Avalonia Book

Contents

1. Welcome to Avalonia and MVVM

Goal - Understand what Avalonia is today, how it has grown, and where it is heading. - Learn the roles of C#, XAML, and MVVM (with their core building blocks) inside an Avalonia app. - Map Avalonia's layered architecture so you can navigate the source confidently. - Compare Avalonia with WPF, WinUI, .NET MAUI, and Uno to make an informed platform choice. - Follow the journey from `AppBuilder.Configure` to the first window, and know how to inspect it in the samples.

Why this matters - Picking a UI framework is a strategic decision. Knowing Avalonia's history, roadmap, and governance helps you judge its momentum. - Understanding the framework layers and MVVM primitives prevents "magic" and makes documentation, samples, and source code less intimidating. - Being able to contrast Avalonia with sibling frameworks keeps expectations realistic and helps you explain the choice to teammates.

Avalonia in simple words - Avalonia is an open-source, cross-platform UI framework. One code base targets Windows, macOS, Linux, Android, iOS, and the browser (WebAssembly). - It brings a modern Fluent-inspired theme, a deep control set, rich data binding, and tooling such as DevTools and the XAML Previewer. - If you have WPF experience, Avalonia feels familiar; if you are new, you get gradual guidance with MVVM, XAML, and C#.

A short history, governance, and roadmap - Origins (2013-2018): The project began as a community effort to bring a modern, cross-platform take on the WPF programming model. - Maturing releases (0.9-0.10): Stabilised control set, styling, and platform backends while adding mobile and browser support. - Avalonia 11 (2023): The 11.x line introduced the Fluent 2 theme refresh, compiled bindings, a new rendering backend, and long-term support. New minor updates land roughly every 2-3 months with patch releases in between. - Governance: AvaloniaUI is stewarded by a core team at Avalonia Solutions Ltd. with an active GitHub community. Development is fully open with public issue tracking and roadmap discussions. - Roadmap themes: continuing Fluent updates, performance and tooling investments, deeper designer integration, and steady platform parity across desktop, mobile, and web.

How Avalonia is layered - **Avalonia.Base**: foundational services–dependency properties (`AvaloniaProperty`), threading, layout primitives, and rendering contracts. Source: src/Avalonia.Base. - **Avalonia.Controls**: the control set, templated controls, panels, windowing, and lifetimes. Source: src/Avalonia.Controls with the `Application` class in Application.cs. - **Styling and themes**: styles, selectors, control themes, and Fluent resources. Source: src/Avalonia.Base/Styling and src/Avalonia.Themes.Fluent. - **Markup**: XAML parsing, compiled XAML, and the runtime loader used at startup. Source: src/Avalonia.Markup.Xaml with AvaloniaXamlLoader.cs. - **Platform backends**: per-OS integrations–for example src/Windows/Avalonia.Win32, src/Avalonia.Native, src/Android/Avalonia.Android, src/iOS/Avalonia.iOS, and src/Browser/Avalonia.Browser.

C#, XAML, and MVVM–who does what - **C#**: application startup (`AppBuilder`), services, models, and view models. Logic lives in strongly typed classes. - **XAML**: declarative UI markup–controls, layout, styles, resources, and data templates. - **MVVM**: separates responsibilities. The View (XAML) binds to a ViewModel (C#) which exposes Models and services. Tests target ViewModels and models directly.

MVVM building blocks you should recognise early - `INotifyPropertyChanged`: standard .NET interface. When a ViewModel property raises `PropertyChanged`, bound controls refresh. - `AvaloniaProperty`: Avalonia's dependency property system (see AvaloniaProperty.cs) powers styling, animation, and templated control state. - Binding expressions: XAML bindings are parsed and applied via the XAML loader. The runtime loader lives in AvaloniaXamlLoader.cs. - Commands: typically `ICommand` implementations on the ViewModel (plain or via libraries such as CommunityToolkit.Mvvm or ReactiveUI) so buttons and menu items can invoke logic. - Data templates: define how ViewModels render in lists and navigation. We will use them extensively starting in Chapter 3.

From `AppBuilder.Configure` to the first window (annotated flow) 1. **Program entry point** creates a builder: `BuildAvaloniaApp()` returns `AppBuilder.Configure<App>()`. 2. **Platform detection** (`UsePlatformDetect`) selects the right backend (Win32, macOS, X11, Android, iOS, Browser). 3. **Rendering setup** (`UseSkia`) chooses the rendering pipeline–Skia by default. 4. **Logging and services** (`LogToTrace`, custom DI) configure diagnostics. 5. **Start a lifetime**:

`StartWithClassicDesktopLifetime(args)` (desktop) or `StartWithSingleViewLifetime` (mobile/browser). Lifetimes live under ApplicationLifetimes. 6. **`Application` initialises**: `App.OnFrameworkInitializationCompleted` is called; this is where you typically create and show the first `Window` or set `MainView`. 7. **XAML loads**: `AvaloniaXamlLoader` reads `App.axaml` and your window/user control XAML. 8. **Bindings connect**: when the window's data context is set to a ViewModel, bindings listen for `PropertyChanged` events and keep UI and data in sync.

Tour the ControlCatalog (your guided sample) - Clone the repo (or open the ControlCatalog sample). - `ControlCatalog.Desktop` demonstrates desktop controls, theming, and navigation. Inspect `App.axaml`, `MainWindow.axaml`, and their code-behind to see how `AppBuilder` and MVVM connect. - Use DevTools (press `F12` when running the sample) to inspect bindings, the visual tree, and live styles. - Explore the repository mapping: the `Button` page in the catalog points to code under src/Avalonia.Controls/Button.cs; style resources originate from Fluent theme XAML under src/Avalonia.Themes.Fluent/Controls.

Why Avalonia instead of… - **WPF** (Windows only): mature desktop tooling and huge ecosystem, but no cross-platform story. Avalonia keeps the mental model while expanding to macOS, Linux, mobile, and web. - **WinUI 3** (Windows 10/11): modern Windows UI with native Win32 packaging. Great for Windows-only solutions; Avalonia wins when you must ship beyond Windows. - **.NET MAUI**: Microsoft's cross-platform evolution of Xamarin.Forms focused on mobile-first UI. Avalonia emphasises desktop parity, theming flexibility, and XAML consistency across platforms. - **Uno Platform**: reuses WinUI XAML across platforms via WebAssembly and native controls. Avalonia offers a single rendering pipeline (Skia) for consistent visuals when you prefer pixel-perfect fidelity over native look-and-feel.

