectedPerson != value) { _selectedPerson = value; OnPropertyChanged(); } }
    }
}
```

- Bind ItemsSource and SelectedItem in XAML. Use a simple DataTemplate to display each person:

```
<DockPanel LastChildFill="True" Margin="0,8,0,0">
  <ListBox DockPanel.Dock="Left"
           Width="160"
           ItemsSource="{Binding People}"
           SelectedItem="{Binding SelectedPerson, Mode=TwoWay}">
    <ListBox.ItemTemplate>
      <DataTemplate>
        <TextBlock Text="{Binding Name}"/>
      </DataTemplate>
    </ListBox.ItemTemplate>
  </ListBox>

  <StackPanel Margin="12,0,0,0" Spacing="8">
    <TextBlock Text="Details" FontWeight="SemiBold"/>
    <TextBlock Text="{Binding SelectedPerson.Name}"/>
    <TextBlock Text="{Binding SelectedPerson.Age}"/>
  </StackPanel>
</DockPanel>
```

- Bindings can traverse properties: SelectedPerson.Name reads from the current DataContext's SelectedPerson.

5) A simple value converter

- Converters translate data from the source to the target type. Create one that maps an age to a category:

```
using System;
using Avalonia.Data.Converters;
using System.Globalization;

public class AgeCategoryConverter : IValueConverter
{
    public object? Convert(object? value, Type targetType, object? parameter, CultureInfo culture)
        => value is int age ? (age >= 18 ? "Adult" : "Minor") : null;

    public object? ConvertBack(object? value, Type targetType, object? parameter, CultureInfo culture)
        => throw new NotSupportedException();
}
```

- Register and use it in XAML:

```
<Window ... xmlns:conv="clr-namespace:MyApp.Converters">
  <Window.Resources>
    <conv:AgeCategoryConverter x:Key="AgeToCategory"/>
  </Window.Resources>
  <StackPanel>
```

```
    <TextBlock Text="{Binding Age, Converter={StaticResource AgeToCategory}}"/>
  </StackPanel>
</Window>
```

6) Minimal validation pattern

- There are several validation approaches. Here's a simple one you can use immediately: expose an Error string and set it when input is invalid.

```
private string? _ageError;
public string? AgeError
{
    get => _ageError;
    set { if (_ageError != value) { _ageError = value; OnPropertyChanged(); } }
}

public int Age
{
    get => _age;
    set
    {
        if (_age != value)
        {
            _age = value;
            AgeError = _age < 0 || _age > 130 ? "Age must be 0-130" : null;
            OnPropertyChanged();
        }
    }
}
```

- Show the error under the input:

```
<StackPanel Spacing="4">
  <NumericUpDown Value="{Binding Age, Mode=TwoWay}" Minimum="0" Maximum="130"/>
  <TextBlock Foreground="#B91C1C" Text="{Binding AgeError}"/>
</StackPanel>
```

- Later, you can adopt IDataErrorInfo or INotifyDataErrorInfo for richer validation (Chapter 11 touches MVVM patterns).

7) Common binding tips

- Use ObservableCollection for lists you modify at runtime.
- When a property depends on another (FullName depends on FirstName/LastName), raise PropertyChanged for both.
- Prefer TwoWay only when the user edits the value; use OneWay otherwise.
- For performance, avoid excessive ConvertBack when unnecessary.

Check yourself - What does the DataContext do, and where does Avalonia look to resolve a binding? - When do you use TwoWay vs OneWay vs OneTime? - How do you bind to another control's property? - Why do you prefer ObservableCollection over List for ItemsSource?

Look under the hood (repo reading list) - Binding engine and base types: src/Avalonia.Base (Data and Binding) - Controls that display lists: src/Avalonia.Controls (ListBox, ItemsControl)

Extra practice - Add a TextBox to filter People by name and update the ListBox in real time. - Show a selected person editor (edit Name and Age) and ensure TwoWay binds update the list item. - Write a converter that shows "Welcome, {FirstName}!" or a default message when empty.

What's next - Next: Chapter 9

9. Commands, events, and user input

Goal - Understand when to use events vs commands and how they relate to MVVM. - Implement and bind ICommand for Buttons and menu items. - Pass CommandParameter, wire up keyboard shortcuts with KeyBinding, and handle pointer/keyboard events. - Manage focus and common user input patterns.

What you'll build - A small screen that: - Uses commands for Save/Delete actions (with CanExecute logic). - Binds a keyboard shortcut (Ctrl+S) to Save. - Handles a double-click and a pointer press event. - Demonstrates focus and Enter-to-submit behavior.

Prerequisites - You can run an Avalonia app, edit XAML, and create a basic view model (Ch. 2–8).

1) Events vs commands (and why MVVM favors commands)

- Events call methods directly in code-behind (imperative): great for low-level input and one-off UI behaviors.
- Commands expose intent (Save, Delete) on your view model via the ICommand interface: great for testability and re-use.
- Rule of thumb:
  - UI intent → Command (Button.Command, MenuItem.Command, KeyBinding → Command)
  - Low-level input or gestures → Event (PointerPressed, KeyDown, DoubleTapped)

2) Create a simple ICommand implementation (RelayCommand)

- Add a lightweight command class:

```
using System;
using System.Windows.Input;

public class RelayCommand : ICommand
{
    private readonly Action<object?> _execute;
    private readonly Func<object?, bool>? _canExecute;

    public RelayCommand(Action<object?> execute, Func<object?, bool>? canExecute = null)
    {
        _execute = execute ?? throw new ArgumentNullException(nameof(execute));
        _canExecute = canExecute;
    }

    public bool CanExecute(object? parameter) => _canExecute?.Invoke(parameter) ?? true;
    public void Execute(object? parameter) => _execute(parameter);

    public event EventHandler? CanExecuteChanged;
    public void RaiseCanExecuteChanged() => CanExecuteChanged?.Invoke(this, EventArgs.Empty);
}
```

3) Expose commands in your view model

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Windows.Input;

public class MainViewModel : INotifyPropertyChanged
{
    private bool _hasChanges;
    public bool HasChanges
    {
        get => _hasChanges;
```

```
        set { if (_hasChanges != value) { _hasChanges = value; OnPropertyChanged(); (SaveCommand as Rela
    }

    public ICommand SaveCommand { get; }
    public ICommand DeleteCommand { get; }

    public MainViewModel()
    {
        SaveCommand = new RelayCommand(_ => Save(), _ => HasChanges);
        DeleteCommand = new RelayCommand(p => Delete(p));
    }

    private void Save()
    {
        // Persist data...
        HasChanges = false; // Re-evaluate CanExecute
    }

    private void Delete(object? parameter)
    {
        // Use parameter (e.g., currently selected item)
        HasChanges = true;
    }

    public event PropertyChangedEventHandler? PropertyChanged;
    void OnPropertyChanged([CallerMemberName] string? n = null) => PropertyChanged?.Invoke(this, new Pro
}
```

4) Bind Button.Command (and pass CommandParameter)

```
<StackPanel Spacing="8">
  <TextBox Watermark="Type something" Text="{Binding SomeText, Mode=TwoWay}"/>

  <Button Content="Save"
          Command="{Binding SaveCommand}"/>

  <Button Content="Delete selected"
          Command="{Binding DeleteCommand}"
          CommandParameter="{Binding SelectedItem}"/>
</StackPanel>
```

- CommandParameter can pass a selected item, an ID, or any value the command needs.
- Buttons automatically disable when CanExecute returns false.

5) Add keyboard shortcuts with KeyBinding

- Wire Ctrl+S to Save at your Window level so it works anywhere in the view:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="MyApp.MainWindow">
  <Window.InputBindings>
    <KeyBinding Gesture="Ctrl+S" Command="{Binding SaveCommand}"/>
  </Window.InputBindings>

  <!-- content -->
</Window>
```

- You can also set CommandParameter on KeyBinding if your command expects one.

6) Handle common events (low-level input)

- Use events when you need raw input or quick UI reactions.

Double-click on a list:

```
<ListBox x:Name="People" ItemsSource="{Binding People}" DoubleTapped="People_DoubleTapped"/>
```

```
using Avalonia.Interactivity;
using Avalonia.Controls;

private void People_DoubleTapped(object? sender, RoutedEventArgs e)
{
    if (sender is ListBox lb && lb.SelectedItem is not null)
    {
        // Open details, or execute a command with the selected item
        if (DataContext is MainViewModel vm && vm.DeleteCommand.CanExecute(lb.SelectedItem))
            vm.DeleteCommand.Execute(lb.SelectedItem);
    }
}
```

Pointer position inside a control:

```
<Border Background="#EEE" Padding="8" PointerPressed="Border_PointerPressed">
  <TextBlock Text="Click in this area"/>
</Border>
```

```
using Avalonia.Input;
using Avalonia.VisualTree;

private void Border_PointerPressed(object? sender, PointerPressedEventArgs e)
{
    if (sender is IVisual v)
    {
        var p = e.GetPosition(v);
        // Use p.X, p.Y as needed
    }
}
```

Handle Enter key in a TextBox to trigger Save:

```
<TextBox x:Name="Input" KeyDown="Input_KeyDown"/>
```

```
using Avalonia.Input;

private void Input_KeyDown(object? sender, KeyEventArgs e)
{
    if (e.Key == Key.Enter && DataContext is MainViewModel vm)
    {
        if (vm.SaveCommand.CanExecute(null))
            vm.SaveCommand.Execute(null);
        e.Handled = true;
    }
}
```

7) Focus and tab navigation

- Any control with Focusable=“True” can receive focus. Call Focus() from code to move focus.

```
<TextBox x:Name="First"/>
<TextBox x:Name="Second"/>
<Button Content="Focus second" Click="FocusSecond_Click"/>
```

```
private void FocusSecond_Click(object? sender, Avalonia.Interactivity.RoutedEventArgs e)
{
    Second.Focus();
}
```

- Use TabIndex to control tab order. Disable focus for decorative elements with Focusable=“False”.

8) CheckBox and RadioButton patterns

- Toggle inputs still work great with commands:

```
<CheckBox Content="Enable advanced" IsChecked="{Binding IsAdvanced, Mode=TwoWay}"/>
<StackPanel IsEnabled="{Binding IsAdvanced}">
  <!-- advanced controls here -->
</StackPanel>
```

```
<StackPanel>
  <RadioButton Content="Small" GroupName="Size" IsChecked="{Binding IsSmall, Mode=TwoWay}"/>
  <RadioButton Content="Large" GroupName="Size" IsChecked="{Binding IsLarge, Mode=TwoWay}"/>
</StackPanel>
```

9) Choosing between events and commands

- Prefer Command for user intentions you might test, invoke from multiple places (button, menu, shortcut), or enable/disable.
- Prefer Events for raw input data and gestures where parameters are positional or transient.
- You can mix both: an event handler may delegate to a command in the view model.

Check yourself - Why do commands fit MVVM better than handling all logic in code-behind events? - What happens to a Button when its Command’s CanExecute returns false? - How do you attach Ctrl+S to a command? - When would you pass a CommandParameter?

Look under the hood (repo reading list) - Input and routed events: src/Avalonia.Interactivity, src/Avalonia.Input - Command binding points: ButtonBase.Command, MenuItem.Command, KeyBinding/KeyGesture

Extra practice - Add a CommandParameter to SaveCommand that includes the current text and a timestamp. - Add Ctrl+Delete to trigger DeleteCommand on the selected list item. - Disable Save when a required TextBox is empty (tie CanExecute to your validation). - Show a context menu with a command that acts on the right-clicked item.

What’s next - Next: Chapter 10

10. Working with resources, images, and fonts

Goal - Know how to package and reference app assets with avares URIs. - Display raster images (PNG/JPG) and prefer vector for icons where possible. - Bundle and use custom fonts; understand FontFamily and the "#FaceName" syntax. - Understand DPI scaling in Avalonia and how to keep images/text crisp.

What you'll build - A small gallery view that shows: - A raster logo with Image. - A circular avatar using ImageBrush. - A vector icon drawn with Path. - Headings styled with a bundled custom font.

Prerequisites - You can edit App.axaml and a Window/UserControl (Ch. 3–7). - Basic XAML and binding familiarity (Ch. 5–9).

1) Project assets and avares URIs

- Avalonia embeds UI assets as AvaloniaResource and addresses them using the avares:// URI scheme.
- Start by organizing files under an Assets folder:

Assets/ - Images/ - logo.png - avatar.png - Fonts/ - Inter.ttf

- Ensure your project includes these as AvaloniaResource. Most templates already include a wildcard. If not, add:

```
<ItemGroup>
  <AvaloniaResource Include="Assets/**" />
</ItemGroup>
```

- Referencing an embedded asset uses the assembly name and path:
  - avares://MyApp/Assets/Images/logo.png
  - If you're referencing within the same assembly, still include the assembly segment for clarity and portability.

2) Show an image in XAML (raster)

- The Image control displays bitmaps. Use Stretch to control scaling.

```
<Image Source="avares://MyApp/Assets/Images/logo.png"
       Stretch="Uniform" Width="160"/>
```

- In code-behind you can load from the same URI using AssetLoader:

```
using System;
using Avalonia.Media.Imaging;
using Avalonia.Platform; // AssetLoader

var uri = new Uri("avares://MyApp/Assets/Images/logo.png");
using var stream = AssetLoader.Open(uri);
LogoImage.Source = new Bitmap(stream);
```

Tips - Prefer PNG for UI assets with transparency (icons, logos). JPG is fine for photos. - Keep the source image reasonably large so downscaling on high-DPI looks sharp.

3) Use ImageBrush for backgrounds, shapes, and masks

- ImageBrush paints with an image anywhere a Brush is expected. Common uses: avatar circles, card covers, tiled backgrounds.

```
<Ellipse Width="80" Height="80">
  <Ellipse.Fill>
    <ImageBrush Source="avares://MyApp/Assets/Images/avatar.png"
                Stretch="UniformToFill" AlignmentX="Center" AlignmentY="Center"/>
  </Ellipse.Fill>
</Ellipse>
```

- Other knobs:
  - TileMode="Tile" to repeat an image.
  - SourceRect to select a sub-region.
  - Stretch determines how the image fits: None, Fill, Uniform, UniformToFill.

4) Prefer vector icons when you can

- Vector art scales perfectly at any DPI and is theme-friendly (you can recolor it with brushes).
- A simple way to draw a vector icon is with Path (Data is a geometry string):

```
<Path Data="M 10 2 L 20 22 L 0 22 Z"
      Fill="#2563EB" Width="20" Height="20"/>
```

- You can also keep geometry in a resource:

```
<Window.Resources>
  <Geometry x:Key="IconCheck">M2 10 L8 16 L18 4</Geometry>
</Window.Resources>
<Canvas>
  <Path Data="{StaticResource IconCheck}"
        Stroke="#16A34A" StrokeThickness="2" StrokeLineCap="Round" StrokeLineJoin="Round"/>
</Canvas>
```

Notes - Vector assets are often provided as SVG. You can either convert small SVG paths to Path Data, or use an SVG package if you need full SVG support.

5) Bundle and use custom fonts

- Put your TTF/OTF files under Assets/Fonts and include them as AvaloniaResource.
- Use FontFamily with an avares URI plus the internal face name after #. The part after # must match the font's family name (not the file name).

```
<Application xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             x:Class="MyApp.App">
  <Application.Styles>
    <FluentTheme />
    <!-- Heading style using embedded font -->
    <Style Selector="TextBlock.h1">
      <Setter Property="FontFamily" Value="avares://MyApp/Assets/Fonts/Inter.ttf#Inter"/>
      <Setter Property="FontSize" Value="28"/>
      <Setter Property="FontWeight" Value="SemiBold"/>
    </Style>
  </Application.Styles>
</Application>
```

- Use it in your views:

```
<TextBlock Classes="h1" Text="Resources, images, and fonts"/>
```

Font tips - If the text doesn't render with your font, check the family name embedded in the file (the # part). - You can specify fallbacks: FontFamily="avares://MyApp/Assets/Fonts/Inter.ttf#Inter, Segoe UI, Roboto, Arial".

6) DPI scaling without mystery

- Avalonia measures sizes in device-independent units (DIPs), where 1 unit = 1/96 inch. Your UI scales with monitor DPI.
- Bitmaps are scaled automatically by the composition system. To keep them crisp:
  - Prefer vector for icons and line art.
  - Use sufficiently large raster sources so downscaling looks good (avoid scaling up small images).

  - Use Stretch="Uniform" or "UniformToFill" to avoid distortion.
- Text is vector-based and stays sharp; embedded fonts render through the GPU via the platform renderer.

7) Common pitfalls and how to fix them

- Wrong avares URI: include the correct assembly segment and exact path. Example: avares://MyApp/Assets/Images/log
- Not included as AvaloniaResource: confirm your csproj has and the file exists under that path.
- Font family mismatch: the # part must match the font's internal family name (use a font viewer to verify if needed).
- Theme-unaware icons: prefer vector icons and bind Fill/Foreground to theme brushes (DynamicResource) so they adapt to light/dark.

8) A tiny "assets gallery" to try

```
<Grid ColumnDefinitions="Auto,12,Auto" RowDefinitions="Auto,12,Auto">
  <!-- Raster logo -->
  <Image Grid.Row="0" Grid.Column="0"
         Source="avares://MyApp/Assets/Images/logo.png"
         Width="160" Height="80" Stretch="Uniform"/>

  <!-- Spacer -->
  <Rectangle Grid.Row="0" Grid.Column="1" Width="12"/>

  <!-- Avatar circle from ImageBrush -->
  <Ellipse Grid.Row="0" Grid.Column="2" Width="80" Height="80">
    <Ellipse.Fill>
      <ImageBrush Source="avares://MyApp/Assets/Images/avatar.png"
                  Stretch="UniformToFill"/>
    </Ellipse.Fill>
  </Ellipse>

  <!-- Spacer row -->
  <Rectangle Grid.Row="1" Grid.ColumnSpan="3" Height="12"/>

  <!-- Vector icon (check mark) -->
  <Canvas Grid.Row="2" Grid.Column="0" Width="24" Height="24">
    <Path Data="M2 12 L9 19 L22 4"
          Stroke="#16A34A" StrokeThickness="3" StrokeLineCap="Round" StrokeLineJoin="Round"/>
  </Canvas>