Repository landmarks (bookmark these) - Framework source: src - Samples: samples - Docs: docs - ControlCatalog entry point: ControlCatalog.csproj

Check yourself - Can you describe how Avalonia evolved to its current release cadence and governance model? - Can you name the key Avalonia layers (Base, Controls, Markup, Themes, Platforms) and what each provides? - Can you explain the MVVM building blocks (`INotifyPropertyChanged`, `AvaloniaProperty`, bindings, commands) in your own words? - Can you sketch the `AppBuilder` startup steps that end with a `Window` or `MainView` being shown? - Can you list one reason you might choose Avalonia over WPF, WinUI, .NET MAUI, or Uno?

Practice and validation - Clone the Avalonia repository, build, and run the desktop ControlCatalog. Set a breakpoint in `Application.OnFrameworkInitializationCompleted` inside `App.axaml.cs` to watch the lifetime hand-off. - While ControlCatalog runs, open DevTools (F12) and track a ViewModel property change (for example, toggle a CheckBox) in the binding diagnostics panel to see `PropertyChanged` events flowing. - Inspect the source jump-offs for `Application` (Application.cs), `AvaloniaProperty` (AvaloniaProperty.cs), and the XAML loader (AvaloniaXamlLoader.cs). Note how the pieces you just read about appear in real code.

What's next - Next: Chapter 2

2. Set up tools and build your first project

Goal - Install the .NET SDK, Avalonia templates, and an IDE on your operating system of choice. - Configure optional workloads (Android, iOS, WebAssembly) so you are ready for multi-target development. - Create, build, and run a new Avalonia project from the command line and from your IDE. - Understand the generated project structure and where startup, resources, and build targets live. - Build the Avalonia framework from source when you need nightly features or to debug the platform.

Why this matters - A confident setup avoids painful environment issues later when you add mobile or browser targets. - Knowing where the generated files live prepares you for upcoming chapters on layout, lifetimes, and MVVM. - Building the framework from source lets you test bug fixes, follow development, and debug into the toolkit.

Prerequisites by operating system

Windows

- Install the latest **.NET SDK** (x64) from https://dotnet.microsoft.com/download.
- Install **Visual Studio 2022** with the ".NET desktop development" workload; add ".NET Multi-platform App UI development" for mobile tooling.
- Optional: `winget install --id Microsoft.DotNet.SDK.8` (replace with the current LTS) and install the **Windows Subsystem for Linux** if you plan to test Linux packages.

macOS

- Install the latest **.NET SDK (Arm64 or x64)** from Microsoft.
- Install **Xcode** (App Store) to satisfy iOS build prerequisites.
- Recommended IDEs: **JetBrains Rider**, **Visual Studio 2022 for Mac** (if installed), or **Visual Studio Code** with the C# Dev Kit.
- Optional: install **Homebrew** and use it for `brew install dotnet-sdk` to keep versions updated.

Linux (Ubuntu/Debian example)

- Add the Microsoft package feed and install the latest **.NET SDK** (`sudo apt install dotnet-sdk-8.0`).
- Install an IDE: **Rider** or **Visual Studio Code** with the C# extension (OmniSharp or C# Dev Kit).
- Ensure GTK dependencies are present (`sudo apt install libgtk-3-0 libwebkit2gtk-4.1-0`) because the ControlCatalog sample relies on them.

  Verify your SDK installation:

  ```
  dotnet --version
  dotnet --list-sdks
  ```

  Make sure the Avalonia-supported SDK (currently .NET 8.x for Avalonia 11) appears in the list before moving on.

Optional workloads for advanced targets

Run these commands only if you plan to target additional platforms soon (you can add them later):

```
dotnet workload install wasm-tools      # Browser (WebAssembly)
dotnet workload install android         # Android toolchain
dotnet workload install ios             # iOS/macOS Catalyst toolchain
```

If a workload fails, run `dotnet workload repair` and confirm your IDE also installed the Android/iOS dependencies (Android SDK Managers, Xcode command-line tools).

Recommended IDE setup

Visual Studio 2022 (Windows)

- Ensure the **Avalonia for Visual Studio** extension is installed (Marketplace) for XAML IntelliSense and the previewer.
- Enable **XAML Hot Reload** under Tools -> Options -> Debugging -> General.
- For Android/iOS, open Visual Studio Installer and add the corresponding mobile workloads.

JetBrains Rider

- Install the **Avalonia plugin** (File -> Settings -> Plugins -> Marketplace -> search "Avalonia").
- Enable the built-in XAML previewer via `View -> Tool Windows -> Avalonia Previewer`.
- Configure Android SDKs under Preferences -> Build Tools if you plan to run Android projects.

Visual Studio Code

- Install the **C# Dev Kit** or **C# (OmniSharp)** extension for IntelliSense and debugging.
- Add the **Avalonia for VS Code** extension for XAML tooling and preview.
- Configure `dotnet watch` tasks or use the Avalonia preview extension's Live Preview panel.

Install Avalonia project templates

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

This adds templates such as `avalonia.app`, `avalonia.mvvm`, `avalonia.reactiveui`, and `avalonia.xplat`.

Verify installation:

```
dotnet new list avalonia
```

You should see a table of available Avalonia templates.

Create and run your first project (CLI-first flow)

```
# Create a new solution folder
mkdir HelloAvalonia && cd HelloAvalonia

# Scaffold a desktop app template (code-behind pattern)
dotnet new avalonia.app -o HelloAvalonia.Desktop

cd HelloAvalonia.Desktop

# Restore packages and build
dotnet build

# Run the app
dotnet run
```

A starter window appears. Close it when done.

Alternative templates

- `dotnet new avalonia.mvvm -o HelloAvalonia.Mvvm` -> includes a ViewModel base class and data-binding sample.
- `dotnet new avalonia.reactiveui -o HelloAvalonia.ReactiveUI` -> adds ReactiveUI integration out of the box.
- `dotnet new avalonia.app --multiplatform -o HelloAvalonia.Multi` -> single-project layout with mobile/browser heads.

Open the project in your IDE

Visual Studio

1. File -> Open -> Project/Solution -> select `HelloAvalonia.Desktop.csproj`.
2. Press **F5** (or the green Run arrow) to launch with the debugger.
3. Verify XAML Hot Reload by editing `MainWindow.axaml` while the app runs.

Rider

1. File -> Open -> choose the solution folder.
2. Use the top-right run configuration to run/debug.
3. Open the Avalonia Previewer tool window to see live XAML updates.

VS Code

1. `code .` inside the project directory.
2. Accept the prompt to add build/debug assets; VS Code generates `launch.json` and `.vscode/tasks.json`.
3. Use the Run and Debug panel (F5) and the Avalonia preview extension for live previews.