  <!-- Heading with embedded font -->
  <TextBlock Grid.Row="2" Grid.Column="2" Classes="h1" Text="Asset gallery"/>
</Grid>
```

Check yourself - What does the avares:// scheme point to, and why include the assembly segment? - When would you choose Image vs ImageBrush? - How do you reference a font file and its internal face name in FontFamily? - Why do vector icons look better on very high-DPI screens?

Look under the hood (repo reading list) - Images and imaging: src/Avalonia.Media.Imaging, src/Avalonia.Controls (Image) - Brushes and drawing primitives: src/Avalonia.Media - Fonts and text: src/Avalonia.Media.TextFormatting - Skia rendering backend: src/Skia/Avalonia.Skia

Extra practice - Add a dark/light adaptive icon by binding a Path Fill to a DynamicResource brush. - Create a tiled background with ImageBrush and TileMode="Tile". - Add another font weight (e.g., Inter Bold) and make a .h1Bold style. - Load a user-picked image at runtime and display it with Image and ImageBrush variants.

What's next - Next: Chapter 11

11. MVVM in depth (with or without ReactiveUI)

Goal - Go beyond the basics of MVVM and learn two practical ways to structure real apps in Avalonia: classic MVVM with INotifyPropertyChanged and commands, and MVVM powered by ReactiveUI with reactive properties, commands, and routing.

Why this matters - Clear separation of responsibilities keeps your app easy to reason about, test, and extend. - A consistent MVVM approach enables reuse across desktop, mobile, and the browser. - Reactive patterns make complex UI state and async flows easier to compose and test.

Prerequisites - Basic C# classes and properties - Basic XAML and bindings (Chapter 8) - Commands and input (Chapter 9)

What you'll build - A tiny "People" example twice: 1) Classic MVVM: a PeopleViewModel exposes a list, selection, and commands. 2) ReactiveUI: the same features using ReactiveObject and ReactiveCommand. - You'll also see two navigation approaches: a simple view-model-first pattern and ReactiveUI routing.

1) MVVM responsibilities in plain words

- Model: Your data shapes (e.g., Person), plus domain logic. No Avalonia types here.
- ViewModel: UI-facing state and commands. Translates domain into bindable properties. No visual logic or control references.
- View: XAML + code-behind for layout and visuals. No business logic; bindings connect to the ViewModel.

2) Classic MVVM you can ship today

2.1 A minimal base for property change notification

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

public abstract class ObservableObject : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler? PropertyChanged;

    protected bool SetProperty<T>(ref T field, T value, [CallerMemberName] string? name = null)
    {
        if (Equals(field, value)) return false;
        field = value;
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(name));
        return true;
    }
}
```

2.2 A simple ICommand implementation

```
using System;
using System.Windows.Input;

public sealed class RelayCommand : ICommand
{
    private readonly Action<object?> _execute;
    private readonly Func<object?, bool>? _canExecute;

    public RelayCommand(Action<object?> execute, Func<object?, bool>? canExecute = null)
    {
        _execute = execute;
        _canExecute = canExecute;
```

```
    }

    public bool CanExecute(object? parameter) => _canExecute?.Invoke(parameter) ?? true;
    public void Execute(object? parameter) => _execute(parameter);

    public event EventHandler? CanExecuteChanged;
    public void RaiseCanExecuteChanged() => CanExecuteChanged?.Invoke(this, EventArgs.Empty);
}
```

2.3 ViewModels for the People screen

```
using System.Collections.ObjectModel;

public sealed class Person
{
    public string FirstName { get; }
    public string LastName { get; }
    public Person(string first, string last) { FirstName = first; LastName = last; }
    public override string ToString() => $"{FirstName} {LastName}";
}

public sealed class PeopleViewModel : ObservableObject
{
    private Person? _selected;

    public ObservableCollection<Person> People { get; } = new()
    {
        new("Ada", "Lovelace"),
        new("Alan", "Turing"),
        new("Grace", "Hopper")
    };

    public Person? Selected
    {
        get => _selected;
        set
        {
            if (SetProperty(ref _selected, value))
                RemovePersonCommand.RaiseCanExecuteChanged();
        }
    }

    public RelayCommand AddPersonCommand { get; }
    public RelayCommand RemovePersonCommand { get; }

    public PeopleViewModel()
    {
        AddPersonCommand = new RelayCommand(_ => People.Add(new Person("New", "Person")));
        RemovePersonCommand = new RelayCommand(_ =>
        {
            if (Selected is not null)
                People.Remove(Selected);
        }, _ => Selected is not null);
    }
}
```

2.4 View and DataTemplates (ViewModel-first)

```
<!-- App.axaml -->
<Application xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             x:Class="MyApp.App">
  <Application.DataTemplates>
    <!-- Map a ViewModel type to a View -->
    <DataTemplate DataType="{x:Type vm:PeopleViewModel}" xmlns:vm="clr-namespace:MyApp.ViewModels" xmlns
      <v:PeopleView/>
    </DataTemplate>
  </Application.DataTemplates>
</Application>
```

```
<!-- PeopleView.axaml -->
<UserControl xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             x:Class="MyApp.Views.PeopleView">
  <DockPanel Margin="12">
    <StackPanel Orientation="Horizontal" Spacing="8" DockPanel.Dock="Top">
      <Button Content="Add" Command="{Binding AddPersonCommand}"/>
      <Button Content="Remove" Command="{Binding RemovePersonCommand}"/>
    </StackPanel>
    <ListBox Items="{Binding People}" SelectedItem="{Binding Selected}"/>
  </DockPanel>
</UserControl>
```

```
// MainWindow.axaml.cs - set the DataContext to the top-level VM
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        DataContext = new PeopleViewModel();
    }
}
```

2.5 Simple navigation without frameworks - Define a ShellViewModel that holds the current page ViewModel. - Expose commands to swap between page ViewModels. - Use a ContentControl bound to Current in the shell view.

```
public sealed class ShellViewModel : ObservableObject
{
    private object _current;
    public object Current
    {
        get => _current;
        set => SetProperty(ref _current, value);
    }

    public RelayCommand GoPeople { get; }
    public RelayCommand GoAbout { get; }

    public ShellViewModel()
    {
        var people = new PeopleViewModel();
```

```
        var about = new AboutViewModel();
        _current = people;
        GoPeople = new RelayCommand(_ => Current = people);
        GoAbout = new RelayCommand(_ => Current = about);
    }
}
```

```
<!-- ShellView.axaml -->
<UserControl xmlns="https://github.com/avaloniaui" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml
  <DockPanel>
    <StackPanel Orientation="Horizontal" Spacing="8" DockPanel.Dock="Top" Margin="8">
      <Button Content="People" Command="{Binding GoPeople}"/>
      <Button Content="About" Command="{Binding GoAbout}"/>
    </StackPanel>
    <ContentControl Content="{Binding Current}"/>
  </DockPanel>
</UserControl>
```

Notes and tips for classic MVVM - Keep ViewModels free of Avalonia controls. Prefer services for IO, dialogs, and persistence. - Raise CanExecuteChanged when state changes. Disable buttons by command state rather than manual IsEnabled. - Use DataTemplates to map ViewModels to Views; keep View constructors empty of business logic.

3) ReactiveUI in Avalonia

When to consider ReactiveUI - You want observable properties and derived values without boilerplate. - You want commands that automatically manage async execution and can-execute. - You want a simple, testable navigation story (routing) and observable composition.

3.1 Setup - Add the Avalonia.ReactiveUI package to your project. - In the app builder, enable ReactiveUI:

```
AppBuilder.Configure<App>()
    .UsePlatformDetect()
    .UseSkia()
    .UseReactiveUI() // important
    .StartWithClassicDesktopLifetime(args);
```

3.2 ReactiveObject, [ObservableAsProperty] and WhenAnyValue

```
using ReactiveUI;
using System.Reactive;
using System.Reactive.Linq;

public sealed class PersonRx : ReactiveObject
{
    private string _first = string.Empty;
    public string First
    {
        get => _first;
        set => this.RaiseAndSetIfChanged(ref _first, value);
    }

    private string _last = string.Empty;
    public string Last
    {
        get => _last;
        set => this.RaiseAndSetIfChanged(ref _last, value);
    }
```

```
    public string FullName => $"{First} {Last}";
}

public sealed class PeopleViewModelRx : ReactiveObject
{
    private PersonRx? _selected;
    public ObservableCollection<PersonRx> People { get; } = new();

    public PersonRx? Selected
    {
        get => _selected;
        set => this.RaiseAndSetIfChanged(ref _selected, value);
    }

    public ReactiveCommand<Unit, Unit> Add { get; }
    public ReactiveCommand<Unit, Unit> Remove { get; }

    public PeopleViewModelRx()
    {
        People.Add(new PersonRx { First = "Ada", Last = "Lovelace" });
        People.Add(new PersonRx { First = "Alan", Last = "Turing" });
        People.Add(new PersonRx { First = "Grace", Last = "Hopper" });

        var canRemove = this.WhenAnyValue(vm => vm.Selected).Select(sel => sel is not null);
        Add = ReactiveCommand.Create(() => People.Add(new PersonRx { First = "New", Last = "Person" }))
        Remove = ReactiveCommand.Create(() => { if (Selected is not null) People.Remove(Selected); }, ca
    }
}
```

3.3 Binding in XAML is the same

```
<!-- PeopleViewRx.axaml -->
<UserControl xmlns="https://github.com/avaloniaui" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml
  <DockPanel Margin="12">
    <StackPanel Orientation="Horizontal" Spacing="8" DockPanel.Dock="Top">
      <Button Content="Add" Command="{Binding Add}"/>
      <Button Content="Remove" Command="{Binding Remove}"/>
    </StackPanel>
    <ListBox Items="{Binding People}" SelectedItem="{Binding Selected}"/>
  </DockPanel>
</UserControl>
```

3.4 ReactiveUI routing in one minute - Define a host screen that owns a RoutingState. - Expose commands that navigate by pushing new view models. - Views can derive from ReactiveUserControl for WhenActivated hooks, but standard UserControl works too.

```
using ReactiveUI;
using System.Reactive;

public interface IAppScreen : IScreen { }

public sealed class ShellRxViewModel : ReactiveObject, IAppScreen
{
    public RoutingState Router { get; } = new();
```

```
    public ReactiveCommand<Unit, IRoutableViewModel> GoPeople { get; }
    public ReactiveCommand<Unit, IRoutableViewModel> GoAbout { get; }

    public ShellRxViewModel()
    {
        GoPeople = ReactiveCommand.CreateFromObservable(() => Router.Navigate.Execute(new PeopleRoutedVi
        GoAbout = ReactiveCommand.CreateFromObservable(() => Router.Navigate.Execute(new AboutRoutedVie
    }
}

public sealed class PeopleRoutedViewModel : ReactiveObject, IRoutableViewModel
{
    public string? UrlPathSegment => "people";
    public IScreen HostScreen { get; }
    public PeopleViewModelRx Inner { get; } = new();
    public PeopleRoutedViewModel(IScreen host) => HostScreen = host;
}

public sealed class AboutRoutedViewModel : ReactiveObject, IRoutableViewModel
{
    public string? UrlPathSegment => "about";
    public IScreen HostScreen { get; }
    public AboutRoutedViewModel(IScreen host) => HostScreen = host;
}
```

```
<!-- ShellRxView.axaml -->
<UserControl xmlns="https://github.com/avaloniaui" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml
  <DockPanel>
    <StackPanel Orientation="Horizontal" Spacing="8" DockPanel.Dock="Top" Margin="8">
      <Button Content="People" Command="{Binding GoPeople}"/>
      <Button Content="About" Command="{Binding GoAbout}"/>
    </StackPanel>
    <!-- RoutedViewHost displays the View for the current IRoutableViewModel -->
    <rxui:RoutedViewHost Router="{Binding Router}" xmlns:rxui="clr-namespace:ReactiveUI;assembly=Reacti
  </DockPanel>
</UserControl>
```

3.5 Interactions (dialogs without coupling) - ReactiveUI's Interaction<TIn, TOut> lets ViewModels request UI work (like a file dialog) while remaining testable. - Views subscribe to interactions and fulfill them at the edge.

```
public sealed class SaveViewModel : ReactiveObject
{
    public Interaction<Unit, bool> ConfirmSave { get; } = new();
    public ReactiveCommand<Unit, Unit> Save { get; }

    public SaveViewModel()
    {
        Save = ReactiveCommand.CreateFromTask(async () =>
        {
            var ok = await ConfirmSave.Handle(Unit.Default);
            if (ok)
            {
                // perform save
            }
```

```
        });
    }
}
```

In the View (code-behind), subscribe when activated:

```
this.WhenActivated(d =>
{
    d(ViewModel!.ConfirmSave.RegisterHandler(async ctx =>
    {
        var result = await ShowDialogAsync("Save?", "Do you want to save?", "Yes", "No");
        ctx.SetOutput(result);
    }));
});
```

4) Validation options that scale

- Minimal: manual validation in commands (already shown earlier).
- INotifyDataErrorInfo: push validation errors per property; Avalonia supports Validation styling and Adorners.
- ReactiveUI.Validation (optional package) offers fluent rules bound to reactive properties.

5) Testing your ViewModels

- Classic MVVM: instantiate the VM and test property changes and command behavior.
- ReactiveUI: use TestScheduler to verify reactive flows; test ReactiveCommand execution and can-execute.

6) Choosing your path

- Start with classic MVVM if you prefer straightforward classes and minimal dependencies.
- Choose ReactiveUI when you need complex async workflows, derived state, or built-in routing/interaction patterns.
- You can mix: classic VMs for simple pages; ReactiveUI for complex areas.

Look under the hood (source) - Avalonia + ReactiveUI integration: Avalonia.ReactiveUI - Validation styles: Avalonia.Themes.Fluent

Check yourself - What problems does MVVM solve in UI apps? - How do DataTemplates map ViewModels to Views? - When would you choose ReactiveCommand over a manual ICommand? - What does Router.Navigate.Execute do in ReactiveUI routing?

Extra practice - Convert the classic People example to ReactiveUI step by step. Keep behavior identical. - Add a Save dialog using ReactiveUI's Interaction pattern, and stub it out in tests. - Add a third page and wire it into both the simple shell and the reactive router.

What's next - Next: Chapter 12

12. Navigation, windows, and lifetimes

Goal - Learn how Avalonia apps start and keep running across platforms, how to create and manage windows on desktop, and simple navigation patterns that work for both desktop (multi-window) and mobile/web (single view) apps.

Why this matters - Understanding lifetimes ensures your app starts, navigates, and shuts down reliably on each platform. - Good windowing and navigation patterns reduce coupling and make features easier to test and evolve.

Prerequisites - Chapters 4 (startup), 8 (bindings), and 11 (MVVM patterns)

What you'll build - A desktop app with a main window, an About dialog (modal), and a basic page switcher. - A single-view setup (mobile/web) that hosts pages inside a single root view.

1) App lifetimes at a glance

- ClassicDesktopStyleApplicationLifetime (desktop):
  - You set MainWindow, can open multiple windows, and control shutdown behavior.
- SingleViewApplicationLifetime (mobile/web):
  - You provide a single root view (MainView). No Window instances; navigation happens inside that view.

Init both from App.OnFrameworkInitializationCompleted

```
public override void OnFrameworkInitializationCompleted()
{
    if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
    {
        desktop.MainWindow = new MainWindow
        {
            DataContext = new ShellViewModel()
        };
        // Optional: choose how the app shuts down
        desktop.ShutdownMode = ShutdownMode.OnLastWindowClose; // or OnMainWindowClose / OnExplicitShut
    }
    else if (ApplicationLifetime is ISingleViewApplicationLifetime singleView)
    {
        singleView.MainView = new ShellView // a UserControl
        {
            DataContext = new ShellViewModel()
        };
    }

    base.OnFrameworkInitializationCompleted();
}
```

2) Windows on desktop: main, owned, and modal

2.1 Creating another window

```
public class AboutWindow : Window
{
    public AboutWindow()
    {
        Title = "About";
        Width = 400; Height = 220;
        WindowStartupLocation = WindowStartupLocation.CenterOwner;
        Content = new TextBlock { Text = "My App v1.0", Margin = new Thickness(16) };
```

```
    }
}
```

2.2 Showing non-modal vs modal

```
// Non-modal (does not block owner)
var about = new AboutWindow { Owner = this }; // 'this' is a Window
about.Show();

// Modal (blocks owner, returns when closed)
var resultTask = new AboutWindow { Owner = this }.ShowDialog(this);
await resultTask; // continues after dialog closes
```

2.3 Returning data from a dialog

```
public class NameDialog : Window
{
    private readonly TextBox _name = new();
    public string? EnteredName { get; private set; }
    public NameDialog()
    {
        Title = "Enter name";
        WindowStartupLocation = WindowStartupLocation.CenterOwner;
        Content = new StackPanel { Margin = new Thickness(16), Children =
        {
            new TextBlock { Text = "Name:" },
            _name,
            new StackPanel { Orientation = Orientation.Horizontal, Spacing = 8, Children =
            {
                new Button { Content = "OK", IsDefault = true, Command = ReactiveCommand.Create(() => {
                new Button { Content = "Cancel", IsCancel = true, Command = ReactiveCommand.Create(() =
            }}
        }};
    }
}

// Usage (in a Window)
var dlg = new NameDialog { Owner = this };
var ok = await dlg.ShowDialog<bool>(this);
if (ok)
{
    var name = dlg.EnteredName;
    // use name
}
```

Tips - Always set Owner for child windows so modality/centering behave as expected. - Use ShowDialog for blocking flows (confirmations, wizards), Show for tool windows.

3) Simple navigation patterns that scale

3.1 View-model–first shell (works for desktop and single-view)

```
public sealed class ShellViewModel : ObservableObject
{
    private object _current;
    public object Current { get => _current; set => SetProperty(ref _current, value); }

    public RelayCommand GoHome { get; }
```

```
    public RelayCommand GoSettings { get; }

    public ShellViewModel()
    {
        var home = new HomeViewModel();
        var settings = new SettingsViewModel();
        _current = home;
        GoHome = new RelayCommand(_ => Current = home);
        GoSettings = new RelayCommand(_ => Current = settings);
    }
}
```

```
<!-- ShellView.axaml (UserControl used for both desktop MainWindow content and single-view MainView) --
<UserControl xmlns="https://github.com/avaloniaui" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml
  <DockPanel>
    <StackPanel Orientation="Horizontal" Spacing="8" DockPanel.Dock="Top" Margin="8">
      <Button Content="Home" Command="{Binding GoHome}"/>
      <Button Content="Settings" Command="{Binding GoSettings}"/>
    </StackPanel>
    <ContentControl Content="{Binding Current}"/>
  </DockPanel>
</UserControl>
```

3.2 Mapping ViewModels to Views via DataTemplates

```
<!-- App.axaml -->
<Application xmlns="https://github.com/avaloniaui" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml
  <Application.DataTemplates>
    <DataTemplate DataType="{x:Type vm:HomeViewModel}" xmlns:vm="clr-namespace:MyApp.ViewModels" xmlns:v
      <v:HomeView/>
    </DataTemplate>
    <DataTemplate DataType="{x:Type vm:SettingsViewModel}" xmlns:vm="clr-namespace:MyApp.ViewModels" xml
      <v:SettingsView/>
    </DataTemplate>
  </Application.DataTemplates>
</Application>
```

Note: If you're using ReactiveUI, you can also adopt its Router + RoutedViewHost (see Chapter 11).

4) Closing, shutdown, and lifetime APIs

4.1 Window closing and cancel

```
// In a Window constructor
this.Closing += (s, e) =>
{
    if (HasUnsavedChanges)
    {
        // Ask the user and optionally cancel
        // e.Cancel = true; // keep window open
    }
};
```

4.2 Controlling application shutdown (desktop)

```
if (Application.Current?.ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
{
    desktop.ShutdownMode = ShutdownMode.OnMainWindowClose; // Or OnLastWindowClose / OnExplicitShutdown
```

```
    // Later, when appropriate
    // desktop.Shutdown();
}
```

Single-view apps don't manage process shutdown directly—platforms (mobile/web) handle lifecycle; navigate back or update the root view instead of closing windows.

5) File dialogs and pickers

5.1 Classic file dialogs (desktop)

```
var ofd = new OpenFileDialog
{
    AllowMultiple = false,
    Filters =
    {
        new FileDialogFilter { Name = "Text", Extensions = { "txt", "md" } },
        new FileDialogFilter { Name = "All", Extensions = { "*" } }
    }
};
var files = await ofd.ShowAsync(this); // 'this' is a Window
if (files?.Length > 0)
{
    var path = files[0];
    // open file
}

var sfd = new SaveFileDialog
{
    InitialFileName = "document.txt",
    Filters = { new FileDialogFilter { Name = "Text", Extensions = { "txt" } } }
};
var path = await sfd.ShowAsync(this);
if (!string.IsNullOrEmpty(path))
{
    // save file
}
```

5.2 Cross-platform storage provider (works in single-view)

```
var top = TopLevel.GetTopLevel(this); // from a Control
if (top?.StorageProvider is { } sp)
{
    var results = await sp.OpenFilePickerAsync(new FilePickerOpenOptions
    {
        AllowMultiple = false,
        FileTypeFilter = new[] { FilePickerFileTypes.TextPlain }
    });
    var file = results.FirstOrDefault();
    if (file is not null)
    {
        await using var stream = await file.OpenReadAsync();
        // read stream
    }
}
```

6) Cross-platform guidelines

- Desktop: prefer windows for tools and modal flows; keep ownership set and use CenterOwner.
- Mobile/web (single-view): keep everything in a single root view; navigate by swapping ViewModels in a ContentControl.
- Shared code: keep services (dialogs, storage) behind interfaces so ViewModels don't depend on Window.

Look under the hood (source) - Window class: Avalonia.Controls/Window.cs - Classic desktop lifetime: Avalonia.Controls/ApplicationLifetimes/ClassicDesktopStyleApplicationLifetime.cs - Single-view lifetime: Avalonia.Controls/ApplicationLifetimes/SingleViewApplicationLifetime.cs - Open/Save dialogs: Avalonia.Controls

Check yourself - What's the difference between ClassicDesktopStyleApplicationLifetime and SingleViewApplicationLifetime? - When should you use Show vs ShowDialog? - How do DataTemplates enable view-model–first navigation? - Where would you set ShutdownMode and why?

Extra practice - Add a Settings dialog to your app that returns a result and updates the main view. - Implement a shell with three pages and keyboard shortcuts for navigation. - Replace OpenFileDialog with the StorageProvider API and make it work in a single-view setup.

What's next - Next: Chapter 13

13. Menus, dialogs, tray icons, and system features

Goal - Build desktop-friendly menus and context menus, design testable dialog flows, add a system tray icon with a menu, and learn platform notes so your features behave correctly on Windows, macOS, and Linux.

Why this matters - Menus and dialogs are core desktop UX. - A clean dialog and tray-icon approach keeps ViewModels testable and UI responsive. - Platform-aware patterns save time when you target multiple OSes.

Prerequisites - Chapters 8–12 (binding, commands, lifetimes, windows)

What you'll build - A top app menu (in-window Menu and native menu bar) with keyboard shortcuts. - Context menus and flyouts for in-place actions. - A reusable dialog pattern (task-based) without coupling VMs to Window. - A tray icon with a small menu and actions.

1) Application menu bar (desktop)

1.1 In-window Menu (cross-platform)

```
<!-- MainWindow.axaml -->
<Window xmlns="https://github.com/avaloniaui" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="MyApp.MainWindow" Width="800" Height="500" Title="My App">
  <DockPanel>
    <Menu DockPanel.Dock="Top">
      <MenuItem Header="_File">
        <MenuItem Header="_New" Command="{Binding NewCommand}" InputGestureText="Ctrl+N"/>
        <MenuItem Header="_Open…" Command="{Binding OpenCommand}" InputGestureText="Ctrl+O"/>
        <Separator/>
        <MenuItem Header="_Exit" Command="{Binding ExitCommand}"/>
      </MenuItem>
      <MenuItem Header="_Help">
        <MenuItem Header="_About…" Command="{Binding ShowAboutCommand}"/>
      </MenuItem>
    </Menu>

    <!-- main content here -->
    <ContentControl Content="{Binding Current}"/>
  </DockPanel>
</Window>
```

- InputGestureText shows the shortcut in the menu; bind actual shortcuts with KeyBindings in the Window or App (see Chapter 9).

1.2 Native menu bar (macOS-style global menu)

```
<!-- MainWindow.axaml (top-level menu that can integrate with the OS) -->
<Window ... xmlns:native="clr-namespace:Avalonia.Controls;assembly=Avalonia.Controls">
  <DockPanel>
    <native:NativeMenuBar DockPanel.Dock="Top">
      <native:NativeMenuBar.Menu>
        <native:NativeMenu>
          <native:NativeMenuItem Header="My App">
            <native:NativeMenuItem Header="About" Command="{Binding ShowAboutCommand}"/>
            <native:NativeMenuSeparator/>
            <native:NativeMenuItem Header="Quit" Command="{Binding ExitCommand}"/>
          </native:NativeMenuItem>
          <native:NativeMenuItem Header="File">
            <native:NativeMenuItem Header="New" Command="{Binding NewCommand}"/>
            <native:NativeMenuItem Header="Open…" Command="{Binding OpenCommand}"/>
          </native:NativeMenuItem>
```

```
        </native:NativeMenu>
      </native:NativeMenuBar.Menu>
    </native:NativeMenuBar>
    <!-- rest of layout -->
  </DockPanel>
</Window>
```

Notes - Use in-window Menu for all platforms; NativeMenuBar gives tighter OS integration on macOS and supported platforms. - Keep commands in your ViewModel; menus should be just bindings.

2) Context menus and flyouts

2.1 ContextMenu on any control

```
<Button Content="Options">
  <Button.ContextMenu>
    <ContextMenu>
      <MenuItem Header="Copy" Command="{Binding Copy}"/>
      <MenuItem Header="Paste" Command="{Binding Paste}"/>
      <Separator/>
      <MenuItem Header="Delete" Command="{Binding Delete}"/>
    </ContextMenu>
  </Button.ContextMenu>
</Button>
```

2.2 Flyouts for lightweight actions

```
<Button Content="More" xmlns:ui="https://github.com/avaloniaui">
  <Button.Flyout>
    <Flyout>
      <StackPanel Margin="8" Spacing="8">
        <TextBlock Text="Quick actions"/>
        <Button Content="Refresh" Command="{Binding Refresh}"/>
        <Button Content="Settings" Command="{Binding OpenSettings}"/>
      </StackPanel>
    </Flyout>
  </Button.Flyout>
</Button>
```

Tips - Prefer ContextMenu for command lists; prefer Flyout for custom content and small toolpanels. - For list items, provide an ItemContainerStyle that sets a ContextMenu per row if needed.

3) Dialogs without tight coupling

3.1 Simple custom dialog window pattern

```
public class AboutWindow : Window
{
    public AboutWindow()
    {
        Title = "About";
        Width = 360; Height = 220;
        WindowStartupLocation = WindowStartupLocation.CenterOwner;
        Content = new StackPanel { Margin = new Thickness(16), Children =
        {
            new TextBlock { Text = "My App v1.0" },
            new Button { Content = "OK", IsDefault = true, Command = ReactiveCommand.Create(() => Close
        }};
```

```
    }
}
```

Show it from a Window

```
var ok = await new AboutWindow { Owner = this }.ShowDialog<bool>(this);
```

3.2 A dialog service interface (testable ViewModels)

```
public interface IDialogService
{
    Task<bool> ShowAboutAsync(Window owner);
}

public sealed class DialogService : IDialogService
{
    public async Task<bool> ShowAboutAsync(Window owner)
        => await new AboutWindow { Owner = owner }.ShowDialog<bool>(owner);
}
```

Use it in a ViewModel via an abstraction

```
public sealed class ShellViewModel : ObservableObject
{
    private readonly IDialogService _dialogs;
    public RelayCommand ShowAboutCommand { get; }

    public ShellViewModel(IDialogService dialogs)
    {
        _dialogs = dialogs;
        ShowAboutCommand = new RelayCommand(async o =>
        {
            // Owner is supplied by the View (e.g., via CommandParameter binding)
            if (o is Window owner)
                await _dialogs.ShowAboutAsync(owner);
        });
    }
}
```

View wiring example

```
<Window ...>
  <Window.DataContext>
    <!-- Assume a DI container provides DialogService; for demo we use x:FactoryMethod in code-behind -
  </Window.DataContext>
  <Button Content="About" Command="{Binding ShowAboutCommand}" CommandParameter="{Binding $parent[Windo
</Window>
```

Notes - This keeps the ViewModel free of Window references; only the View passes the owner. - For more advanced flows, consider ReactiveUI Interactions (covered in Chapter 11).

4) Tray icon (system notification area)

4.1 Creating and showing a tray icon

```
// In App.OnFrameworkInitializationCompleted (desktop lifetime)
if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
{
    var tray = new TrayIcon
    {
```

```
        ToolTipText = "My App",
        Icon = new WindowIcon("avares://MyApp/Assets/AppIcon.ico"),
        Menu = new NativeMenu
        {
            Items =
            {
                new NativeMenuItem("Show") { Command = ReactiveCommand.Create(() => desktop.MainWindow?
                new NativeMenuItem("Exit") { Command = ReactiveCommand.Create(() => desktop.Shutdown())
            }
        }
    };
    tray.Show();
}
```

Notes - Keep a reference to the TrayIcon if you need to toggle visibility or update its menu. - Tray menus should be short and essential; keep the main app UI for complex tasks.

5) Shortcuts and accelerators in menus

- Use InputGestureText on MenuItem to display the shortcut (e.g., Ctrl+N) and pair it with a KeyBinding at Window/App level to trigger the same command.
- On macOS, Cmd is the conventional modifier; consider platform-specific gesture strings in your help text.

6) Platform notes and guidance

- macOS: Prefer NativeMenuBar for the top menu; tray icon shows in the status bar. Some menu roles are handled by the OS.
- Windows/Linux: In-window Menu is the default. Tray icons rely on a running notification area.
- Mobile/Web (single-view): Menus and tray icons don't apply; use flyouts, toolbars, and page navigation instead.

Look under the hood (source) - Menus (in-window): Avalonia.Controls - Native menus: Avalonia.Controls - ContextMenu and Flyout: Avalonia.Controls - TrayIcon: Avalonia.Controls - Optional notifications: Avalonia.Controls.Notifications

Check yourself - When would you choose NativeMenuBar over an in-window Menu? - How do you attach a ContextMenu to a control, and when would a Flyout be a better fit? - How do you invoke a dialog without coupling the ViewModel to Window? - Where should the tray icon be created and how do you handle its menu actions?

Extra practice - Add keyboard shortcuts for File → New/Open using KeyBinding, and show them via InputGestureText. - Add a context menu to a ListBox that exposes row-level actions (Rename/Delete). - Add a tray icon with "Show/Hide" and "Exit", and ensure it restores the window when closed to tray.

What's next - Next: Chapter 14

14. Lists, virtualization, and performance

Goal - Render thousands to millions of items smoothly by using the right control, lightweight templates, and UI virtualization.

Why this matters - Lists are everywhere: mail, logs, search results, tables. Without virtualization, memory and CPU costs explode and scrolling stutters. - Avalonia gives you the tools to keep UIs fast if you pick the right patterns.

Pick the right control - ItemsControl: simplest items host; no selection or keyboard navigation built in. Good for read-only, simple visuals. - ListBox: adds selection (single/multiple), keyboard navigation, and item container generation. Good default for list UIs. - DataGrid: tabular data with columns, sorting, editing, and virtualization; great for large datasets that fit a grid. - TreeView: hierarchical lists; consider flattening + grouping if deep trees hurt performance.

Data and templates you'll actually use - Back your list with ObservableCollection so add/remove updates are cheap and incremental. - Provide ItemTemplate to render lightweight visuals: - Prefer a single panel (e.g., Grid with defined columns/rows) over nested StackPanels. - Minimize triggers/animations per item; avoid heavy effects and large images. - Do work in view models (pre-format strings, compute colors) instead of costly converters per item.

UI virtualization: the mental model - Only the items in (or near) the viewport have visual containers; off-screen items are not realized. - Recycling reuses containers as you scroll so the app avoids allocating/destroying many controls. - Virtualization depends on the items panel and scroll host. Don't accidentally disable it by changing panels.

Enable virtualization (ListBox and ItemsControl) - Use a virtualizing panel for the items host. The common choice is VirtualizingStackPanel.

Example: ListBox with virtualization and a lightweight template

```
<ListBox Items="{Binding Items}" SelectedItem="{Binding Selected}">
  <ListBox.ItemsPanel>
    <ItemsPanelTemplate>
      <VirtualizingStackPanel/>
    </ItemsPanelTemplate>
  </ListBox.ItemsPanel>
  <ListBox.ItemTemplate>
    <DataTemplate>
      <Grid ColumnDefinitions="Auto,*,Auto" Margin="4" Height="32">
        <TextBlock Grid.Column="0" Text="{Binding Id}" Width="56" HorizontalAlignment="Right"/>
        <TextBlock Grid.Column="1" Text="{Binding Title}" Margin="8,0"/>
        <TextBlock Grid.Column="2" Text="{Binding Status}" Foreground="{Binding StatusColor}"/>
      </Grid>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Notes - Keep row Height fixed when possible for smoother virtualization. - Avoid placing a ScrollViewer inside each item; the ListBox already scrolls.

ItemsControl with virtualization

```
<ItemsControl Items="{Binding Items}">
  <ItemsControl.ItemsPanel>
    <ItemsPanelTemplate>
      <VirtualizingStackPanel/>
    </ItemsPanelTemplate>
  </ItemsControl.ItemsPanel>
```

```
  <ItemsControl.ItemTemplate>
    <DataTemplate>
      <Border Margin="2" Padding="6">
        <TextBlock Text="{Binding}"/>
      </Border>
    </DataTemplate>
  </ItemsControl.ItemTemplate>
</ItemsControl>
```

Selection and commands without leaks - Bind SelectedItem (or SelectedItems for multi-select) to your view model. - Prefer commands in the item view model (e.g., OpenCommand) instead of per-item event handlers.

Incremental loading: load what you need when you need it - Pattern: begin with the first N items, then append more when the user scrolls near the bottom. - Keep an IsLoading flag and a cancellation token to avoid overlapping requests.

Simple pattern using a sentinel “Load more” item

```
<ListBox Items="{Binding PagedItems}">
  <ListBox.ItemTemplate>
    <DataTemplate x:DataType="vm:ItemOrCommand">
      <ContentControl>
        <ContentControl.Styles>
          <Style Selector="ContentControl:has(^vm|LoadMore)">
            <Setter Property="ContentTemplate">
              <Setter.Value>
                <DataTemplate>
                  <Button Command="{Binding DataContext.LoadMoreCommand, RelativeSource={RelativeSource
                          Content="Load more…"/>
                </DataTemplate>
              </Setter.Value>
            </Setter>
          </Style>
          <Style Selector="ContentControl:has(^vm|Item)">
            <Setter Property="ContentTemplate">
              <Setter.Value>
                <DataTemplate>
                  <TextBlock Text="{Binding Title}"/>
                </DataTemplate>
              </Setter.Value>
            </Setter>
          </Style>
        </ContentControl.Styles>
      </ContentControl>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

View model sketch

```
public partial class ListPageViewModel
{
    public ObservableCollection<ItemOrCommand> PagedItems { get; } = new();
    public ICommand LoadMoreCommand { get; }
    private int _page = 0;
    private const int PageSize = 200;
    private bool _loading;
```

```
    public ListPageViewModel()
    {
        LoadMoreCommand = new RelayCommand(async () => await LoadPageAsync(), () => !_loading);
        _ = LoadPageAsync();
    }

    private async Task LoadPageAsync()
    {
        if (_loading) return;
        _loading = true;
        try
        {
            if (PagedItems.LastOrDefault() is LoadMore lm)
                PagedItems.Remove(lm);

            var next = await Repository.GetPageAsync(_page, PageSize);
            foreach (var item in next)
                PagedItems.Add(new Item(item));

            _page++;
            if (next.Count == PageSize)
                PagedItems.Add(new LoadMore());
        }
        finally
        {
            _loading = false;
            (LoadMoreCommand as RelayCommand)?.RaiseCanExecuteChanged();
        }
    }
}

public abstract record ItemOrCommand;
public record Item(Model Model) : ItemOrCommand { public string Title => Model.Title; }
public record LoadMore : ItemOrCommand;
```

DataGrid performance quick wins - Define columns explicitly; avoid AutoGenerateColumns for huge datasets. - Prefer TextBlock for display cells; use editing templates only when needed. - Keep cell templates lean; avoid images/effects in cells unless necessary. - Paging and server-side filtering/sorting reduce memory and keep UI snappy.

TreeView tips - Keep item visuals light and collapse subtrees not in view. - If the tree is deep and wide, consider an alternative UX (search + flat list/grouping) for performance.

Scrolling smoothness and item size - Fixed item heights help virtualization predict layout and reduce jank. - If items vary, cap the maximum size and truncate/clip text rather than wrapping across many lines.