Generated project tour (why each file matters)

- `HelloAvalonia.Desktop.csproj`: project metadata–target frameworks, NuGet packages, Avalonia build tasks (`Avalonia.Build.Tasks` compiles XAML to BAML-like assets; see CompileAvaloniaXamlTask.cs).
- `Program.cs`: entry point returning `BuildAvaloniaApp()`. Calls `UsePlatformDetect`, `UseSkia`, `LogToTrace`, and starts the classic desktop lifetime (definition in AppBuilderDesktopExtensions.cs).
- `App.axaml` / `App.axaml.cs`: global resources and startup logic. `App.OnFrameworkInitializationCompleted` creates and shows `MainWindow` (implementation defined in Application.cs).
- `MainWindow.axaml` / `.axaml.cs`: your initial view. XAML is loaded by AvaloniaXamlLoader.
- `Assets/` and `Styles/`: sample resource dictionaries you can expand later.

Make a visible change and rerun

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="HelloAvalonia.MainWindow"
        Width="400" Height="260"
        Title="Hello Avalonia!">
  <StackPanel Margin="16" Spacing="12">
    <TextBlock Text="It works!" FontSize="24"/>
    <Button Content="Click me" HorizontalAlignment="Left"/>
  </StackPanel>
</Window>
```

Rebuild and run (`dotnet run` or IDE Run) to confirm the change.

Troubleshooting checklist

- **`dotnet` command missing**: reinstall the .NET SDK and restart the terminal/IDE. Confirm environment variables (`PATH`) include the dotnet installation path.
- **Template not found**: rerun `dotnet new install Avalonia.Templates` or remove outdated versions with `dotnet new uninstall Avalonia.Templates`.
- **NuGet restore issues**: clear caches (`dotnet nuget locals all --clear`), ensure internet access or configure an offline mirror, then rerun `dotnet restore`.

- **Workload errors**: run `dotnet workload repair`. Ensure Visual Studio or Xcode installed the matching tooling.
- **IDE previewer fails**: confirm the Avalonia extension/plugin is installed, build the project once, and check the Output window for loader errors.
- **Runtime missing native dependencies** (Linux): install GTK, Skia, and OpenGL packages (`libmesa`, `libx11-dev`).

Build Avalonia from source (optional but recommended once)

- Clone the framework: `git clone https://github.com/AvaloniaUI/Avalonia.git`.
- Initialise submodules if prompted: `git submodule update --init --recursive`.
- On Windows: run `.\build.ps1 -Target Build`.
- On macOS/Linux: run `./build.sh --target=Build`.
- Docs reference: docs/build.md.
- Launch the ControlCatalog from source: `dotnet run --project samples/ControlCatalog.Desktop/ControlCatalo`

Building from source gives you binaries with the latest commits, useful for testing fixes or contributing.

Practice and validation

1. Confirm your environment with `dotnet --list-sdks` and `dotnet workload list`.
2. Install the Avalonia templates and scaffold a new project.
3. Run the app from the CLI and from your IDE, verifying hot reload or the previewer works.
4. Clone the Avalonia repo, build it, and run the ControlCatalog sample.
5. Set a breakpoint in `App.axaml.cs` (`OnFrameworkInitializationCompleted`) and step through startup to watch the lifetime initialise.

Look under the hood (source bookmarks)

- Build pipeline tasks: src/Avalonia.Build.Tasks.
- Desktop lifetime helpers: src/Avalonia.Desktop/AppBuilderDesktopExtensions.cs.
- ControlCatalog project: samples/ControlCatalog/ControlCatalog.csproj.
- Framework application startup: src/Avalonia.Controls/Application.cs.

Check yourself

- Which command installs Avalonia templates and how do you verify the install?
- How do you list installed .NET SDKs and workloads?
- Where does `App.OnFrameworkInitializationCompleted` live and what does it do?
- Which files control project startup, resources, and views in a new template?
- What steps are required to build Avalonia from source on your OS?

What's next - Next: Chapter 3

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

Goal - Build your first meaningful window with StackPanel, Grid, and reusable user controls. - Learn how `ContentControl`, `UserControl`, and `NameScope` help you compose UIs cleanly. - See how logical and visual trees differ so you can find controls and debug bindings. - Use `ItemsControl` with `DataTemplate` and a simple value converter to repeat UI for collections. - Understand XAML namespaces (`xmlns:`) and how to reference custom classes or Avalonia namespaces.

Why this matters - Real apps are more than a single window–you compose views, reuse user controls, and bind lists of data. - Understanding the logical tree versus the visual tree makes tooling (DevTools, FindControl, bindings) predictable. - Data templates and converters are the backbone of MVVM-friendly UIs; learning them early prevents hacks later.

Prerequisites - Chapter 2 completed. You can run `dotnet new`, `dotnet build`, and `dotnet run` on your machine.

1. Scaffold the sample project

```
# Create a new sample app for this chapter
dotnet new avalonia.mvvm -o SampleUiBasics
cd SampleUiBasics

# Restore packages and run once to ensure the template works
dotnet run
```

Open the project in your IDE before continuing.

2. Quick primer on XAML namespaces

The root `<Window>` tag declares namespaces so XAML can resolve types:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:ui="clr-namespace:SampleUiBasics.Views"
        x:Class="SampleUiBasics.Views.MainWindow">
```

- The default namespace maps to common Avalonia controls (Button, Grid, StackPanel).
- `xmlns:x` exposes XAML keywords like `x:Name`, `x:Key`, and `x:DataType`.
- Custom prefixes (e.g., `xmlns:ui`) point to CLR namespaces in your project or other assemblies so you can reference your own classes or controls (`ui:AddressCard`).