Avoid these pitfalls - Nesting ScrollViewer inside each item: breaks virtualization and harms perf. - Binding to huge images per row: use thumbnails or async image loading. - Heavy converters per row: precompute in view models. - Too many nested panels: flatten with Grid for fewer elements. - Selecting all items frequently: track only what you need.

Hands-on: build a fast log viewer 1) Create a ListBox with VirtualizingStackPanel and a fixed-height row template. 2) Stream log lines into an ObservableCollection. 3) Add a Toggle to pause autoscroll when the user interacts. 4) Add a filter box that updates the source collection in batches to avoid UI thrash.

Look under the hood (browse the source) - Controls & containers: src/Avalonia.Controls - DataGrid:

src/Avalonia.Controls.DataGrid - Diagnostics/DevTools: src/Avalonia.Diagnostics

Self-check - What's the difference between ItemsControl and ListBox? - Why does a fixed item height help virtualization? - How would you implement incremental loading for a remote API? - Name three things that commonly break virtualization.

Extra practice - Replace nested StackPanels in an item template with a single Grid and compare element counts. - Add "Load more" to a list that currently fetches everything at once. - Profile memory while scrolling 100k items with and without virtualization.

What's next - Next: Chapter 15

15. Accessibility and internationalization

Goal - Make your app usable by everyone, with keyboard, screen readers, and assistive tech. Localize your UI for different languages and regions.

Why it matters - Accessibility is not optional: keyboard users, low-vision users, and screen-reader users should complete all key tasks. - Internationalization (i18n) broadens your audience: localized strings, culture-aware formatting, right-to-left (RTL) layouts, and font fallback ensure a first-class experience worldwide.

What you'll build in this chapter - Keyboard-friendly forms and menus with predictable tab order and access keys. - Screen-reader hints via AutomationProperties. - A simple localization pipeline using .resx resources, a tiny Localizer helper, and runtime culture switching. - RTL–aware layouts and a default font family that supports your target scripts.

1) Keyboard accessibility from the start

- Focus and Tab order
  - Every interactive control should be reachable with Tab/Shift+Tab.
  - Use IsTabStop and TabIndex on controls.
  - Use KeyboardNavigation.TabNavigation on containers to define how focus moves within them.

XAML example: predictable tab order in a form

```
<StackPanel Spacing="8" KeyboardNavigation.TabNavigation="Cycle">
  <TextBlock Text="_User name" RecognizesAccessKey="True"/>
  <TextBox TabIndex="0" Name="UserName"/>

  <TextBlock Text="_Password" RecognizesAccessKey="True"/>
  <TextBox TabIndex="1" Name="Password" PasswordChar="•"/>

  <CheckBox TabIndex="2" Content="_Remember me"/>

  <StackPanel Orientation="Horizontal" Spacing="8">
    <Button TabIndex="3">
      <ContentPresenter>
        <TextBlock Text="_Sign in" RecognizesAccessKey="True"/>
      </ContentPresenter>
    </Button>
    <Button TabIndex="4">
      <ContentPresenter>
        <TextBlock Text="_Cancel" RecognizesAccessKey="True"/>
      </ContentPresenter>
    </Button>
  </StackPanel>
</StackPanel>
```

Notes - RecognizesAccessKey=“True” lets the underscore (_) mark an access key; Alt+letter triggers the nearest logical action. - KeyboardNavigation.TabNavigation: - Continue (default): tab moves into and then out of the container - Cycle: focus wraps inside the container - Once: the first Tab focuses the first child, the next Tab leaves the container - Local: Tab only moves inside the container - None: Tab does not move focus inside

2) Access keys that actually work

- Access keys let users activate commands using Alt+Letter. In Avalonia, you can:
  - Use AccessText around text that contains underscores.
  - Or set RecognizesAccessKey=“True” on TextBlock within the content of Button/MenuItem.

XAML examples

```
<Button>
  <AccessText Text="_Open"/>
</Button>

<MenuItem>
  <MenuItem.Header>
    <AccessText Text="_File"/>
  </MenuItem.Header>
</MenuItem>
```

3) Screen reader semantics with AutomationProperties

- AutomationProperties is how you describe UI semantics for assistive technologies.
- Common attached properties:
  - AutomationProperties.Name: the accessible label (what the control is called)
  - AutomationProperties.HelpText: extra explanation or hint
  - AutomationProperties.AutomationId: stable ID used by UI tests and screen readers
  - AutomationProperties.LabeledBy: link a control to its visible label element
  - AutomationProperties.LiveSetting: announce dynamic updates (polite/assertive)

XAML examples

```
<StackPanel Spacing="8">
  <TextBlock x:Name="UserNameLabel" Text="User name"/>
  <TextBox Name="UserName"
           AutomationProperties.LabeledBy="{Binding #UserNameLabel}"
           AutomationProperties.HelpText="Enter your account name"/>

  <TextBlock x:Name="StatusLabel" Text="Status"/>
  <TextBlock Name="StatusText"
             AutomationProperties.LabeledBy="{Binding #StatusLabel}"
             AutomationProperties.LiveSetting="Polite"
             Text="Ready"/>
</StackPanel>
```

Tips - Prefer visible labels connected with LabeledBy. If there's no visible label, set AutomationProperties.Name. - Keep HelpText short and specific ("Press Enter to search").

4) Testing keyboard and screen readers

- Keyboard: Tab through every interactive element. Make sure focus is visible and the order is logical.
- Screen readers: Try Narrator (Windows), VoiceOver (macOS/iOS), Orca (Linux). Verify names, roles, states, and readout order.
- Menus and dialogs: Ensure access keys and Esc/Enter behave as expected.

5) Internationalization: the simplest path that scales

- Approach
  - Store localizable strings in .resx files (e.g., Properties/Resources.resx, plus Resources.fr.resx etc.).
  - Use a tiny Localizer that reads from ResourceManager and raises notifications when culture changes.
  - Bind to that Localizer from XAML using an indexer binding. No special XAML extension required.

Localizer helper (C#)

```
using System;
using System.ComponentModel;
using System.Globalization;
```

```
using System.Resources;

namespace MyApp.Localization;

public sealed class Loc : INotifyPropertyChanged
{
    private readonly ResourceManager _resources = Properties.Resources.ResourceManager;

    public string this[string key]
        => _resources.GetString(key, CultureInfo.CurrentUICulture) ?? key;

    public void SetCulture(CultureInfo culture)
    {
        CultureInfo.CurrentUICulture = culture;
        CultureInfo.CurrentCulture = culture;
        // Notify indexer bindings to refresh all strings
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Item[]"));
    }

    public event PropertyChangedEventHandler? PropertyChanged;
}
```

Wiring it in XAML - Put a Loc instance in resources you can reach from every view (e.g., App.Resources or a top-level Window).

App.xaml (snippet)

```
<Application xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="using:MyApp.Localization">
  <Application.Resources>
    <local:Loc x:Key="Loc"/>
  </Application.Resources>
</Application>
```

Using localized strings

```
<Menu>
  <MenuItem Header="{Binding [File], Source={StaticResource Loc}}"/>
  <MenuItem Header="{Binding [Edit], Source={StaticResource Loc}}"/>
</Menu>

<TextBlock Text="{Binding [Ready], Source={StaticResource Loc}}"/>
```

Switching languages at runtime (C#)

```
using System.Globalization;
using Avalonia;
using MyApp.Localization;

// For example, in a menu command or settings page:
var loc = (Loc)Application.Current!.Resources["Loc"];
loc.SetCulture(new CultureInfo("pl-PL"));
```

Formatting that respects culture - .NET formatting uses CurrentCulture automatically.

```
<TextBlock Text="{Binding Price, StringFormat={}{0:C}}"/>
<TextBlock Text="{Binding Date, StringFormat={}{0:D}}"/>
```

6) Right-to-left (RTL) support with FlowDirection

- Some languages (Arabic, Hebrew, Farsi) require RTL text and mirrored layouts.
- Set FlowDirection on a Window or any container; children inherit unless overridden.

Examples

```
<Window FlowDirection="RightToLeft">
  <StackPanel>
    <TextBlock Text="{Binding [Hello], Source={StaticResource Loc}}"/>
    <!-- Controls mirror automatically when possible -->
  </StackPanel>
</Window>

<!-- Override for a specific control -->
<TextBox FlowDirection="LeftToRight"/>
```

7) Fonts and fallback: show all glyphs

- Pick a default font family that supports your target scripts, or bundle fonts with your app.
- Configure a default family with FontManagerOptions during app startup.

Program.cs (desktop)

```
using Avalonia;
using Avalonia.Media;

BuildAvaloniaApp()
    .With(new FontManagerOptions
    {
        // A family with broad Unicode coverage
        DefaultFamilyName = "Noto Sans"
    });

static AppBuilder BuildAvaloniaApp()
    => AppBuilder.Configure<App>()
                 .UsePlatformDetect()
                 .LogToTrace();
```

Notes - You can also embed font files as Avalonia resources and reference them in XAML with FontFamily. - Test for glyph coverage (CJK, Arabic/Hebrew, emoji) on each platform.

8) Checklist you can actually use

- Keyboard
  - Every action reachable by keyboard (tab, access keys, shortcuts)
  - Visual focus indicator is always visible
- Screen reader
  - Important elements have clear Name and HelpText
  - Status updates use LiveSetting when appropriate
- Internationalization
  - All visible strings come from resources
  - Prices, dates, numbers show in the selected culture
  - Switching language at runtime refreshes UI text
- RTL and fonts
  - FlowDirection works as expected; icons look correct when mirrored
  - The chosen default font displays all required scripts

Common pitfalls to avoid - Hard-coded strings in XAML or code — put them in resources. - Hidden controls

that still receive focus — set IsTabStop="False" or remove from tab order. - Relying only on icons or color — add text labels and sufficient contrast.

Look under the hood (source links) - Access keys rendering: AccessText - Avalonia.Controls/Primitives/AccessText.cs - Access key routing and handling - Avalonia.Base/Input/AccessKeyHandler.cs - Keyboard navigation (Tab/arrow movement) - Avalonia.Base/Input/KeyboardNavigation.cs - Focusability and tabbing properties - Avalonia.Base/Input/InputElement.cs - Accessibility attached properties - Avalonia.Controls/Automation/AutomationProperties.cs - Right-to-left direction - Avalonia.Visuals/FlowDirection.cs - Default font configuration - Avalonia.Base/Media/FontManagerOptions.cs

Check yourself - How do you link a TextBox to its visible label so a screen reader announces them together? - What does KeyboardNavigation.TabNavigation="Cycle" change compared to the default? - How do you update all localized strings when the user changes language at runtime? - Where do you set a default font family that supports your target scripts?

Extra practice - Add access keys to your app's main menu and verify they work with Alt key. - Add AutomationProperties.Name and HelpText to a form and test with a screen reader on your OS. - Create Resources.es.resx and Resources.ar.resx, switch to Spanish and Arabic at runtime, enable RTL, and review layout.

What's next - Next: Chapter 16

16. Files, storage, drag/drop, and clipboard

Goal - Learn how to open/save files and pick folders using the platform Storage Provider - Learn safe patterns for reading and writing files asynchronously - Add drag-and-drop support to accept files and text, and to start a drag from your app - Use the clipboard to copy, cut, and paste text and richer data

Why this matters - Every real app moves data in and out: import/export, user selections, assets, logs - Users expect familiar OS-native pickers, drag-and-drop, and clipboard behavior - Avalonia provides a single API that works across Windows, macOS, Linux, Android, iOS, and the Browser

Quick start: pick a file and read text 1) Get the storage provider from a TopLevel (Window, control, etc.) 2) Show the Open File Picker 3) Read the selected file using a stream

C#

```
using Avalonia;
using Avalonia.Controls;
using Avalonia.Platform.Storage;
using System.IO;
using System.Text;

public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    private async void OnOpenTextFile(object? sender, Avalonia.Interactivity.RoutedEventArgs e)
    {
        var sp = this.StorageProvider; // same as TopLevel.GetTopLevel(this)!.StorageProvider

        var files = await sp.OpenFilePickerAsync(new FilePickerOpenOptions
        {
            Title = "Open a text file",
            AllowMultiple = false,
            FileTypeFilter = new[]
            {
                new FilePickerFileType("Text files") { Patterns = new [] { "*.txt", "*.log" } },
                FilePickerFileTypes.All
            }
        });

        var file = files?.Count > 0 ? files[0] : null;
        if (file is null)
            return;

        // Safe async read
        await using var stream = await file.OpenReadAsync();
        using var reader = new StreamReader(stream, Encoding.UTF8, leaveOpen: false);
        var text = await reader.ReadToEndAsync();
        // TODO: show text in your UI
    }
}
```

Saving a file - Use SaveFilePickerAsync to ask for the target path and name - You can suggest a default file

name and extension

```
var sp = this.StorageProvider;
var sf = await sp.SaveFilePickerAsync(new FilePickerSaveOptions
{
    Title = "Save report",
    SuggestedFileName = "report.txt",
    DefaultExtension = "txt",
    FileTypeChoices = new[]
    {
        new FilePickerFileType("Text") { Patterns = new[] { "*.txt" } },
        new FilePickerFileType("Markdown") { Patterns = new[] { "*.md" } }
    }
});
if (sf is not null)
{
    await using var dst = await sf.OpenWriteAsync();
    await using var writer = new StreamWriter(dst, Encoding.UTF8, leaveOpen: false);
    await writer.WriteAsync("Hello from Avalonia!\n");
}
```

Pick multiple files - Set AllowMultiple = true - You'll get IReadOnlyList

```
var files = await this.StorageProvider.OpenFilePickerAsync(new FilePickerOpenOptions
{
    Title = "Pick images",
    AllowMultiple = true,
    FileTypeFilter = new[]
    {
        new FilePickerFileType("Images") { Patterns = new [] { "*.png", "*.jpg", "*.jpeg", "*.gif", "*.
    }
});
```

Pick a folder and enumerate items - Use OpenFolderPickerAsync to choose one or more folders - Enumerate files/folders via IStorageFolder.GetItemsAsync

```
var folders = await this.StorageProvider.OpenFolderPickerAsync(new FolderPickerOpenOptions
{
    Title = "Pick a folder",
    AllowMultiple = false
});
var folder = folders?.Count > 0 ? folders[0] : null;
if (folder is not null)
{
    var items = await folder.GetItemsAsync(); // files and subfolders
    foreach (var item in items)
    {
        // item is IStorageItem; you can check if it's a file or folder
        if (item is IStorageFile f)
        {
            // use f.OpenReadAsync / OpenWriteAsync
        }
        else if (item is IStorageFolder d)
        {
            // recurse or display
        }
```

```
    }
}
```

Access well-known folders - Some platforms expose Desktop, Documents, Downloads, Music, Pictures, Videos - Ask via TryGetWellKnownFolderAsync; it returns null if not available

```
var pictures = await this.StorageProvider.TryGetWellKnownFolderAsync(WellKnownFolder.Pictures);
if (pictures is not null)
{
    var items = await pictures.GetItemsAsync();
    // …
}
```

Safe file IO patterns - Always use async APIs (OpenReadAsync/OpenWriteAsync) to avoid blocking the UI thread - Wrap streams in using/await using to ensure disposal - Prefer UTF-8 unless you must match a specific encoding - Consider cancellation (pass CancellationToken if available in your app flow)

Drag-and-drop: accept files and text 1) Enable AllowDrop on the target control or container 2) Handle DragOver to indicate allowed effects 3) Handle Drop to read IDataObject content

XAML

```
<Border AllowDrop="True"
        DragOver="OnDragOver"
        Drop="OnDrop"
        BorderThickness="2" BorderBrush="Gray" Padding="16">
    <TextBlock Text="Drop files or text here"/>
</Border>
```

C#

```
using Avalonia.Input;
using Avalonia.Platform.Storage;

private void OnDragOver(object? sender, DragEventArgs e)
{
    // Advertise the effect based on available data
    if (e.Data.Contains(DataFormats.Files) || e.Data.Contains(DataFormats.Text))
        e.DragEffects = DragDropEffects.Copy;
    else
        e.DragEffects = DragDropEffects.None;
}

private async void OnDrop(object? sender, DragEventArgs e)
{
    // Files (as IStorageItem list)
    var storageItems = await e.Data.GetFilesAsync();
    if (storageItems is not null)
    {
        foreach (var item in storageItems)
        {
            if (item is IStorageFile file)
            {
                await using var s = await file.OpenReadAsync();
                // read or import
            }
        }
        return;
```

```
    }

    // Or plain text
    if (e.Data.Contains(DataFormats.Text))
    {
        var text = await e.Data.GetTextAsync();
        // use text
    }
}
```

Start a drag from your app - Build an IDataObject and call DragDrop.DoDragDrop from a pointer event handler - Choose the allowed effects (Copy/Move/Link)

```
using Avalonia.Input;
using Avalonia;

private async void OnPointerPressed(object? sender, PointerPressedEventArgs e)
{
    var data = new DataObject();
    data.Set(DataFormats.Text, "Dragged text from my app");
    var result = await DragDrop.DoDragDrop(e, data, DragDropEffects.Copy | DragDropEffects.Move);
    // result tells you what happened
}
```

Notes on drag-and-drop - Use e.KeyModifiers in DragOver to adjust effects (e.g., Ctrl for copy) - IDataObject supports multiple formats; you can set text, files, custom types (within process) - For file drags, many platforms supply virtual files that you read via IStorageFile stream APIs

Clipboard basics - Access the clipboard via this.Clipboard or Application.Current.Clipboard from a TopLevel - Get/Set text and richer data via IDataObject

```
var clipboard = this.Clipboard; // TopLevel clipboard
await clipboard.SetTextAsync("Hello clipboard");
var text = await clipboard.GetTextAsync();
```

Advanced clipboard - Set a full IDataObject (e.g., text + HTML + custom in-process data) - List available formats with GetFormatsAsync - Clear the clipboard with ClearAsync; use FlushAsync on platforms that support it to persist after exit

```
var dobj = new DataObject();

dobj.Set(DataFormats.Text, "plain");
dobj.Set("text/html", "<b>bold</b>");

await this.Clipboard.SetDataObjectAsync(dobj);
var formats = await this.Clipboard.GetFormatsAsync();
```

Cross-platform notes and limitations - Desktop (Windows/macOS/Linux): Full-featured pickers, drag-and-drop, and clipboard - Mobile (Android/iOS): Pickers use native UI; file system sandboxes and permissions apply - Browser (WASM): Pickers and clipboard require user gestures; not all formats are available; drag-and-drop limited to browser capabilities - SystemDialog APIs are obsolete; use TopLevel.StorageProvider for dialogs

Troubleshooting - If StorageProvider is null, ensure you're calling it from a control attached to the visual tree (after the Window is opened) - For drag-and-drop not firing, confirm AllowDrop=True and that handlers are attached on the element under the pointer - Clipboard failures on the browser usually mean missing user gesture or permissions - File filters are hints; some platforms may still let the user choose other files

Check yourself - Add a button that opens a text file and displays its contents in a TextBox - Add a Save

button that writes the TextBox contents to a user-chosen file - Enable drag-and-drop of one or more files; count them and list their names - Add Copy/Paste buttons that use IClipboard to copy and paste TextBox text

Extra practice - Add a filter to allow only ".csv" and ".xlsx" files - Drag data from your app (text) into another app; observe the effect result - Copy HTML to the clipboard and verify how different platforms paste it - Use TryGetWellKnownFolderAsync to show user pictures and let them pick one

Look under the hood - IStorageProvider interface (open/save/folder pickers): Avalonia.Base/Platform/Storage/IStorageProvi
- File/folder items: IStorageFile, IStorageFolder: Avalonia.Base/Platform/Storage/FileIO/IStorageFile.cs and Avalonia.Base/Platform/Storage/FileIO/IStorageFolder.cs - Picker options and filters: FilePickerOpenOptions, FilePickerSaveOptions, FilePickerFileType, FilePickerFileTypes: Avalonia.Base/Platform/Storage/FilePickerOpenOpt and Avalonia.Base/Platform/Storage/FilePickerSaveOptions.cs and Avalonia.Base/Platform/Storage/FilePickerFileType.cs - WellKnownFolder enum: Avalonia.Base/Platform/Storage/WellKnownFolder.cs - DragDrop APIs and events: Avalonia.Base/Input/DragDrop.cs - IDataObject and formats: Avalonia.Base/Input/IDataObject.cs and Avalonia.Base/Input/DataFormats.cs - Clipboard interface: Avalonia.Base/Platform/IClipboard.cs - TextBox clipboard events (copy/cut/paste hooks): Avalonia.Controls/TextBox.cs

What's next - Next: Chapter 17

17. Background work and networking

Goal - Run long operations without freezing the UI using async/await - Report progress and support cancel for a great UX - Make simple and reliable HTTP calls (GET/POST) and download files with progress

Why this matters - Real apps load data, process files, and talk to web services - Users expect responsive UIs with a spinner or progress, and the option to cancel - Async-first code is simpler, safer, and scales across desktop, mobile, and browser

Understand the UI thread - Avalonia has a UI thread that must update visuals and properties for UI elements - Keep the UI thread free: use async I/O or move CPU-heavy work to a background thread - Use Dispatcher.UIThread to marshal back to UI when you need to update visuals from background code

Quick start: run background work safely C#

```
using Avalonia.Threading;

private async Task<int> CountToAsync(int limit, CancellationToken ct)
{
    var count = 0;
    // Simulate CPU-bound work; for real CPU-heavy tasks, consider Task.Run
    for (int i = 0; i < limit; i++)
    {
        ct.ThrowIfCancellationRequested();
        await Task.Delay(1, ct); // non-blocking wait
        count++;
        if (i % 100 == 0)
        {
            // Update the UI safely
            await Dispatcher.UIThread.InvokeAsync(() =>
            {
                StatusText = $"Working... {i}/{limit}"; // assume a property that notifies
            });
        }
    }
    return count;
}
```

Progress reporting with IProgress - Prefer IProgress to decouple work from UI - Updates are automatically marshaled to the captured context when created on UI thread

```
public async Task ProcessAsync(IProgress<double> progress, CancellationToken ct)
{
    var total = 1000;
    for (int i = 0; i < total; i++)
    {
        ct.ThrowIfCancellationRequested();
        await Task.Delay(1, ct);
        progress.Report((double)i / total);
    }
}

// Usage from UI (e.g., ViewModel or code-behind)
var cts = new CancellationTokenSource();
var progress = new Progress<double>(p =>
{
    ProgressValue = p * 100; // 0..100 for ProgressBar
```

```
});
await ProcessAsync(progress, cts.Token);
```

Bind a ProgressBar XAML

```
<StackPanel Spacing="8">
  <ProgressBar Minimum="0" Maximum="100" Value="{Binding ProgressValue}" IsIndeterminate="{Binding IsBu
  <TextBlock Text="{Binding StatusText}"/>
  <StackPanel Orientation="Horizontal" Spacing="8">
    <Button Content="Start" Command="{Binding StartCommand}"/>
    <Button Content="Cancel" Command="{Binding CancelCommand}"/>
  </StackPanel>
</StackPanel>
```

ViewModel (simplified)

```
public class WorkViewModel : INotifyPropertyChanged
{
    private double _progressValue;
    private bool _isBusy;
    private string? _statusText;
    private CancellationTokenSource? _cts;

    public double ProgressValue { get => _progressValue; set { _progressValue = value; OnPropertyChange
    public bool IsBusy { get => _isBusy; set { _isBusy = value; OnPropertyChanged(); } }
    public string? StatusText { get => _statusText; set { _statusText = value; OnPropertyChanged(); } }

    public ICommand StartCommand => new RelayCommand(async _ => await StartAsync(), _ => !IsBusy);
    public ICommand CancelCommand => new RelayCommand(_ => _cts?.Cancel(), _ => IsBusy);

    private async Task StartAsync()
    {
        IsBusy = true;
        _cts = new CancellationTokenSource();
        var progress = new Progress<double>(p => ProgressValue = p * 100);
        try
        {
            StatusText = "Starting...";
            await ProcessAsync(progress, _cts.Token);
            StatusText = "Done";
        }
        catch (OperationCanceledException)
        {
            StatusText = "Canceled";
        }
        finally
        {
            IsBusy = false;
            _cts = null;
        }
    }

    // Example background work
    private static async Task ProcessAsync(IProgress<double> progress, CancellationToken ct)
    {
        const int total = 1000;
```

```
        for (int i = 0; i < total; i++)
        {
            ct.ThrowIfCancellationRequested();
            await Task.Delay(2, ct);
            progress.Report((double)(i + 1) / total);
        }
    }

    // INotifyPropertyChanged helper omitted for brevity
}
```

Networking basics: HttpClient - Use a single HttpClient for your app or feature area - Prefer async methods: GetAsync, PostAsync, ReadAsStreamAsync, ReadFromJsonAsync - Handle timeouts and cancellation; check IsSuccessStatusCode

Fetch JSON

```
using System.Net.Http;
using System.Net.Http.Json;

private static readonly HttpClient _http = new HttpClient
{
    Timeout = TimeSpan.FromSeconds(30)
};

public async Task<MyDto?> LoadDataAsync(CancellationToken ct)
{
    using var resp = await _http.GetAsync("https://example.com/api/data", ct);
    resp.EnsureSuccessStatusCode();
    return await resp.Content.ReadFromJsonAsync<MyDto>(cancellationToken: ct);
}

public record MyDto(string Name, int Count);
```

Post JSON

```
public async Task SaveDataAsync(MyDto dto, CancellationToken ct)
{
    using var resp = await _http.PostAsJsonAsync("https://example.com/api/data", dto, ct);
    resp.EnsureSuccessStatusCode();
}
```

Download a file with progress

```
public async Task DownloadAsync(Uri url, IStorageFile destination, IProgress<double> progress, Cancella
{
    using var resp = await _http.GetAsync(url, HttpCompletionOption.ResponseHeadersRead, ct);
    resp.EnsureSuccessStatusCode();

    var contentLength = resp.Content.Headers.ContentLength;
    await using var httpStream = await resp.Content.ReadAsStreamAsync(ct);
    await using var fileStream = await destination.OpenWriteAsync();

    var buffer = new byte[81920];
    long totalRead = 0;
    int read;
    while ((read = await httpStream.ReadAsync(buffer.AsMemory(0, buffer.Length), ct)) > 0)
```

```
    {
        await fileStream.WriteAsync(buffer.AsMemory(0, read), ct);
        totalRead += read;
        if (contentLength.HasValue)
        {
            progress.Report((double)totalRead / contentLength.Value);
        }
    }
}
```

UI threading tips - Never block with .Result or .Wait() on async tasks; await them instead - For CPU-heavy work, wrap synchronous code in Task.Run and report progress back via IProgress - Use Dispatcher.UIThread.InvokeAsync/Post to update UI from background threads if you didn't use IProgress

Cross-platform notes - Desktop: Full threading available; async/await with HttpClient works as expected - Mobile (Android/iOS): Add network permissions as required by the platform; background tasks may be throttled when app is suspended - Browser (WebAssembly): HttpClient uses fetch under the hood; CORS applies; sockets and some protocols may not be available; avoid long blocking loops

Troubleshooting - UI freeze? Look for synchronous waits (.Result/.Wait) or blocking I/O on UI thread - Progress not updating? Ensure property change notifications fire and bindings are correct - Networking errors? Check HTTPS, certificates, CORS (in browser), and timeouts - Cancel not working? Pass the same CancellationToken to all async calls in the operation

Check yourself - Add a Start button that runs a fake task for 5 seconds and updates a ProgressBar - Add a Cancel button that stops the task midway and sets a status message - Load a small JSON from a public API and show one field in the UI - Download a file to disk and show progress from 0 to 100

Extra practice - Wrap a CPU-intensive calculation with Task.Run and report progress - Add retry with exponential backoff around a flaky HTTP call - Let the user pick a destination file (using Chapter 16's SaveFilePicker) for the downloader

Look under the hood - Avalonia UI thread and dispatcher: Avalonia.Base/Threading/Dispatcher.cs - ProgressBar control: Avalonia.Controls/ProgressBar.cs - Binding basics (see binding implementation): Markup/Avalonia.Markup

What's next - Next: Chapter 18

18. Desktop targets: Windows, macOS, Linux

Goal: Give you a practical map of Avalonia's desktop features across Windows, macOS, and Linux, focusing on windowing, system decorations, transparency, multi-monitor support, and scaling.

Why this matters: Desktop apps live in windows. Understanding how to size, position, decorate, and move windows (and how that varies by platform) prevents many bugs and gives your app a polished, native feel.

What you'll learn - Window basics you'll use all the time: state, size-to-content, resizability, startup location, topmost/taskbar - System decorations vs custom chrome (client area extension), and safe drag/resize - Transparency levels (blur, acrylic, mica) and how to use them safely - Multiple monitors and DPI scaling with Screens and DesktopScaling/RenderScaling - Platform differences and troubleshooting tips

1) Window basics

- Window state and resizability
  - WindowState: Minimized, Normal, Maximized, FullScreen
  - CanResize: whether the user can resize the window
  - SizeToContent: Manual, Width, Height, WidthAndHeight
- Show in taskbar and always-on-top
  - ShowInTaskbar: show/hide the taskbar or dock icon
  - Topmost: keep the window above others
- Startup position
  - WindowStartupLocation: Manual (default), CenterScreen, CenterOwner

Example (XAML):

```
<Window
    xmlns="https://github.com/avaloniaui"
    x:Class="MyApp.MainWindow"
    Width="960" Height="640"
    CanResize="True"
    SizeToContent="Manual"
    WindowStartupLocation="CenterScreen"
    ShowInTaskbar="True"
    Topmost="False">
    <!-- Content here -->
</Window>
```

Example (code-behind):

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();

        WindowState = WindowState.Normal;
        CanResize = true;
        SizeToContent = SizeToContent.Manual;
        WindowStartupLocation = WindowStartupLocation.CenterScreen;
        ShowInTaskbar = true;
        Topmost = false;
    }
}
```

2) System decorations and custom chrome You can let the OS draw the standard window frame and title bar, or draw a custom one.

- SystemDecorations: Full (default) or None
- Extend client area into the title bar area to build custom chrome:
  - ExtendClientAreaToDecorationsHint (bool)
  - ExtendClientAreaChromeHints (flags)
  - ExtendClientAreaTitleBarHeightHint (double)

Safe drag/resize - BeginMoveDrag(PointerPressedEventArgs) to let users drag your custom title bar - BeginResizeDrag(WindowEdge, PointerPressedEventArgs) to let users resize by grabbing your custom edges

Minimal custom title bar example:

```
<Window
    xmlns="https://github.com/avaloniaui"
    x:Class="MyApp.MainWindow"
    SystemDecorations="None"
    ExtendClientAreaToDecorationsHint="True"
    ExtendClientAreaChromeHints="PreferSystemChrome"
    ExtendClientAreaTitleBarHeightHint="30">
    <Border Background="#1F1F1F" Height="30"
            PointerPressed="TitleBar_OnPointerPressed">
        <StackPanel Orientation="Horizontal" VerticalAlignment="Center" Margin="8,0">
            <TextBlock Text="My App" Foreground="White"/>
            <!-- Add your own caption buttons here -->
        </StackPanel>
    </Border>
</Window>
```

```
private void TitleBar_OnPointerPressed(object? sender, PointerPressedEventArgs e)
{
    // Only start a drag on left button press
    if (e.GetCurrentPoint(this).Properties.IsLeftButtonPressed)
        BeginMoveDrag(e);
}
```

Notes - Keep accessibility in mind: ensure hit-targets and tooltips for custom caption buttons. - Use your theme's resources for hover/pressed states.

3) Transparency levels (blur, acrylic, mica) Avalonia exposes a cross-platform transparency abstraction on TopLevel (Window derives from TopLevel).

- Set a preferred list via TransparencyLevelHint (ordered by preference):
  - WindowTransparencyLevel.None
  - WindowTransparencyLevel.Transparent
  - WindowTransparencyLevel.Blur
  - WindowTransparencyLevel.AcrylicBlur
  - WindowTransparencyLevel.Mica
- Read the achieved level via ActualTransparencyLevel (platform picks the first supported one).

Example:

```
<Window
    xmlns="https://github.com/avaloniaui"
    x:Class="MyApp.MainWindow"
    TransparencyLevelHint="AcrylicBlur, Blur, Transparent">
    <!-- Content -->
</Window>
```

```
public MainWindow()
{
```

```
    InitializeComponent();
    TransparencyLevelHint = new[]
    {
        WindowTransparencyLevel.AcrylicBlur,
        WindowTransparencyLevel.Blur,
        WindowTransparencyLevel.Transparent
    };

    this.GetObservable(TopLevel.ActualTransparencyLevelProperty)
        .Subscribe(level => Debug.WriteLine($"Actual transparency: {level}"));
}
```

Platform support summary (typical) - Windows: Transparent, AcrylicBlur, Mica - macOS: Transparent, AcrylicBlur - Linux (X11 + compositor): Transparent, Blur - Headless/Browser: None (not applicable for this chapter)

Tip - Always provide a fallback chain (e.g., Mica → AcrylicBlur → Blur → Transparent) and design your theme to look good even with None.

4) Multiple monitors and scaling Desktop apps must handle multiple displays and DPI scaling.

- Screens: enumerate and query monitors
  - this.Screens.All, this.Screens.Primary
  - this.Screens.ScreenFromPoint(PixelPoint), ScreenFromWindow(Window), ScreenFromBounds(PixelRect)
- Screen.Scaling: system DPI scaling factor for that monitor
- Window/Desktop scaling vs render scaling
  - Window.DesktopScaling: used for window positioning/sizing
  - TopLevel.RenderScaling: used for rendering primitives

Center the window on the primary screen in code:

```
protected override void OnOpened(EventArgs e)
{
    base.OnOpened(e);

    var screen = Screens?.Primary;
    if (screen is null)
        return;

    // Convert logical size to pixel size using DesktopScaling
    var frame = PixelRect.FromBounds(new PixelPoint(0, 0),
        PixelSize.FromSize(ClientSize, DesktopScaling));

    var target = screen.WorkingArea.CenterRect(frame.Size);

    Position = target.Position;
}
```

React to scaling changes (e.g., when moving between monitors):

```
ScalingChanged += (_, __) =>
{
    // RenderScaling changed; update sizes or pixel perfect resources if needed
};
```

5) Fullscreen, z-order, and window interactions

- Fullscreen: set WindowState = WindowState.FullScreen; toggle back to Normal to exit
- Topmost: keep the window on top of others

- ShowDialog(owner): open modal child windows centered on the owner (see Chapter 12)
- BeginResizeDrag: implement resize handles in custom chrome

6) Platform notes and differences Windows

- Transparency: AcrylicBlur and Mica are available on supported OS versions; Transparent works broadly
- System decorations: rich control; ExtendClientArea recommended for custom title bars
- Taskbar and z-order: ShowInTaskbar and Topmost are fully supported

macOS - Transparency: Transparent and Acrylic-style blur supported via the native compositor - Title area: ExtendClientAreaTitleBarHeightHint lets you align custom content; keep native feel - Taskbar (Dock): ShowInTaskbar maps to Dock visibility semantics

Linux (X11) - Transparency: Transparent and Blur depend on the window manager/compositor (e.g., GNOME, KDE) - Decorations: behavior can vary by WM; test with SystemDecorations=None + ExtendClientArea - Scaling: fractional scaling support depends on the environment; verify RenderScaling at runtime

7) Troubleshooting

- Window looks blurry on high-DPI displays
  - Ensure images/icons are vector or have multiple raster scales; read RenderScaling to pick assets
- Transparency request ignored
  - Check ActualTransparencyLevel; fall back gracefully
- Custom title bar dragging doesn't work
  - Call BeginMoveDrag only on a left-button press; don't start a drag from interactive children
- Window opens on the wrong monitor
  - Set WindowStartupLocation to CenterScreen for primary screen; or compute a position using Screens

Look under the hood (source) - Window: Avalonia.Controls/Window.cs - WindowStartupLocation: Avalonia.Controls/WindowStartupLocation.cs - WindowState: Avalonia.Controls/WindowState.cs - SystemDecorations and extend-client-area hints: Window.cs (L100–L161) - TopLevel transparency properties: TopLevel.cs (L69–L86) - WindowTransparencyLevel values: Avalonia.Controls/WindowTransparencyLevel.cs - Screens and Screen: Avalonia.Controls/Screens.cs and Avalonia.Controls/Platform/Screen.cs

Check yourself - Can you toggle between Normal, Maximized, and FullScreen at runtime? - Can you build a minimal custom title bar that supports drag and window state buttons? - Can you request Mica/Acrylic and fall back to Transparent when not supported? - Can you query the current Screen and move the window to its center?

Extra practice - Add a "Move To Next Monitor" command that cycles the window through Screens.All - Create a theme that visually adapts to ActualTransparencyLevel - Implement resize handles around your custom chrome using BeginResizeDrag

What's next - Next: Chapter 19

19. Mobile targets: Android and iOS

Goal - Build and run Avalonia apps on Android and iOS - Understand SingleView lifetime and what's different from desktop - Learn mobile-friendly navigation, input, insets/safe areas, and soft keyboard handling

Why this matters Desktop habits don't directly map to phones. On mobile there's no multi-window UI, touch is the primary input, and the OS controls system bars and safe areas. Small, intentional patterns keep your app feeling native while staying 100% Avalonia.

Quick start: SingleView lifetime (the mobile way) On Android and iOS, Avalonia apps use a single top-level view instead of windows. In your App class, assign MainView when the application runs with a single-view lifetime.

C# (App)