3. Build the main layout (StackPanel + Grid)

Open `Views/MainWindow.axaml` and replace the `<Window.Content>` with:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:ui="clr-namespace:SampleUiBasics.Views"
        x:Class="SampleUiBasics.Views.MainWindow"
        Width="540" Height="420"
        Title="Customer overview">
  <DockPanel LastChildFill="True" Margin="16">
    <TextBlock DockPanel.Dock="Top"
               Classes="h1"
               Text="Customer overview"
               Margin="0,0,0,16"/>

    <Grid ColumnDefinitions="2*,3*"
```

```
          RowDefinitions="Auto,*"
          ColumnSpacing="16"
          RowSpacing="16">

      <StackPanel Grid.Column="0" Spacing="8">
        <TextBlock Classes="h2" Text="Details"/>

        <Grid ColumnDefinitions="Auto,*" RowDefinitions="Auto,Auto,Auto" RowSpacing="8" ColumnSpacing="
          <TextBlock Text="Name:"/>
          <TextBox Grid.Column="1" Width="200" Text="{Binding Customer.Name}"/>

          <TextBlock Grid.Row="1" Text="Email:"/>
          <TextBox Grid.Row="1" Grid.Column="1" Text="{Binding Customer.Email}"/>

          <TextBlock Grid.Row="2" Text="Status:"/>
          <ComboBox Grid.Row="2" Grid.Column="1" SelectedIndex="0">
            <ComboBoxItem>Prospect</ComboBoxItem>
            <ComboBoxItem>Active</ComboBoxItem>
            <ComboBoxItem>Dormant</ComboBoxItem>
          </ComboBox>
        </Grid>
      </StackPanel>


      <StackPanel Grid.Column="1" Spacing="8">
        <TextBlock Classes="h2" Text="Recent orders"/>
        <ItemsControl Items="{Binding RecentOrders}">
          <ItemsControl.ItemTemplate>
            <DataTemplate>
              <ui:OrderRow />
            </DataTemplate>
          </ItemsControl.ItemTemplate>
        </ItemsControl>
      </StackPanel>
    </Grid>
  </DockPanel>
</Window>
```

What you just used: - `DockPanel` places a title bar on top and fills the rest. - `Grid` split into two columns for the form (left) and list (right). - `ItemsControl` repeats a data template for each item in `RecentOrders`.

4. Create a reusable user control (`OrderRow`)

Add a new file `Views/OrderRow.axaml`:

```
<UserControl xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             x:Class="SampleUiBasics.Views.OrderRow"
             Padding="8"
             Classes="card">
  <Border Background="{DynamicResource ThemeBackgroundBrush}"
          CornerRadius="6"
          Padding="12">
    <Grid ColumnDefinitions="*,Auto" RowDefinitions="Auto,Auto" ColumnSpacing="12">
      <TextBlock Classes="h3" Text="{Binding Title}"/>
```

```
      <TextBlock Grid.Column="1"
                 Foreground="{DynamicResource ThemeAccentBrush}"
                 Text="{Binding Total, Converter={StaticResource CurrencyConverter}}"/>

      <TextBlock Grid.Row="1" Grid.ColumnSpan="2" Text="{Binding PlacedOn, StringFormat='Ordered on {0:
    </Grid>
  </Border>
</UserControl>
```

- `UserControl` encapsulates UI so you can reuse it via `<ui:OrderRow />`.
- It relies on bindings (`Title`, `Total`, `PlacedOn`) which come from the current item in the data template.
- Using a user control keeps the item template readable and testable.

5. Add a value converter

Converters adapt data for display. Create `Converters/CurrencyConverter.cs`:

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

namespace SampleUiBasics.Converters;

public sealed class CurrencyConverter : IValueConverter
{
    public object? Convert(object? value, Type targetType, object? parameter, CultureInfo culture)
    {
        if (value is decimal amount)
            return string.Format(culture, "{0:C}", amount);

        return value;
    }

    public object? ConvertBack(object? value, Type targetType, object? parameter, CultureInfo culture) =
}
```

Register the converter in `App.axaml` so XAML can reference it:

```
<Application xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:converters="clr-namespace:SampleUiBasics.Converters"
             x:Class="SampleUiBasics.App">
  <Application.Resources>
    <converters:CurrencyConverter x:Key="CurrencyConverter"/>
  </Application.Resources>

  <Application.Styles>
    <FluentTheme />
  </Application.Styles>
</Application>
```

6. Populate the ViewModel with nested data

Open `ViewModels/MainWindowViewModel.cs` and replace its contents with:

```
using System;
using System.Collections.ObjectModel;
```

```
namespace SampleUiBasics.ViewModels;

public sealed class MainWindowViewModel
{
    public CustomerViewModel Customer { get; } = new("Avery Diaz", "avery@example.com");

    public ObservableCollection<OrderViewModel> RecentOrders { get; } = new()
    {
        new OrderViewModel("Starter subscription", 49.00m, DateTime.Today.AddDays(-2)),
        new OrderViewModel("Design add-on", 129.00m, DateTime.Today.AddDays(-12)),
        new OrderViewModel("Consulting", 900.00m, DateTime.Today.AddDays(-20))
    };
}

public sealed record CustomerViewModel(string Name, string Email);

public sealed record OrderViewModel(string Title, decimal Total, DateTime PlacedOn);
```

Now bindings like `{Binding Customer.Name}` and `{Binding RecentOrders}` have backing data.

7. Understand `ContentControl`, `UserControl`, and `NameScope`

- **`ContentControl`** (see ContentControl.cs) holds a single content object. Windows, Buttons, and many controls inherit from it. Setting `Content` or placing child XAML elements populates that content.
- **`UserControl`** (see UserControl.cs) is a convenient way to package a small view with its own XAML and code-behind. Each `UserControl` has its own `NameScope`.
- **`NameScope`** (see NameScope.cs) governs how `x:Name` lookups work. By default, names are scoped to the nearest `NameScope` provider (Window, UserControl). Use `this.FindControl<T>("CounterText")` or `NameScope.GetNameScope(this)` to resolve names inside the scope.

When you nest user controls, remember: a name defined in `OrderRow` is not visible in `MainWindow` because each `UserControl` has its own scope. This avoids name collisions in templated scenarios.

8. Logical tree vs visual tree (why it matters)

- The **logical tree** tracks content relationships: windows -> user controls -> ItemsControl items. Bindings and resource lookups walk the logical tree. Inspect with `this.GetLogicalChildren()` or DevTools -> Logical tree.
- The **visual tree** includes the actual visuals created by templates (Borders, TextBlocks, Panels). DevTools -> Visual tree shows the rendered hierarchy.
- Some controls (e.g., `ContentPresenter`) exist in the visual tree but not in the logical tree. When `FindControl` fails, confirm whether the element is in the logical tree.
- Reference implementation: LogicalTreeExtensions.cs and Visual.cs.

9. Data templates explained

- `ItemsControl.ItemTemplate` applies a `DataTemplate` for each item. Inside a data template, the `DataContext` is the individual item (an `OrderViewModel`).
- You can inline XAML or reference a key: `<DataTemplate x:Key="OrderTemplate"> ...` and then `ItemTemplate="{StaticResource OrderTemplate}"`.
- Data templates can contain user controls, panels, or inline elements. They are the foundation for list virtualization later.
- Template source: DataTemplate.cs.