```
public partial class App : Application
{
    public override void OnFrameworkInitializationCompleted()
    {
        if (ApplicationLifetime is ISingleViewApplicationLifetime singleView)
        {
            singleView.MainView = new MainView
            {
                DataContext = new MainViewModel()
            };
        }
        else if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
        {
            // Desktop fallback so the same app runs everywhere
            desktop.MainWindow = new MainWindow
            {
                DataContext = new MainViewModel()
            };
        }

        base.OnFrameworkInitializationCompleted();
    }
}
```

XAML (MainView)

```
<UserControl xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             x:Class="MyApp.Views.MainView">
  <StackPanel Spacing="12" Margin="16">
    <TextBlock Text="Hello mobile" FontSize="24"/>
    <Button Content="Tap me" Command="{Binding TapCommand}"/>
  </StackPanel>
</UserControl>
```

Run targets - Android: build and deploy to an emulator or device via the Android head project - iOS: build and deploy to the Simulator or device via the iOS head project Note: The platform "head" projects host your shared Avalonia app and provide platform manifests, icons, and entitlements.

Navigation on mobile (no windows, just views) On phones you typically show one screen at a time and navigate forward/back. Two simple patterns:

1) Swap views in a ContentControl

- Keep a navigation stack (List or view models) and set a ContentControl's Content to the current view
- Provide back/forward commands that push/pop the stack

2) Router-like approach

- You can implement a lightweight router (string or enum for routes → factory method producing a view)
- Or use a routing library (e.g., ReactiveUI's routing) if you already use ReactiveUI

Minimal example (content swap)

```
public class NavService
{
    private readonly Stack<object> _stack = new();

    public object? Current { get; private set; }

    public void NavigateTo(object vm)
    {
        if (Current is not null)
            _stack.Push(Current);
        Current = vm;
        OnChanged?.Invoke();
    }

    public bool GoBack()
    {
        if (_stack.Count == 0)
            return false;
        Current = _stack.Pop();
        OnChanged?.Invoke();
        return true;
    }

    public event Action? OnChanged;
}
```

Bind a ContentControl to NavService.Current and provide a Back button in your UI. On Android, the system Back button will usually close the activity if you don't intercept it—so expose a Back command and wire it from your head project if you need to consume system back instead of exiting. On iOS, users expect a visible Back affordance in-app.

Touch input and gestures - Tapped/DoubleTapped events work well for touch-first interaction - Pointer-Pressed/PointerReleased/PointerMoved are available when you need fine-grained control - Avoid hover-only affordances; ensure controls are large enough to tap comfortably (44×44dp+)

Soft keyboard (IInputPane) and layout When the on-screen keyboard appears, your UI may need to move or resize elements so inputs aren't obscured. Subscribe to the input pane notifications and adjust paddings/margins accordingly.

```
public partial class LoginView : UserControl
{
    public LoginView()
    {
        InitializeComponent();
        this.AttachedToVisualTree += (_, __) =>
        {
            var tl = TopLevel.GetTopLevel(this);
```

```
            var pane = tl?.InputPane;
            if (pane is null) return;

            pane.Showing += (_, __) => MoveContentUp();
            pane.Hiding += (_, __) => ResetLayout();
        };
    }
}
```

Safe areas and cutouts (IInsetsManager) Modern phones have notches and system bars. Respect safe areas by adding padding from the insets manager and updating when insets change.

```
this.AttachedToVisualTree += (_, __) =>
{
    var tl = TopLevel.GetTopLevel(this);
    var insets = tl?.InsetsManager;
    if (insets is null) return;

    void Apply() => RootPanel.Padding = new Thickness(
        left: insets.SafeAreaPadding.Left,
        top: insets.SafeAreaPadding.Top,
        right: insets.SafeAreaPadding.Right,
        bottom: insets.SafeAreaPadding.Bottom);

    Apply();
    insets.Changed += (_, __) => Apply();
};
```

Resources and assets for mobile - Prefer vectors (Path/Icon) for crisp results at any DPI - If you ship bitmaps, keep them reasonably sized; Avalonia scales device-independently but huge images still cost memory - Fonts work the same as desktop (embed and reference by FontFamily); verify legibility on small screens - App icons, splash screens, and entitlements live in the platform head projects (Android/iOS)

Storage and permissions - Use StorageProvider for user file picks; don't assume open file system access on mobile - Android and iOS enforce permission models; requests and declarations live in the platform head (AndroidManifest.xml / Info.plist)

Platform differences at a glance - Android: one activity hosts the app; hardware Back can exit unless handled; navigation/status bars vary by device/theme - iOS: status bar and home indicator define safe areas; Back is typically an in-app control; background execution is more restricted

Troubleshooting - Emulator/simulator doesn't start: confirm SDKs, device images, and architecture match your machine - App immediately exits on Android when pressing Back: your navigation stack returned false; provide an in-app Back or intercept system back in the head project - Keyboard covers inputs: handle IInputPane showing/hiding and adjust layout - Content under status bar or notch: apply padding from IInsetsManager safe area

Exercise Convert your desktop sample to mobile: 1) Create a MainView that fits on a phone screen and set it as MainView for ISingleViewApplicationLifetime 2) Introduce a simple NavService and two screens (List → Details) with a visible Back button 3) Handle IInputPane to keep login inputs visible when the keyboard appears 4) Add safe area padding via IInsetsManager

Look under the hood - Lifetimes: ISingleViewApplicationLifetime and SingleViewApplicationLifetime Avalonia.Controls/ApplicationLifetimes - Input pane abstraction (soft keyboard): IInputPane Avalonia.Controls/Platform/IInputPane.cs - Insets/safe areas: IInsetsManager Avalonia.Controls/Platform/IInsetsManager.cs - Platform heads and samples Android: src/Android | sample heads under samples/ iOS: src/iOS | sample heads under samples/

What's next - Next: Chapter 20

20. Browser (WebAssembly) target

Goal - Build and run your Avalonia app in the browser using WebAssembly - Understand startup with StartBrowserAppAsync and the single-view lifetime - Choose rendering modes (WebGL2/WebGL1/Software2D) and know web-specific limits

Why this matters Running in the browser lets you reuse your UI and logic without installing native apps. It's perfect for demos, admin screens, and tools. The browser has different rules (security, file access, multi-window) and you'll make better design choices if you know them.

Quick start: StartBrowserAppAsync In the browser, Avalonia runs with a single-view lifetime and renders into a specific HTML element by id. The simplest startup creates and attaches a view for you.

Program.cs

```
using Avalonia;
using Avalonia.Browser;

internal class Program
{
    private static AppBuilder BuildAvaloniaApp()
        => AppBuilder.Configure<App>()
            .UsePlatformDetect()
            .LogToTrace();

    public static Task Main(string[] args)
        => BuildAvaloniaApp()
            .StartBrowserAppAsync("out"); // attaches to <div id="out"></div>
}
```

HTML host (conceptual)

```
<body>
  <div id="out"></div>
</body>
```

Note: In real projects the template sets up the host page and static assets for you. The important part is that an element with id=“out” exists.

Rendering modes and options You can pick renderers and configure web-specific behaviors via BrowserPlatformOptions.

```
await BuildAvaloniaApp().StartBrowserAppAsync(
    "out",
    new BrowserPlatformOptions
    {
        // Try WebGL2, then WebGL1, then fallback to Software2D
        RenderingMode = new[]
        {
            BrowserRenderingMode.WebGL2,
            BrowserRenderingMode.WebGL1,
            BrowserRenderingMode.Software2D
        },

        // Register a service worker used for save-file polyfill (optional)
        RegisterAvaloniaServiceWorker = true,
        AvaloniaServiceWorkerScope = "/",
```

```
        // Force using the file dialog polyfill even if native is available
        PreferFileDialogPolyfill = false,

        // Use a managed dispatcher on a worker thread when WASM threads are enabled
        PreferManagedThreadDispatcher = true,
    });
```

- RenderingMode: a priority list, first supported value wins (best performance: WebGL2).
- RegisterAvaloniaServiceWorker/AvaloniaServiceWorkerScope: enables a service worker used by the save file polyfill on browsers without a native File System Access API.
- PreferFileDialogPolyfill: forces use of the "native-file-system-adapter" polyfill even if a native API is present.
- PreferManagedThreadDispatcher: when WASM threads are enabled, run the dispatcher on a worker thread for responsiveness.

Alternative: SetupBrowserAppAsync (advanced) SetupBrowserAppAsync loads the browser backend without creating a view. This is useful for custom embedding scenarios; most apps should use StartBrowserAppAsync.

Single view lifetime on the web Avalonia uses ISingleViewApplicationLifetime in the browser. In App.OnFrameworkInitializationCompleted, set MainView like you do for mobile:

```
public override void OnFrameworkInitializationCompleted()
{
    if (ApplicationLifetime is ISingleViewApplicationLifetime singleView)
    {
        singleView.MainView = new MainView
        {
            DataContext = new MainViewModel()
        };
    }

    base.OnFrameworkInitializationCompleted();
}
```

Storage and file dialogs in the browser - Use IStorageProvider for open/save/folder pickers. On supported browsers Avalonia uses the File System Access API; otherwise it uses a polyfill with a service worker to enable saving. - Browsers require a secure context (HTTPS or localhost) for advanced file APIs. Expect different UX than desktop.

Networking and CORS - Browser networking follows CORS rules. If your API doesn't set the right headers, requests can be blocked. - Use HTTPS and correct Access-Control-Allow-* headers on your server; the browser controls what's allowed, not Avalonia.

Platform capabilities and limitations - Windows & menus: Browser runs with a single view; native menus/tray icons, system dialogs, and OS integrations are not available. - Input and focus: Works with keyboard, mouse, touch; clipboard access is gated by browser rules and user gestures. - Graphics: WebGL2 is fastest; WebGL1 is a fallback; Software2D is a last resort and is slower. - Local files: You don't have broad file system access; always go through IStorageProvider. - Threads: WASM threads require explicit hosting support and appropriate headers; if unavailable the app runs single-threaded.

Blazor hosting option You can host Avalonia inside a Blazor app using Avalonia.Browser.Blazor. This is handy when you need existing Blazor routing/layout with embedded Avalonia UI. See the ControlCatalog.Browser.Blazor sample for a working project structure.

Troubleshooting - Blank page: Verify the div id matches StartBrowserAppAsync("…") and that the static assets are served. Check the browser console for module load errors. - WebGL errors or poor performance: Ensure your GPU/browser supports WebGL2; try WebGL1 fallback or Software2D. - Save file doesn't work: Enable the service worker option, serve over HTTPS/localhost, and verify the polyfill is allowed by the

browser. - CORS failures: Fix server headers; the browser blocks disallowed cross-origin requests.

Exercise 1) Add a browser head to your app and wire StartBrowserAppAsync("out"). 2) Configure RenderingMode to try WebGL2→WebGL1→Software2D and verify the app runs on at least two different browsers. 3) Implement an Export button using IStorageProvider to test the save polyfill with and without the service worker.

Look under the hood - Browser startup and options: BrowserAppBuilder Avalonia.Browser/BrowserAppBuilder.cs - Single view lifetime (browser): BrowserSingleViewLifetime Avalonia.Browser/BrowserSingleViewLifetime.cs - ControlCatalog browser sample (Program.cs) samples/ControlCatalog.Browser/Program.cs - Input/keyboard pane for browser Avalonia.Browser/BrowserInputPane.cs - Insets/safe areas for browser Avalonia.Browser/BrowserInsetsManager.cs - Storage provider and polyfill Avalonia.Browser/Storage/BrowserStorageProvider.cs - Platform settings (browser) Avalonia.Browser/BrowserPlatformSettings.cs - Blazor hosting Avalonia.Browser.Blazor

What's next - Next: Chapter 21

21. Headless and testing

Goal - Write fast, reliable UI tests that run on CI with no display server - Simulate user input (keyboard/mouse) and verify UI behavior programmatically - Capture and assert rendered frames for visual regression tests (optional)

Why this matters UI you can't test will regress. Avalonia's headless platform lets you run your app and controls without a window manager, so tests run anywhere (including CI) and stay deterministic.

What "headless" means in Avalonia - Headless is a special platform backend that implements windowing, input, and rendering without a real OS window. - You can drive your UI programmatically and even capture frames for pixel tests. - There are helpers for popular test frameworks (xUnit, NUnit) so you don't write plumbing.

Quick start (xUnit): [Avalonia.Headless.XUnit] 1) Add test packages to your test project: - Avalonia.Headless - Avalonia.Headless.XUnit - Optionally Avalonia.Skia for Skia rendering when you need screenshots 2) Use the [AvaloniaFact] attribute to run a test on the Avalonia UI thread with a headless platform.

Example: a minimal UI interaction test

```
using System.Threading.Tasks;
using Avalonia;
using Avalonia.Controls;
using Avalonia.Headless; // Headless helpers + [AvaloniaFact]
using Avalonia.Threading;
using Xunit;

public class TextBoxTests
{
    private static AppBuilder BuildApp() => AppBuilder.Configure<App>()
        .UseHeadless(new Avalonia.Headless.AvaloniaHeadlessPlatformOptions
        {
            // For logic-only tests, keep this true (fast, no Skia); for screenshot tests set to false
            UseHeadlessDrawing = true
        })
        .AfterSetup(_ => { /* put global test services if needed */ });

    [AvaloniaFact]
    public async Task TextBox_Receives_Typed_Text()
    {
        BuildApp();

        var textBox = new TextBox { Width = 200, Height = 30 };
        var window = new Window { Content = textBox };
        window.Show();

        // Focus and type via headless helpers
        await Dispatcher.UIThread.InvokeAsync(() => textBox.Focus());
        window.KeyPress(Key.A, RawInputModifiers.Control, PhysicalKey.A, ""); // Ctrl+A (select all)
        window.TextInput("Hello");

        // Let layout/rendering advance one tick
        Avalonia.Headless.AvaloniaHeadlessPlatform.ForceRenderTimerTick();

        Assert.Equal("Hello", textBox.Text);
    }
}
```

Notes - BuildApp(): In test assemblies, the Headless runner auto-starts a session; you can configure extra services via AppBuilder as needed. - Input helpers: After `using Avalonia.Headless;`, extension methods like `KeyPress`, `KeyRelease`, `TextInput`, `MouseDown`, `MouseMove`, `MouseUp`, `MouseWheel`, and `DragDrop` are available on TopLevel/Window. - Rendering tick: Use `AvaloniaHeadlessPlatform.ForceRenderTimerTick()` to advance timers and trigger layout/render when needed.

Capturing rendered frames (visual regression) To capture frames you must render with Skia and disable headless drawing: - Call UseSkia() during setup - Pass UseHeadlessDrawing = false when using UseHeadless - Then use GetLastRenderedFrame() from HeadlessWindowExtensions

Example: capture a frame and assert size

```
using Avalonia;
using Avalonia.Controls;
using Avalonia.Headless;
using Xunit;

public class SnapshotTests
{
    private static AppBuilder BuildApp() => AppBuilder.Configure<App>()
        .UseSkia() // enable Skia
        .UseHeadless(new Avalonia.Headless.AvaloniaHeadlessPlatformOptions { UseHeadlessDrawing = false

    [AvaloniaFact]
    public void Window_Renders_Frame()
    {
        BuildApp();
        var window = new Window { Width = 300, Height = 200, Content = new Button { Content = "Click" }
        window.Show();