10. Run, inspect, and iterate

```
dotnet run
```

While the app runs: - Press **F12** (DevTools). Explore both logical and visual trees for `OrderRow` entries. - Select an `OrderRow` TextBlock and confirm the binding path (`Total`) resolves to the right data. - Try editing `OrderViewModel` values in code and rerun to see updates.

Troubleshooting

- **Binding path errors**: DevTools -> Diagnostics -> Binding Errors shows typos. Ensure properties exist or set `x:DataType="vm:OrderViewModel"` in templates for compile-time checks (once you add namespaces for view models).
- **Converter not found**: ensure the namespace prefix in `App.axaml` matches the converter's CLR namespace and the key matches `StaticResource CurrencyConverter`.
- **User control not rendering**: confirm the namespace prefix `xmlns:ui` matches the CLR namespace of `OrderRow` and that the class is `partial` with matching `x:Class`.
- **FindControl returns null**: check `NameScope`. If the element is inside a data template, use `e.Source` from events or bind through the ViewModel instead of searching.

Practice and validation

1. Add a `ui:AddressCard` user control showing billing address details. Bind it to `Customer` using `ContentControl.Content="{Binding Customer}"` and define a data template for `CustomerViewModel`.
2. Add a `ValueConverter` that highlights orders above $500 by returning a different brush; apply it to the Border background via `{Binding Total, Converter=...}`.
3. Add a `ListBox` instead of `ItemsControl` and observe how selection adds visual states in the visual tree.
4. Use DevTools to inspect both logical and visual trees for the `AddressCard`. Note which elements appear in one tree but not the other.

Look under the hood (source bookmarks)

- Content control composition: src/Avalonia.Controls/ContentControl.cs
- User controls and name scopes: src/Avalonia.Controls/UserControl.cs
- Logical tree helpers: src/Avalonia.Base/LogicalTree/LogicalTreeExtensions.cs
- Data template implementation: src/Markup/Avalonia.Markup.Xaml/Templates/DataTemplate.cs
- Value converters: src/Avalonia.Base/Data/Converters

Check yourself

- How do XAML namespaces (`xmlns`) relate to CLR namespaces and assemblies?
- What is the difference between the logical and visual tree, and why does it matter for bindings?
- How do `ContentControl` and `UserControl` differ and when would you choose each?
- Where do you register value converters so they can be referenced in XAML?
- Inside a `DataTemplate`, what object provides the `DataContext`?

What's next - Next: Chapter 4

4. Application startup: AppBuilder and lifetimes

Goal - Trace the full AppBuilder pipeline from `Program.Main` to the first window or view. - Understand how each lifetime (`ClassicDesktopStyleApplicationLifetime`, `SingleViewApplicationLifetime`, `BrowserSingleViewLifetime`, `HeadlessApplicationLifetime`) boots and shuts down your app. - Learn where to register services, logging, and global configuration before the UI appears. - Handle startup exceptions gracefully and log early so failures are diagnosable. - Prepare a project that can swap between desktop, mobile/browser, and headless test lifetimes.

Why this matters - The startup path decides which platforms you can target and where dependency injection, logging, and configuration happen. - Knowing the lifetime contracts keeps your code organised when you add secondary windows, mobile navigation, or browser shells later. - Understanding the AppBuilder steps helps you debug platform issues (e.g., missing native dependencies or misconfigured rendering).

Prerequisites - You have completed Chapter 2 and can build/run a template project. - You are comfortable editing `Program.cs`, `App.axaml`, and `App.axaml.cs`.

1. Follow the AppBuilder pipeline step by step

`Program.cs` (or `Program.fs` in F#) is the entry point. A typical template looks like this:

```
using Avalonia;
using Avalonia.ReactiveUI; // optional in ReactiveUI template

internal static class Program
{
    [STAThread]
    public static void Main(string[] args) => BuildAvaloniaApp()
        .StartWithClassicDesktopLifetime(args);

    public static AppBuilder BuildAvaloniaApp()
        => AppBuilder.Configure<App>()       // 1. Choose your Application subclass
            .UsePlatformDetect()             // 2. Detect the right native backend (Win32, macOS, X11, 
            .UseSkia()                       // 3. Configure the rendering pipeline (Skia GPU/CPU render
            .With(new SkiaOptions {          // 4. (Optional) tweak renderer settings
                MaxGpuResourceSizeBytes = 96 * 1024 * 1024
            })
            .LogToTrace()                    // 5. Hook logging before startup completes
            .UseReactiveUI();                // 6. (Optional) enable ReactiveUI integration
}
```

Each call returns the builder so you can chain configuration. Relevant source: - `AppBuilder` implementation: `src/Avalonia.Controls/AppBuilder.cs` - Skia configuration: `src/Skia/Avalonia.Skia/SkiaOptions.cs` - Desktop helpers (`StartWithClassicDesktopLifetime`): `src/Avalonia.Desktop/AppBuilderDesktopExtensions.cs`

Builder pipeline diagram (mental map)

```
Program.Main
  `--- BuildAvaloniaApp()
        |-- Configure<App>()        (create Application instance)
        |-- UsePlatformDetect()     (choose backend)
        |-- UseSkia()/UseReactiveUI (features)
        |-- LogToTrace()/With(...)  (diagnostics/options)
        `-- StartWith...Lifetime()  (select lifetime and enter main loop)
```

If anything in the pipeline throws, the process exits before UI renders. Log early to catch those cases.

2. Lifetimes in detail

Lifetime type	Purpose	Typical targets	Key members
`ClassicDesktopStyleApplicationLifetime`	Windowed desktop apps with startup/shutdown events and main window	Windows, macOS, Linux	`MainWindow`, `ShutdownMode`, `Exit`, `OnExit`
`SingleViewApplicationLifetime`	Hosts a single root control (`MainView`)	Android, iOS, Embedded	`MainView`, `MainViewClosing`, `OnMainViewClosed`
`BrowserSingleViewLifetime` (implements `ISingleViewApplicationLifetime`)	Same contract as single view, tuned for WebAssembly	Browser (WASM)	`MainView`, async app init
`HeadlessApplicationLifetime`	No visible UI; runs for tests or background services	Unit/UI tests	`TryGetTopLevel()`, manual pumping

Key interfaces and classes to read: - Desktop lifetime: `ClassicDesktopStyleApplicationLifetime.cs` - Single view lifetime: `SingleViewApplicationLifetime.cs` - Browser lifetime: `BrowserSingleViewLifetime.cs` - Headless lifetime: `src/Headless/Avalonia.Headless/AvaloniaHeadlessApplicationLifetime.cs`