        // Make sure a render tick happens
        AvaloniaHeadlessPlatform.ForceRenderTimerTick();

        using var frame = window.GetLastRenderedFrame();
        Assert.NotNull(frame);
        Assert.Equal(300, frame.Size.Width);
        Assert.Equal(200, frame.Size.Height);
    }
}
```

Tip: You can persist frames to disk for debugging when running locally; for CI, prefer comparing against a baseline image with a small tolerance. Keep baselines per theme/DPI if relevant.

NUnit option - Use Avalonia.Headless.NUnit and the provided attributes/utilities (AvaloniaTheory, test wrappers) to initialize a HeadlessUnitTestSession for your assembly. - The patterns mirror xUnit; prefer your team's test framework.

Driving complex interactions - Pointer/mouse: `MouseDown(point, button, modifiers)`, `MouseMove(point, modifiers)`, `MouseUp(point, modifiers)` - Keyboard: `KeyPress`, `KeyRelease`, `TextInput` - Drag and drop: `DragDrop(point, type, data, effects, modifiers)` - Always focus the control first, and advance one tick afterward to flush input effects: `Focus()`, then `ForceRenderTimerTick()`

Dispatcher and async work in tests - Use `Dispatcher.UIThread.InvokeAsync` to execute code on the UI thread. - Use `await Task.Yield()` and a render tick to allow bindings and layout to settle. - Avoid unbounded waits; if you poll for a condition, cap attempts and fail with a helpful message.

Headless VNC (debug a headless app visually) - For app-level diagnostics (not typical for unit tests), you can run the Headless VNC platform and connect with a VNC client to see frames. - See ControlCatalog

sample: it supports `--vnc`/`--full-headless` switches and uses `StartWithHeadlessVncPlatform(...)` to boot an app with a VNC framebuffer.

What to test where - ViewModels: test without any Avalonia dependency (fastest); verify commands, properties, validation. - Controls/Views: use headless tests to simulate input and verify behavior/visuals. - Integration flows: a few end-to-end headless tests are valuable; keep them focused to avoid flakiness.

Troubleshooting - "TopLevel must be a headless window.": Ensure tests initialize the headless platform (UseHeadless) and that the TopLevel is created after setup. - "Frame is null" or empty: Call UseSkia + set UseHeadlessDrawing=false and ensure you tick the render timer. - Input does nothing: Ensure the control has focus and a render tick occurs after the simulated input. - Hanging tests: Never block the UI thread; prefer InvokeAsync + short waits and ticks.

Exercise 1) Write a test that types "Avalonia" into a TextBox via `TextInput` and asserts the text. 2) Add a Button with a command bound to a ViewModel; simulate a `MouseDown`/`MouseUp` to click it and assert the command executed. 3) Create a snapshot test that captures the frame of a 200×100 Border with a red background; assert the bitmap size and optionally compare with a baseline image.

Look under the hood - Headless platform entry: AvaloniaHeadlessPlatform - Rendering interface (headless stubs): HeadlessPlatformRenderInterface.cs - Simulated input and frame capture: HeadlessWindowExtensions - xUnit integration: Avalonia.Headless.XUnit - NUnit integration: Avalonia.Headless.NUnit - ControlCatalog example switches (VNC/headless): ControlCatalog.NetCore/Program.cs

What's next - Next: Chapter 22

22. Rendering pipeline in plain words

Goal - Understand how Avalonia turns your visual tree into pixels on screen - Know the core pieces: UI thread, render loop, renderer, compositor, Skia - Learn the few options you can safely tune (SkiaOptions, RenderOptions)

Why this matters - Performance and correctness: knowing what triggers redraws (and what doesn't) helps you write smooth, battery-friendly UI - Debugging: when frames don't appear, knowing who is responsible saves hours - Confidence: you'll recognize what is platform-specific and what is cross-platform by design

A simple mental model - You manipulate a tree of Visuals (Controls are Visuals) on the UI thread - Changes mark parts of the scene as "dirty" and schedule work on the render loop - The renderer converts visuals to draw calls (Skia commands) - The compositor coordinates sending updates to the render thread and presents frames to a window/surface - Skia draws into GPU textures or CPU bitmaps; the platform presents them to the screen

What runs where (threads) - UI thread: you create/update controls, styles, bindings, animations, and handle input - Render thread: receives serialized batches of composition changes and performs GPU/Skia work, then presents - Separation keeps input/UI responsive even if a heavy frame is rendering

Requesting and producing frames - Marking visuals dirty: controls call InvalidateVisual (protected) or update properties that affect rendering; the renderer's queue is notified - The renderer implements lifecycle methods: - AddDirty(Visual): a visual or region needs redraw - Resized(Size): target size changed - Paint(Rect): handle a paint request from the platform - Start()/Stop(): hook the render loop - SceneInvalidated event signals that low-level scene data changed (useful for input hit-testing state)

Skia at the core - Avalonia uses Skia for cross-platform drawing: shapes, text, images, effects - Skia can render using CPU or GPU; Avalonia prefers GPU when available - AppBuilder.UseSkia() enables the Skia backend; you can pass SkiaOptions for tuning

GPU backends at a glance - Avalonia abstracts GPU access with IPlatformGraphics; platform heads bind an available backend (OpenGL/ANGLE, Metal, Vulkan, etc.) - On Windows, macOS, Linux, Android, iOS, and Browser, Skia draws into a surface backed by the platform's GPU context or a software bitmap; the windowing system then presents the result - You don't choose the low-level API directly; you use UseSkia and optional platform options, and Avalonia picks the most appropriate graphics stack

Composition and presentation - The compositor coordinates updates between UI and render threads using batches; commits serialize object changes and send them to the render side - A render loop ticks at a platform-determined cadence; when there are dirty visuals or animations, a new frame is rendered and presented - Effects like opacity, transforms, and clips are applied while traversing the visual tree; platform composition APIs may assist with efficient presentation on some systems

Immediate vs. normal rendering - Normal application rendering uses the threaded compositor + Skia pipeline described above - ImmediateRenderer is a utility that walks a Visual subtree directly into a DrawingContext without the full presentation path (used by features like VisualBrush or RenderTargetBitmap) - Think of ImmediateRenderer as a synchronous "draw this once into a bitmap" tool, not the app's live render loop

Tuning Skia with UseSkia(SkiaOptions) - SkiaOptions.MaxGpuResourceSizeBytes (long?): caps Skia's GPU resource cache (textures, glyph atlases) - Defaults to a value suitable for typical apps; set null to let Skia decide; set lower to constrain memory, higher to reduce cache churn - SkiaOptions.UseOpacitySaveLayer (bool): forces use of Skia's SaveLayer for opacity handling - Can fix edge cases with nested opacity but may cost performance; leave off unless you need it - Example:

AppBuilder.Configure() .UsePlatformDetect() .UseSkia(new SkiaOptions { MaxGpuResourceSizeBytes = 64L * 1024 * 1024, // 64 MB UseOpacitySaveLayer = false }) .StartWithClassicDesktopLifetime(args);

RenderOptions: per-visual quality knobs - RenderOptions is a value applied per Visual and merged down the tree; it controls how bitmaps and text are sampled/blended - Properties you can set: - BitmapInterpolationMode: Unspecified, LowQuality/MediumQuality/HighQuality - BitmapBlendingMode: Unspecified or

a blend mode for images - EdgeMode: Antialias, Aliased, Unspecified (affects geometry edges and text defaults) - TextRenderingMode: Default, Antialias, SubpixelAntialias, Aliased - RequiresFullOpacityHandling: bool? (forces full opacity handling for complex compositions) - Use attached helpers:

// Make images crisper when scaled down RenderOptions.SetBitmapInterpolationMode(myImage, BitmapInterpolationMode.MediumQuality);

// Force aliased text on a small LED-style display RenderOptions.SetTextRenderingMode(myTextBlock, TextRenderingMode.Aliased);

What actually triggers redraws - Property changes that affect layout or appearance (e.g., Brush, Text, Bounds) mark visuals dirty - Animations and timers schedule continuous frames while active - Input and window resize generate paint/size events - Pure ViewModel changes trigger rendering only when they update bound UI properties

Practical tips - Prefer vector drawing and let RenderOptions/Interpolation control quality on scaled assets - Avoid layout thrash: batch property changes; let animations drive smooth frames instead of manual timers - Do image decoding/resizing off the UI thread; then set the final Bitmap on UI thread - Profile on the slowest target first; GPU availability and drivers vary across platforms

Troubleshooting - “Nothing updates until I interact”: ensure the app is started with a lifetime that runs a message loop and that you haven’t stopped the renderer; long-running work should be off the UI thread - “Blurry text or images”: adjust TextRenderingMode/EdgeMode and BitmapInterpolationMode as needed; check DPI settings - “High GPU memory”: tune MaxGpuResourceSizeBytes and use smaller images; free large bitmaps when not needed - “Opacity stacking looks wrong”: try SkiaOptions.UseOpacitySaveLayer = true for correctness, then measure

Look under the hood (selected source) - Renderer interface: IRenderer.cs (methods like AddDirty, Paint, Start/Stop) — Avalonia.Base/Rendering/IRenderer.cs - Immediate renderer utility — Avalonia.Base/Rendering/ImmediateRenderer.cs - Compositor (UI render threads, commit batches) — Avalonia.Base/Rendering/Composition/Compositor.cs - RenderOptions (bitmap/text/edge/blend/opacity) — Avalonia.Base/Media/RenderOptions.cs - Skia options (resource cache, opacity save layer) — Skia/Avalonia.Skia/SkiaOptions.cs - Skia render interface and GPU plumbing — Skia/Avalonia.Skia/PlatformRenderInterface - Platform GPU abstraction (IPlatformGraphics) — Avalonia.Base/Platform/IPlatformGpu.cs

Exercise - Create a small page with an Image and a TextBlock. Try these: 1) Set BitmapInterpolationMode to LowQuality, then HighQuality while scaling the image; observe differences 2) Toggle TextRenderingMode between Antialias and Aliased on small font sizes; note readability 3) Start the app with UseSkia(new SkiaOptions { UseOpacitySaveLayer = true }) and layer two semi-transparent panels; compare visuals and measure frame time on an animated resize

What’s next - Next: Chapter 23

23. Custom drawing and custom controls

In this chapter you'll learn when to draw by hand and when to build a templated control, what the DrawingContext can do, how invalidation works, and how to structure a simple custom control that is easy to style and fast to render.

What you'll build - A minimal custom-drawn control that renders a sparkline from numbers - A templated Badge control that can be restyled in XAML without code changes

When should you draw vs template? - Custom drawing (override Render) is great when: - You need pixel-level control (charts/graphs/special effects) - You want maximum performance and minimum visual tree overhead - The visuals don't need to be deeply interactive or individually templated - Templated control (XAML ControlTemplate) is great when: - You want consumers to restyle with pure XAML - The control is composed from existing primitives (Border, Grid, Path, TextBlock) - You prefer layout flexibility over raw drawing performance

Your rendering hook: override Render - Every Visual has a virtual Render(DrawingContext) you can override to draw. - Call InvalidateVisual() whenever something changes that affects the output so the renderer repaints. - For properties that affect rendering, register AffectsRender in the static constructor so changes auto-invalidate.

DrawingContext in 5 minutes - Primitives you'll use most: - DrawGeometry(brush, pen, geometry) — fill/stroke arbitrary shapes (use StreamGeometry to build paths) - DrawImage(image, sourceRect, destRect) — draw bitmaps or render targets - DrawText(formattedText, origin) — draw measured text - State stack (always use using): - PushClip(Rect or RoundedRect) - PushOpacity(value[, bounds]) and PushOpacityMask(brush, bounds) - PushTransform(Matrix) These return a disposable "pushed state"; dispose in reverse order (the using pattern makes this automatic).

Minimal custom-drawn control: Sparkline Goal: render a small polyline from a sequence of doubles.

Steps 1) Create a class Sparkline : Control with a Numbers property and a Stroke property. 2) In the static ctor, call AffectsRender(NumbersProperty, StrokeProperty) so changes trigger redraw. 3) Override Render and draw using StreamGeometry + DrawGeometry.

Example (C#)

```
public class Sparkline : Control
{
    public static readonly StyledProperty<IReadOnlyList<double>?> NumbersProperty =
        AvaloniaProperty.Register<Sparkline, IReadOnlyList<double>?>(nameof(Numbers));

    public static readonly StyledProperty<IBrush?> StrokeProperty =
        AvaloniaProperty.Register<Sparkline, IBrush?>(nameof(Stroke), Brushes.CornflowerBlue);

    static Sparkline()
    {
        AffectsRender<Sparkline>(NumbersProperty, StrokeProperty);
    }

    public IReadOnlyList<double>? Numbers
    {
        get => GetValue(NumbersProperty);
        set => SetValue(NumbersProperty, value);
    }

    public IBrush? Stroke
    {
        get => GetValue(StrokeProperty);
```

```
        set => SetValue(StrokeProperty, value);
    }

    public override void Render(DrawingContext ctx)
    {
        base.Render(ctx);
        var data = Numbers;
        if (data is null || data.Count < 2)
            return;

        var bounds = Bounds;
        if (bounds.Width <= 0 || bounds.Height <= 0)
            return;

        // Normalize values into [0..1]
        double min = data.Min();
        double max = data.Max();
        double range = Math.Max(1e-9, max - min);

        using var geo = new StreamGeometry();
        using (var gctx = geo.Open())
        {
            for (int i = 0; i < data.Count; i++)
            {
                double t = (double)i / (data.Count - 1);
                double x = bounds.X + t * bounds.Width;
                double yNorm = (data[i] - min) / range;
                double y = bounds.Y + (1 - yNorm) * bounds.Height;
                if (i == 0)
                    gctx.BeginFigure(new Point(x, y), isFilled: false);
                else
                    gctx.LineTo(new Point(x, y));
            }
            gctx.EndFigure(isClosed: false);
        }

        var pen = new Pen(Stroke, thickness: 1.5);
        ctx.DrawGeometry(null, pen, geo);
    }
}
```

Usage (XAML)

```
<local:Sparkline Width="120" Height="24"
                 Stroke="DeepSkyBlue"
                 Numbers="3,5,4,6,9,8,12,7,6"/>
```

Notes and tips - Do not allocate in Render if you can avoid it. Cache immutable pens/brushes if they depend on rarely changing properties. - Use AffectsRender to auto-invalidate on property changes, and call InvalidateVisual() for imperative invalidation. - Use PushClip for rounded corners or to avoid overdrawing outside Bounds. - Measure/arrange still apply. Your control's layout is separate from drawing; override MeasureOverride/ArrangeOverride for custom sizing behavior.

Templated control: Badge Goal: a restylable badge that supports content and themeable colors.

Steps 1) Create Badge : TemplatedControl with StyledProperties: Content, Background, Foreground, Cor-

nerRadius. 2) Provide a default theme style with ControlTemplate composed from Border + ContentPresenter. 3) Consumers can restyle by replacing the template in XAML without touching your C#.

Example default style (XAML)

```
<Style Selector="local|Badge">
  <Setter Property="Template">
    <ControlTemplate TargetType="local:Badge">
      <Border Background="{TemplateBinding Background}"
              CornerRadius="{TemplateBinding CornerRadius}"
              Padding="4,0"
              MinWidth="16" Height="16"
              HorizontalAlignment="Left"
              VerticalAlignment="Top">
        <ContentPresenter Content="{TemplateBinding Content}"
                          HorizontalAlignment="Center"
                          VerticalAlignment="Center"
                          Foreground="{TemplateBinding Foreground}"/>
      </Border>
    </ControlTemplate>
  </Setter>
  <Setter Property="Background" Value="#E53935"/>
  <Setter Property="Foreground" Value="White"/>
  <Setter Property="CornerRadius" Value="8"/>
  <Setter Property="FontSize" Value="11"/>
</Style>
```

When to pick which approach - Pick drawing (override Render) when visuals are algorithmic or heavy and don't need nested controls. - Pick templating when the control is a composition of existing elements and must be easily restyled. - You can combine both: a templated control that contains a light custom-drawn child for a specific part.

Invalidation that "just works" - AffectsRender ties StyledProperty changes to InvalidateVisual. Put it in your static ctor. - For dependent caches (e.g., a geometry built from multiple properties), rebuild lazily on next Render after invalidation.

Text and images - DrawText: format once, reuse many times. For dynamic text, rebuild only on changes. - DrawImage: prefer the overload with source/dest rectangles for atlases; set RenderOptions.BitmapInterpolationMode as needed per visual.

Accessibility and input - If your control is purely drawn, make sure it's focusable when needed and expose AutomationProperties.Name/HelpText. - For hit testing (e.g., series selection in a chart), map pointer positions into your geometry space and handle PointerPressed/Released.

Troubleshooting - Nothing draws: ensure your control has non-zero size and your Render override actually draws inside Bounds. - Flicker or jank: avoid per-frame allocations; cache pens/brushes/geometries; prefer using statements for Push* calls. - Blurry output: check transforms and DPI scaling; for fine lines, align to device pixels when needed.

Look under the hood (source links) - Visual.Render and invalidation: Avalonia.Base/Visual.cs - DrawingContext API: Avalonia.Base/Media/DrawingContext.cs - IDrawingContextImpl (platform bridge): Avalonia.Base/Platform/IDrawingContextImpl.cs - Skia DrawingContext implementation: Skia/Avalonia.Skia/DrawingContextImpl.cs

Practice - Implement a simple BarGauge control that draws N vertical bars from an array of values with colors derived from thresholds. Add a Templated header above it. Ensure value and color property changes trigger redraw using AffectsRender.

What's next - Next: Chapter 24

24. Performance, diagnostics, and DevTools

Goal: Give you a practical toolkit to find, understand, and fix performance issues in Avalonia apps — using logs, built-in DevTools, and lightweight measurements.

Why it matters: Most "slow UI" reports aren't about the renderer — they're caused by excessive layout, re-templating, heavy data binding, non-virtualized lists, or expensive work on the UI thread. Measure first. Then change one thing at a time.

What you'll learn - When and how to measure (and why Release builds matter) - Enabling logs and choosing log areas - Attaching and using DevTools (F12) - Reading debug overlays (FPS, dirty rects, layout/render graphs) - A simple performance checklist and fixes

1) Measure first (small, reliable checks)

- Run your app in Release: JIT and inlining matter. A quick check is to run both Debug and Release and compare feel/fps.
- Use a stopwatch for hot code paths: time just the suspected section. Don't time entire startup at first — narrow it down.
- Reproduce with small data: isolate the control or page that's slow, then scale up data size gradually to see growth patterns.
- Change one thing at a time: after each small change, re-measure.

2) Enable logs and tracing Avalonia has a flexible logging sink system. You can send logs to System.Diagnostics.Trace, a TextWriter, or a custom delegate via AppBuilder extension methods. See source: LoggingExtensions.cs

- GitHub: Avalonia.Controls/LoggingExtensions.cs

Common setup patterns

C#: enable logs to Trace

AppBuilder ConfigureAppLogging(AppBuilder builder) { // Log selected areas at Information or Warning to reduce noise. return builder.LogToTrace(Avalonia.Logging.LogEventLevel.Information, "Binding", "Property", "Layout", "Render" // pick the areas you care about); }

C#: log to a rolling file

using var writer = new StreamWriter("avalonia.log", append: true) { AutoFlush = true }; BuildAvaloniaApp().LogToTextWriter(writer, Avalonia.Logging.LogEventLevel.Information, "Binding", "Property");

Notes - Areas are strings (see Avalonia.Logging.LogArea constants in the source). Start with "Binding", "Layout", "Render", "Property". - Use Information while investigating, then raise to Warning or Error in production to keep output lean.

3) Attach DevTools (F12) and what it offers DevTools ships with Avalonia and can be attached to a TopLevel (Window) or to the Application. The default open gesture is F12. See DevToolsExtensions.cs

- GitHub: Avalonia.Diagnostics/DevToolsExtensions.cs

Attach to a window (typical desktop)

public override void OnFrameworkInitializationCompleted() { if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime d) d.MainWindow = new MainWindow();