3. Wiring lifetimes in `App.OnFrameworkInitializationCompleted`

`App.axaml.cs` is the right place to react once the framework is ready:

```
using Avalonia;
using Avalonia.Controls.ApplicationLifetimes;
using Microsoft.Extensions.DependencyInjection; // if using DI

namespace MultiLifetimeSample;

public partial class App : Application
{
    private IServiceProvider? _services;

    public override void Initialize()
        => AvaloniaXamlLoader.Load(this);

    public override void OnFrameworkInitializationCompleted()
    {
        // Create/register services only once
        _services ??= ConfigureServices();

        if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
        {
            var shell = _services.GetRequiredService<MainWindow>();
            desktop.MainWindow = shell;
            desktop.Exit += (_, _) => _services.Dispose();
        }
        else if (ApplicationLifetime is ISingleViewApplicationLifetime singleView)
        {
            singleView.MainView = _services.GetRequiredService<MainView>();
        }
        else if (ApplicationLifetime is IControlledApplicationLifetime controlled)
```

```
        {
            controlled.Exit += (_, _) => Console.WriteLine("Application exited");
        }

        base.OnFrameworkInitializationCompleted();
    }

    private IServiceProvider ConfigureServices()
    {
        var services = new ServiceCollection();
        services.AddSingleton<MainWindow>();
        services.AddSingleton<MainView>();
        services.AddSingleton<DashboardViewModel>();
        services.AddLogging(builder => builder.AddDebug());
        return services.BuildServiceProvider();
    }
}
```

Notes: - `ApplicationLifetime` always implements `IControlledApplicationLifetime`, so you can subscribe to `Exit` for cleanup even if you do not know the exact subtype. - Use dependency injection (any container) to share views/view models. Avalonia does not ship a DI container, so you control the lifetime. - For headless tests, your `App` still runs but you typically return `SingleView` or host view models manually.

4. Handling exceptions and logging

Important logging points: - `AppBuilder.LogToTrace()` uses Avalonia's logging infrastructure (see `src/Avalonia.Base/Logging`). For production apps, plug in `Serilog`, `Microsoft.Extensions.Logging`, or your preferred provider. - Subscribe to `AppDomain.CurrentDomain.UnhandledException` and `TaskScheduler.UnobservedTaskException` inside `Main` to catch fatal issues before the dispatcher tears down.

Example:

```
[STAThread]
public static void Main(string[] args)
{
    AppDomain.CurrentDomain.UnhandledException += (_, e) => LogFatal(e.ExceptionObject);
    TaskScheduler.UnobservedTaskException += (_, e) => LogFatal(e.Exception);

    try
    {
        BuildAvaloniaApp().StartWithClassicDesktopLifetime(args);
    }
    catch (Exception ex)
    {
        LogFatal(ex);
        throw;
    }
}
```

`ClassicDesktopStyleApplicationLifetime` exposes `ShutdownMode` and `Shutdown()` so you can exit explicitly when critical failures occur.

5. Switching lifetimes inside one project

You can provide different entry points or compile-time switches:

```
public static void Main(string[] args)
{
#if HEADLESS
    BuildAvaloniaApp().Start(AppMain);
#elif BROWSER
    BuildAvaloniaApp().SetupBrowserApp("app");
#else
    BuildAvaloniaApp().StartWithClassicDesktopLifetime(args);
#endif
}
```

- `SetupBrowserApp` is defined in `BrowserAppBuilder.cs` and attaches the app to a DOM element.
- `Start` (with `AppMain`) lets you provide your own lifetime, often used in headless/integration tests.

6. Headless/testing scenarios

Avalonia's headless assemblies let you boot an app without rendering:

```
using Avalonia;
using Avalonia.Headless;

public static class Program
{
    public static void Main(string[] args)
        => BuildAvaloniaApp().StartWithHeadless(new HeadlessApplicationOptions
        {
            RenderingMode = HeadlessRenderingMode.None,
            UseHeadlessDrawingContext = true
        });
}
```

- `Avalonia.Headless` lives under `src/Headless` and powers automated UI tests (`Avalonia.Headless.XUnit`, `Avalonia.Headless.NUnit`).
- You can pump the dispatcher manually to run asynchronous UI logic in tests (`HeadlessUnitTestSession.Run` displays an example).

7. Putting it together: desktop + single-view sample

`Program.cs`:

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

[STAThread]
public static void Main(string[] args)
{
    if (args.Contains("--single-view"))
    {
        BuildAvaloniaApp().StartWithSingleViewLifetime(new MainView());
    }
    else
    {
        BuildAvaloniaApp().StartWithClassicDesktopLifetime(args);
```

```
    }
}
```

`App.axaml.cs` sets up both `MainWindow` and `MainView` (as shown earlier). At runtime, you can switch lifetimes via command-line or compile condition.

Troubleshooting

- **Black screen on startup**: check `UsePlatformDetect()`; on Linux you might need extra packages (mesa, libwebkit) or use `UseSkia` explicitly.
- **No window appearing**: ensure `desktop.MainWindow` is assigned before calling `base.OnFrameworkInitializationCo`
- **Single view renders but inputs fail**: confirm you used the right lifetime (`StartWithSingleViewLifetime`) and that your root view is a `Control` with focusable children.
- **DI container disposed too early**: if you `using` the provider, keep it alive for the app lifetime and dispose in `Exit`.
- **Unhandled exception after closing last window**: check `ShutdownMode`. Default is `OnLastWindowClose`; switch to `OnMainWindowClose` or call `Shutdown()` to exit on demand.

Practice and validation

1. Modify your project so the same `App` supports both desktop and single-view lifetimes. Use a command-line switch (`--mobile`) to select `StartWithSingleViewLifetime` and verify your `MainView` renders inside a mobile head (Android emulator or `dotnet run -- --mobile` + `SingleView` desktop simulation).
2. Register a logging provider using `Microsoft.Extensions.Logging`. Log the current lifetime type inside `OnFrameworkInitializationCompleted` and observe the output.
3. Add a simple DI container (as shown) and resolve `MainWindow`/`MainView` through it. Confirm disposal happens when the app exits.
4. Create a headless console entry point (`BuildAvaloniaApp().Start(AppMain)`) and run a unit test that constructs a view, invokes bindings, and pumps the dispatcher.
5. Intentionally throw inside `OnFrameworkInitializationCompleted` and observe how logging captures the stack. Then add a `try/catch` to show a fallback dialog or log and exit gracefully.

Look under the hood (source bookmarks)

- `AppBuilder` internals: `src/Avalonia.Controls/AppBuilder.cs`
- Desktop startup helpers: `src/Avalonia.Desktop/AppBuilderDesktopExtensions.cs`
- Desktop lifetime implementation: `src/Avalonia.Controls/ApplicationLifetimes/ClassicDesktopStyleApplicati`
- Single-view lifetime: `src/Avalonia.Controls/ApplicationLifetimes/SingleViewApplicationLifetime.cs`
- Browser lifetime: `src/Browser/Avalonia.Browser/BrowserSingleViewLifetime.cs`
- Headless lifetime and tests: `src/Headless`

Check yourself

- What steps does `BuildAvaloniaApp()` perform before choosing a lifetime?
- Which lifetime would you use for Windows/macOS, Android/iOS, browser, and automated tests?
- Where should you place dependency injection setup and where should you dispose the container?
- How can you capture and log unhandled exceptions thrown during startup?
- How would you attach the app to a DOM element in a WebAssembly host?

What's next - Next: Chapter 5

5. Layout system without mystery

Goal - Understand Avalonia's layout pass (`Measure` then `Arrange`) and how `Layoutable` and `LayoutManager` orchestrate it. - Master the core panels (StackPanel, Grid, DockPanel, WrapPanel) plus advanced tools (`GridSplitter`, `Viewbox`, `LayoutTransformControl`, `SharedSizeGroup`). - Learn when to create custom panels by overriding `MeasureOverride`/`ArrangeOverride`. - Know how scrolling, virtualization, and `Panel.ZIndex` interact with layout. - Practice diagnosing layout issues with DevTools overlays and logging.

Why this matters - Layout defines the user experience: predictable resizing, adaptive forms, responsive dashboards. - Panels are reusable building blocks. Understanding the underlying contract helps you read control templates and write your own. - Troubleshooting layout without a plan wastes time; with DevTools and knowledge of the pass order, you debug confidently.

Prerequisites - You can run a basic Avalonia app and edit XAML (Chapters 2-4). - You have DevTools (F12) available to inspect layout rectangles.

1. Mental model: measure and arrange

Every control inherits from `Layoutable` (Layoutable.cs). The layout pass runs in two stages:

1. **Measure**: Parent asks each child "How big would you like to be?" providing an available size. The child can respond with any size up to that constraint. Override `MeasureOverride` in panels to lay out children.
2. **Arrange**: Parent decides where to place each child within its final bounds. Override `ArrangeOverride` to position children based on the measured sizes.

The `LayoutManager` (LayoutManager.cs) schedules layout passes when controls invalidate measure or arrange (`InvalidateMeasure`, `InvalidateArrange`).

2. Start a layout playground project

```
dotnet new avalonia.app -o LayoutPlayground
cd LayoutPlayground
```

Replace `MainWindow.axaml` with an experiment playground that demonstrates the core panels and alignment tools:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="LayoutPlayground.MainWindow"
        Width="880" Height="560"
        Title="Layout Playground">
  <Grid ColumnDefinitions="*,*" RowDefinitions="Auto,*" Padding="16" RowSpacing="16" ColumnSpacing="16"
    <TextBlock Grid.ColumnSpan="2" Classes="h1" Text="Layout system without mystery"/>

    <StackPanel Grid.Row="1" Spacing="12">
      <TextBlock Classes="h2" Text="StackPanel"/>
      <Border BorderBrush="#CCC" BorderThickness="1" Padding="8">
        <StackPanel Spacing="6">
          <Button Content="Top"/>
          <Button Content="Middle"/>
          <Button Content="Bottom"/>
          <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="Last child fills remaining space"/>
          </Border>
        </DockPanel>
      </Border>
    </StackPanel>

    <StackPanel Grid.Column="1" Grid.Row="1" Spacing="12">
      <TextBlock Classes="h2" Text="Grid + WrapPanel"/>
      <Border BorderBrush="#CCC" BorderThickness="1" Padding="8">
        <Grid ColumnDefinitions="Auto,*" RowDefinitions="Auto,Auto,Auto" ColumnSpacing="8" RowSpacing="8
          <TextBlock Text="Name:"/>
          <TextBox Grid.Column="1" MinWidth="200"/>

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

          <TextBlock Grid.Row="2" Text="Notes:" VerticalAlignment="Top"/>
          <TextBox Grid.Row="2" Grid.Column="1" Height="80" AcceptsReturn="True" TextWrapping="Wrap"/>
        </Grid>
      </Border>

      <Border BorderBrush="#CCC" BorderThickness="1" Padding="8">
        <WrapPanel ItemHeight="32" MinWidth="200" ItemWidth="100" 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 the app and resize the window. Observe how StackPanel, DockPanel, Grid, and WrapPanel distribute space.

3. Alignment and sizing toolkit recap

- `Margin` vs `Padding`: Margin adds space around a control; Padding adds space inside a container.
- `HorizontalAlignment`/`VerticalAlignment`: `Stretch` makes controls fill available space; `Center`, `Start`, `End` align within the assigned slot.
- `Width`/`Height`: fixed sizes; use sparingly. Prefer `MinWidth`, `MaxWidth`, `MinHeight`, `MaxHeight` for adaptive layouts.
- Grid sizing: `Auto` (size to content), `*` (take remaining space), `2*` (take twice the share). Column/row definitions can mix Auto, star, and pixel values.

4. Advanced layout tools

Grid with SharedSizeGroup

`SharedSizeGroup` lets multiple grids share sizes within a scope. Mark the parent with `Grid.IsSharedSizeScope="True"`:

```
<Grid ColumnDefinitions="Auto,*" RowDefinitions="Auto,Auto" Grid.IsSharedSizeScope="True">
  <Grid.ColumnDefinitions>
    <ColumnDefinition SharedSizeGroup="Label"/>
    <ColumnDefinition Width="*"/>
  </Grid.ColumnDefinitions>
  <Grid RowDefinitions="Auto,Auto" ColumnDefinitions="Auto,*">
    <TextBlock Text="First" Grid.Column="0"/>
    <TextBox Grid.Column="1" MinWidth="200"/>
  </Grid>
  <Grid Grid.Row="1" ColumnDefinitions="Auto,*">
    <TextBlock Text="Second" Grid.Column="0"/>
    <TextBox Grid.Column="1" MinWidth="200"/>
  </Grid>
</Grid>
```

All label columns share the same width. Source: `Grid.cs` and `DefinitionBase.cs`.

GridSplitter

```
<Grid ColumnDefinitions="3*,Auto,2*">
  <StackPanel Grid.Column="0">...</StackPanel>
  <GridSplitter Grid.Column="1" Width="6" ShowsPreview="True" Background="#DDD"/>
  <StackPanel Grid.Column="2">...</StackPanel>
</Grid>
```

`GridSplitter` lets users resize star-sized columns/rows. Implementation: `GridSplitter.cs`.

Viewbox and LayoutTransformControl

- `Viewbox` scales its child proportionally to fit the available space.
- `LayoutTransformControl` applies transforms (rotate, scale, skew) while preserving layout.