```
base.OnFrameworkInitializationCompleted();
this.AttachDevTools(); // F12 to open
```

}

Attach with options (choose startup screen, etc.)

this.AttachDevTools(new Avalonia.Diagnostics.DevToolsOptions { StartupScreenIndex = 1, // open on a specific monitor if you have multiple });

Tip: Only enable DevTools in debug builds or behind a flag if you ship your app to end-users.

What's inside DevTools (high-level tour) - Visual Tree: inspect hierarchy, pick a control on screen, see its size, properties, and pseudo-classes (:pointerover, :pressed, etc.). - Logical Tree: inspect content and data template relationships — useful for understanding DataContext and templated children. - Properties & Styles: live property viewer with resources and styles; toggle pseudo-classes to see state-based styles. - Layout Explorer: see measure/arrange sizes and constraints; helps pinpoint "why is this control so big/small?". - Events: watch routed events fire as you interact (pointer, key, etc.). - Hotkeys page and settings: view/change the gesture to open DevTools. - Highlight adorners: enable highlighting to see layout bounds and hit test areas of the selected control.

Extra helpers in the repo - Visual tree printing helper: VisualTreeDebug.PrintVisualTree(visual) — useful for quick console diagnostics. Source: Avalonia.Diagnostics/Diagnostics/VisualTreeDebug.cs

4) Read the debug overlays (your real-time dashboard) DevTools exposes toggles for debug overlays backed by the RendererDebugOverlays enum. Source files:

- RendererDebugOverlays.cs: Avalonia.Base/Rendering/RendererDebugOverlays.cs
- Where DevTools toggles them: Diagnostics MainViewModel: Avalonia.Diagnostics/Diagnostics/ViewModels/MainView

Overlays you can enable - FPS: shows frames per second to gauge overall responsiveness. - DirtyRects: draws the areas that are actually repainted each frame — if the whole window repaints, you'll see it. - LayoutTimeGraph: a rolling chart of layout time — spikes hint at measure/arrange cost or re-layout storms. - RenderTimeGraph: a rolling chart of render time — spikes hint at custom drawing/bitmap work, or GPU uploads.

How to use overlays effectively - Turn on FPS + RenderTimeGraph. Interact with your slow view. Do spikes correlate with pointer moves, scrolling, or data updates? - If dirty rects cover the entire window on small changes, find what's invalidating broadly (global properties, effects, or a single control that invalidates too aggressively). - Combine LayoutTimeGraph with DevTools Layout Explorer to find which subtree is causing repeated measures.

5) Quick performance checklist (fix the common causes)

- Virtualize long lists: use ItemsPanel with VirtualizingStackPanel when appropriate. Keep item templates simple and cheap.
- Avoid re-creating heavy visuals: prefer bindings/state changes over replacing entire controls or DataTemplates repeatedly.
- Defer expensive work off the UI thread: use async/await and IProgress to report progress to the UI.
- Use images wisely: prefer correct sizes to avoid runtime scaling; choose BitmapInterpolationMode carefully when scaling. Per-visual RenderOptions are available and can be pushed during drawing. Source: RenderOptions.cs — Avalonia.Base/Media/RenderOptions.cs
- Cache and reuse text/geometry where possible: freeze re-usable geometries, keep FormattedText or GlyphRun if you render often.
- Minimize layout churn: avoid frequently changing properties that trigger re-measure/re-arrange for large subtrees.
- Measure and render in release: always verify improvements in Release builds.

6) DevTools vs. logs — when to use which

- Use DevTools first when the problem is "visual": too many re-layouts, big dirty rects, low FPS only when hovering/scrolling.
- Use logs when the problem is "structural": noisy bindings, property change storms, repeated template application, or unexpected errors.
- Use both together: turn on overlays and collect minimal logs (Information) to correlate what happened and when.

7) Troubleshooting

- DevTools doesn't open: ensure AttachDevTools is called after App initialization (e.g., at the end of OnFrameworkInitializationCompleted). If you changed the hotkey, verify the gesture. See DevToolsExtensions remarks in source.
- Overlays don't show: make sure the DevTools debug overlay toggles are enabled in the DevTools UI; some overlays need a frame or two to appear.
- Logs too noisy: reduce the level (Warning) or restrict areas to the ones you're investigating.
- Release is fast, Debug is slow: that's expected — use Release for realistic performance checks.

Exercise - Add this.AttachDevTools() to your app and open DevTools with F12. Turn on FPS and RenderTimeGraph. Interact with your slowest view and note the pattern. - Enable logging to Trace at Information for areas: Binding, Property, Layout, Render. Reproduce the issue and look for bursts. - Apply one fix from the checklist (e.g., replace an ItemsPanel with VirtualizingStackPanel or simplify a DataTemplate). Re-measure: did FPS improve or did render/layout spikes shrink?

Look under the hood (source links) - DevTools attach helpers: Avalonia.Diagnostics/DevToolsExtensions.cs - DevTools options and window plumbing: Avalonia.Diagnostics/Diagnostics/DevToolsOptions.cs and Avalonia.Diagnostics/Diagnostics/DevTools.cs - Debug overlays enum: Avalonia.Base/Rendering/RendererDebugOverlays.cs - Where DevTools toggles overlays: Avalonia.Diagnostics/Diagnostics/ViewModels/MainViewModel.cs - Logging extensions: Avalonia.Controls/LoggingExtensions.cs - Per-visual render options: Avalonia.Base/Media/RenderOptions.cs

What's next - Next: Chapter 25

25. Design-time tooling and the XAML Previewer

In this chapter you'll make the Previewer work for you every day. You'll learn how Avalonia's design mode works, how to feed realistic sample data to your views, how to preview styles and resources, and how to avoid common previewer pitfalls that waste time.

What you'll learn - How the Previewer and design mode work at a high level - Design-time attached properties (Design.DataContext, Design.Width/Height, DesignStyle) and when to use them - Feeding sample data safely (without running production services) - Previewing resource dictionaries and styles with Design.PreviewWith - Practical IDE usage tips and common troubleshooting

Big picture: how the Previewer works - Your IDE opens a small "designer host" process that loads your view or resource XAML and sets the app into design mode. Internally, the host uses a special entry point and windowing platform to run your XAML under tighter control. Design mode signals are propagated so your code can opt out of expensive work. - Key signals and helpers: - Design.IsDesignMode is true when running in a designer/previewer session. Your code can branch on this to skip real services, network calls, or timers. See Design.IsDesignMode in source. - A XAML compiler transformer removes all Design.* attached properties in normal runtime so they don't affect shipping builds, and applies them in design mode when loading XAML for preview. - The preview host uses a dedicated window implementation and windowing platform shim to render your views without relying on a user's desktop environment.

Design-time attached properties you'll actually use - Design.DataContext: Provide lightweight sample view models so bindings show meaningful data in the Previewer without constructing your real services. - Design.Width and Design.Height: Force a control's size in the designer so you can style it comfortably without relying on outer layout. - Design.DesignStyle: Inject an extra style only in design mode to highlight bounds, show placeholder backgrounds, or adjust layout just for preview.

Example: Design-time DataContext with a simple sample VM - Add a small sample type (keep it in your UI project for easy access): - public class SamplePerson { public string Name { get; set; } = "Ada"; public int Age { get; set; } = 42; } - Use it in XAML (map the namespace and attach Design.DataContext). At runtime, the transformer strips this, so your real DataContext takes over.

Example: Sizing and design-only style - You can set Design.Width/Design.Height to get a consistent designer canvas size. - Use Design.DesignStyle to add a dashed outline, helpful for templated controls while iterating.

Previewing styles and resources with Design.PreviewWith - You can preview a ResourceDictionary or style in isolation by providing a small host control with Design.PreviewWith. This renders your dictionary wrapped in the host, so you can iterate on colors/templates quickly. - Typical pattern in a ResourceDictionary: - Add a simple host, such as a Border or Panel with a child that uses your styles. - Set Design.PreviewWith to that host so the Previewer knows what to render for this dictionary.

Safety first: what not to run in design mode - Never start network requests, database connections, background threads, or timers from view constructors if Design.IsDesignMode is true. - Avoid static initialization that reaches out to the environment (files, registry, user profile) in design mode. - If your ViewModel normally uses services, inject stub/fake implementations when Design.IsDesignMode is true, or use the simple POCO sample objects shown above.

Practical IDE tips - Keep your view constructors cheap and side-effect free. Heavy work belongs in async commands triggered by user actions, not in constructors or OnApplyTemplate. - Prefer simple sample models for preview data over spinning up your composition root. - If the previewer crashes on a view, open a smaller piece (e.g., a UserControl used inside that view) to narrow the issue. - If a style/resource dictionary doesn't preview, add Design.PreviewWith with a minimal host and a representative control that consumes your style.

Troubleshooting checklist - Blank or flickering preview: remove animations/triggers, reduce effects, or temporarily comment expensive bindings. Heavy effects can overwhelm the design host. - Crashes on load: guard code with if (Design.IsDesignMode) return; in constructors/init paths that run in the designer. - Stale data: rebuild the project to flush caches. Some IDEs keep a warm previewer instance. - Missing resources: verify avares URIs and resource include scopes. In design mode, the designer may load only the UI assembly;

ensure resources are in the correct project. - Platform assumptions: don't assume a particular OS/GPU. The previewer may use a special windowing backend.

Look under the hood (source tour) - Design-time API surface (Design.*): Design.cs - Avalonia.Controls/Design.cs - Designer XAML loader and property application: - Avalonia.DesignerSupport/DesignWindowLoader.cs - Previewer entry point and design mode enablement: - Avalonia.DesignerSupport/Remote/RemoteDesignerEntryPoint.cs - XAML compiler transformer that strips Design.* at runtime: - Avalonia.Markup.Xaml.Loader/CompilerExtensions/Transforn - Designer window/platform shim used by the preview host: - Avalonia.DesignerSupport/Remote/PreviewerWindowImpl.cs - Avalonia.DesignerSupport/Remote/PreviewerWindowingPlatform.cs - PlatformManager helpers used in designer mode: - Avalonia.Controls/Platform/PlatformManager.cs - XAML runtime loader with designMode parameter: - Avalonia.Markup.Xaml.Loader/AvaloniaRuntimeXamlLoader.cs

Exercise: Make a view designer-friendly 1) Pick an existing UserControl in your app that currently shows poorly in the Previewer. 2) Create a tiny sample POCO model with realistic values and attach it with Design.DataContext. 3) Add Design.Width/Design.Height so you have a predictable canvas while styling. 4) If the view relies on styles from a dictionary, add Design.PreviewWith to that dictionary with a host and a representative control to preview the style. 5) Confirm the preview shows your sample data and styles. Remove any unnecessary design-only helpers once you're done.

What's next - Next: Chapter 26

26. Build, publish, and deploy

In this chapter you'll learn how to turn your Avalonia project into distributable builds for each platform. You'll understand the difference between building and publishing, how to choose the right runtime identifier (RID), and how to ship self-contained, single-file, and trimmed builds responsibly.

What you'll learn - Build vs publish in .NET and why Release builds matter - Runtime identifiers (RID) and cross-platform publishing - Framework-dependent vs self-contained builds - Single-file, ReadyToRun, and trimming options (and their trade-offs) - Where files land, how to run them, and what to test before shipping

Build vs publish (in plain words) - Build compiles your project into assemblies for running from your dev box. Publish creates a folder you can copy to a target machine and run there (optionally without installing .NET). - Always test performance and behavior with Release builds. Debug builds include extra checks and are slower.

Runtime identifiers (RIDs) you'll actually use - Windows: win-x64, win-arm64 - macOS: osx-x64 (Intel), osx-arm64 (Apple Silicon) - Linux: linux-x64, linux-arm64 - Pick the RID(s) your users need. You can publish multiple variants.

Framework-dependent vs self-contained - Framework-dependent: smaller download; requires the correct .NET runtime to be installed on the target machine. - Self-contained: includes the .NET runtime; larger download; runs on machines without .NET installed. Recommended for consumer apps to reduce support friction.

Common publish layouts and options - Minimal framework-dependent build: - dotnet publish -c Release -r win-x64 –self-contained false - Self-contained build: - dotnet publish -c Release -r osx-arm64 –self-contained true - Single-file (packs your app into one executable and a few support files as needed): - dotnet publish -c Release -r linux-x64 /p:SelfContained=true /p:PublishSingleFile=true - ReadyToRun (improves startup by precompiling IL to native code; increases size): - dotnet publish -c Release -r win-x64 /p:SelfContained=true /p:PublishReadyToRun=true - Trimming (reduces size by removing unused code; use with care due to reflection and data binding): - dotnet publish -c Release -r osx-arm64 /p:SelfContained=true /p:PublishTrimmed=true

Trade-offs and cautions - Single-file may still extract native libraries to a temp folder on first run; measure startup and disk impact. - ReadyToRun boosts cold start but can make binaries larger; verify for your app size/benefit. - Trimming can remove types used via reflection (including XAML/bindings). Test thoroughly; avoid trimming until you verify that everything works, or add preservation hints incrementally. Start without trimming, then iterate.

Where to find your output - After publishing, look under bin/Release///publish. For example: - bin/Release/net8.0/win-x64/publish - bin/Release/net8.0/osx-arm64/publish - bin/Release/net8.0/linux-x64/publish - Run your app directly from that publish folder on a matching OS/CPU.

Platform notes (high-level) - Windows: a signed self-contained single-file EXE is a simple way to distribute. For enterprise or store delivery, consider installer packages (MSIX/MSI) and code signing. - macOS: you'll likely want an app bundle (.app) and code signing/notarization for a smooth Gatekeeper experience. For development, you can run the published binary; for distribution, follow Apple's signing guidance. - Linux: many users are comfortable with a tar.gz of your publish folder. For a desktop-native feel, consider packaging systems like AppImage, Flatpak, or Snap used by various distros.

Quality checklist before shipping - Publish in Release for each RID you plan to support. - Run the app on real target machines (or VMs) for each platform. Verify rendering, fonts, DPI, file dialogs, and hardware acceleration behave as expected. - Check that resources (images, styles, fonts) load correctly from the publish folder. - If you use single-file or trimming, exercise all major screens and dynamic features (templates, reflection, localization). - If you ship self-contained, verify size is acceptable and startup times are reasonable.

Troubleshooting - Missing dependencies on Linux: install common desktop libraries (font and ICU packages). If the app starts only from a terminal with errors, note missing libraries and install them via your distro's

package manager. - Crashes only in Release: ensure you aren't relying on Debug-only conditions, and remove dev-only code paths. Enable logging to a file during testing to capture issues. - Graphics differences: different GPUs/drivers can affect performance. Test with integrated and discrete GPUs where possible. - File associations and icons: packaging systems (MSIX, app bundles, AppImage/Flatpak) handle these better than raw folders. Plan packaging early if you need OS integration.

Look under the hood (docs and sources) - Build and guidance in the Avalonia repo docs: - docs/build.md - Samples you can build and publish for reference: - samples/ControlCatalog - samples

Exercise: Publish and run your app 1) Publish a self-contained build for your current OS RID with single-file enabled. Locate the publish folder and run the app directly from there. 2) Repeat for a second RID (e.g., win-x64 or linux-x64). If you can't run it locally, copy it to a matching machine/VM and test. 3) Note the publish size and startup time with and without PublishSingleFile/ReadyToRun. Keep the variant that best balances size and speed for your audience.

What's next - Next: Chapter 27

27. Read the source, contribute, and grow

This final chapter is an invitation to go beyond this book. Reading real framework code deepens your understanding, contributing makes you a better engineer, and engaging with the community helps you stay current and grow.

What you'll learn in this chapter - How to navigate the Avalonia source tree and build it locally - Where to look in the code when you want to understand a feature - Practical tips for stepping into framework sources while debugging your app - How to file great issues and contribute high-quality pull requests - How to contribute to documentation and samples - Ways to stay involved and keep learning

Why read the source - Solidify mental models: reading implementation details clarifies how layout, input, rendering, and styling actually work in practice. - Improve debugging: once you know where code lives, you can step into it confidently and diagnose tricky problems. - Contribute fixes and features: you'll be able to propose targeted improvements with realistic scope.

Tour the repository (what to look for) - Core sources and platform code - src - You'll find core assemblies (e.g., Base, Controls, Diagnostics, Skia, etc.) here. Browse folders to see how subsystems are organized. - Tests - tests - Tests are a goldmine for learning: they capture expected behaviors and edge cases. When adding features or fixing bugs, add or update tests here. - Samples - samples - Run and read samples (like the Control Catalog) to see idiomatic patterns and verify changes. - Project guidance - CONTRIBUTING.md - CODE_OF_CONDUCT.md - readme.md - Documentation site (source) - avalonia-docs/docs - If you enjoy writing, this is where you can improve official docs, tutorials, and guides.

Build the framework locally - Scripts for building on your OS are in the repo root: - build.ps1 - build.sh - build.cmd - You can also open the solution to explore and build individual projects: - Avalonia.sln - Run a sample to verify your environment: - Control Catalog: samples/ControlCatalog

Read with purpose: where to look for… - Logging and diagnostics - LoggingExtensions.cs - DevToolsExtensions.cs - RendererDebugOverlays.cs - Design mode and previewing - Design.cs - Rendering and options - Skia options and rendering backends live under Skia/ and platform rendering folders in src. - Controls and styling - Controls and styles are under Controls/ with templates/resources organized alongside. Tests illustrate expected styling behaviors in the tests tree.

Step into sources while debugging your app - Enable stepping into external code and include debug symbols for framework assemblies when possible. This lets you follow execution into Avalonia internals. - Start from your call site (e.g., a control event handler) and step forward to see how data flows through layout, rendering, or input. - Keep the DevTools open during debugging to correlate what you see in the tree/overlays with the code paths you step through.

File great issues (and get faster resolutions) - Always include a minimal repro: the smallest sample that demonstrates the bug. Link to a repository or attach a tiny project. - Specify platform(s), .NET version, Avalonia version, and whether the problem reproduces in Release. - Add screenshots or screen recordings when visual behavior is involved, and note any debug overlays or DevTools findings. - Be precise about expected vs. actual behavior and list steps to reproduce.

Contribute high-quality pull requests - Keep scope focused and change sets small. Smaller PRs review faster and are easier to merge. - Add tests in tests that cover the fix or feature. Tests protect your change and prevent regressions. - Follow project guidance: - - Match coding style and file organization used in neighboring files. - Explain your approach in the PR description, reference related issues, and call out trade-offs or follow-ups. - Be responsive to review feedback; maintainers and contributors are collaborators.

Contribute to documentation and samples - Docs live in avalonia-docs/docs. Improvements to conceptual docs, guides, and API explanations are always valuable. - Samples live in samples. New focused samples that illustrate tricky scenarios are welcomed. - When you fix a bug or add a feature, consider also updating docs and adding a small sample demonstrating it.

Grow with the community - Start by reading the project readme.md and contribution docs to learn how the community organizes work. - Look for labels like "good first issue" to find beginner-friendly tasks. If

you're unsure, ask in the issue before starting. - Share knowledge: blog, speak, or help answer questions in community channels listed in the repository's README.

Checklist for sustainable contributions - Can you reproduce the issue consistently with a minimal sample? - Do tests cover your change, including edge cases? - Did you benchmark or measure performance when relevant? - Did you run samples across platforms that your change touches? - Did you update docs and samples where appropriate?

Exercise: Follow a feature from UI to rendering - Pick a simple visual element (e.g., Border, TextBlock) in the Control Catalog sample. - Set a breakpoint in your app code where you configure it. - Step into the framework source and trace how it measures, arranges, and renders. - Locate associated tests and read their assertions. - Make a tiny change locally (e.g., add a comment or an extra test) to practice the contribution workflow.

What's next - Back to Table of Contents