```
<Viewbox Stretch="Uniform" Width="200" Height="200">
  <TextBlock Text="Scaled" FontSize="24"/>
</Viewbox>

<LayoutTransformControl>
  <LayoutTransformControl.LayoutTransform>
    <RotateTransform Angle="-10"/>
  </LayoutTransformControl.LayoutTransform>
  <Border Padding="12" Background="#E7F1FF">
    <TextBlock Text="Rotated layout"/>
  </Border>
</LayoutTransformControl>
```

Sources: `Viewbox.cs`, `LayoutTransformControl.cs`.

Panel.ZIndex

Controls inside the same panel respect `Panel.ZIndex` for stacking order. Higher ZIndex renders above lower values.

```
<Canvas>
  <Rectangle Width="100" Height="80" Fill="#60FF0000" Panel.ZIndex="1"/>
  <Rectangle Width="120" Height="60" Fill="#6000FF00" Panel.ZIndex="2" Margin="20,10,0,0"/>
</Canvas>
```

5. Scrolling and LogicalScroll

`ScrollViewer` wraps content to provide scrolling. When the child implements `ILogicalScrollable` (e.g., `ItemsPresenter` with virtualization), the scrolling is smoother and can skip measurement of offscreen content.

```
<ScrollViewer HorizontalScrollBarVisibility="Auto" VerticalScrollBarVisibility="Auto">
  <StackPanel>

  </StackPanel>
</ScrollViewer>
```

- For virtualization, panels may implement `ILogicalScrollable` (see `LogicalScroll.cs`).
- `ScrollViewer` triggers layout when viewports change.

6. Custom panels (when the built-ins aren't enough)

Derive from `Panel` and override `MeasureOverride`/`ArrangeOverride` to create custom layout logic. Example: a simplified `UniformGrid`:

```
using Avalonia;
using Avalonia.Controls;
using Avalonia.Layout;

namespace LayoutPlayground.Controls;

public class UniformGridPanel : Panel
{
    public static readonly StyledProperty<int> ColumnsProperty =
        AvaloniaProperty.Register<UniformGridPanel, int>(nameof(Columns), 2);

    public int Columns
    {
        get => GetValue(ColumnsProperty);
        set => SetValue(ColumnsProperty, value);
    }

    protected override Size MeasureOverride(Size availableSize)
    {
        foreach (var child in Children)
        {
            child.Measure(Size.Infinity);
        }

        var rows = (int)Math.Ceiling(Children.Count / (double)Columns);
        var cellWidth = availableSize.Width / Columns;
        var cellHeight = availableSize.Height / rows;

        return new Size(cellWidth * Columns, cellHeight * rows);
    }
```

```
    protected override Size ArrangeOverride(Size finalSize)
    {
        var rows = (int)Math.Ceiling(Children.Count / (double)Columns);
        var cellWidth = finalSize.Width / Columns;
        var cellHeight = finalSize.Height / rows;

        for (var index = 0; index < Children.Count; index++)
        {
            var child = Children[index];
            var row = index / Columns;
            var column = index % Columns;
            var rect = new Rect(column * cellWidth, row * cellHeight, cellWidth, cellHeight);
            child.Arrange(rect);
        }

        return finalSize;
    }
}
```

- This panel ignores child desired sizes for simplicity; real panels usually respect `child.DesiredSize` from `Measure`.
- Read `Layoutable` and `Panel` sources to understand helper methods like `ArrangeRect`.

7. Layout diagnostics with DevTools

While running the app press **F12** -> Layout tab: - Inspect the measurement and arrange rectangles for each control. - Toggle the Layout Bounds overlay to visualise margins and paddings. - Use the Render Options overlay to show dirty rectangles (requires enabling `RendererDebugOverlays` in code: see `RendererDebugOverlays.cs`).

You can also enable layout logging:

```
AppBuilder.Configure<App>()
    .UsePlatformDetect()
    .LogToTrace(LogEventLevel.Debug, new[] { LogArea.Layout })
    .StartWithClassicDesktopLifetime(args);
```

`LogArea.Layout` logs measure/arrange operations to the console.

8. Practice scenarios

1. **Shared field labels**: Use `Grid.IsSharedSizeScope` and `SharedSizeGroup` across multiple form sections so labels align perfectly, even when collapsed sections are toggled.
2. **Resizable master-detail**: Combine `GridSplitter` with a two-column layout; ensure minimum sizes keep content readable.
3. **Rotated card**: Wrap a Border in `LayoutTransformControl` to rotate it; evaluate how alignment behaves inside the transform.
4. **Custom panel**: Replace a WrapPanel with your `UniformGridPanel` and compare measurement behaviour in DevTools.
5. **Scroll diagnostics**: Place a long list inside `ScrollViewer`, enable DevTools Layout overlay, and observe how viewport size changes the arrange rectangles.

Look under the hood (source bookmarks)

- Base layout contract: `Layoutable.cs`
- Layout manager: `LayoutManager.cs`

- Grid + shared size: `Grid.cs`, `DefinitionBase.cs`
- Layout transforms: `LayoutTransformControl.cs`
- Scroll infrastructure: `ScrollViewer.cs`, `LogicalScroll.cs`
- Custom panels inspiration: `VirtualizingStackPanel.cs`

Check yourself

- What two steps does the layout system run for every control, and which classes coordinate them?
- How does `SharedSizeGroup` influence multiple grids? What property enables shared sizing?
- When would you use `LayoutTransformControl` instead of a render transform?
- What happens if you change `Panel.ZIndex` for children inside the same panel?
- How can DevTools and logging help you diagnose a control that does not appear where expected?

What's next - Next: Chapter 6

6. Controls tour you'll actually use

Goal - Build confidence with Avalonia's everyday controls grouped by scenario: text input, selection, navigation, editing, and feedback. - Learn how to bind controls to view models, template items, and customise interaction states. - Discover specialised controls such as `NumericUpDown`, `MaskedTextBox`, `AutoCompleteBox`, `ColorPicker`, `TreeView`, `TabControl`, and `SplitView`. - Understand selection models, virtualization, and templating so large lists stay responsive. - Know where to find styles, templates, and extension points in the source code.

Why this matters - Real apps mix many controls on the same screen. Understanding their behaviour and key properties saves time. - Avalonia's control set is broad; learning the structure of templates and selection models prepares you for customisation later.

Prerequisites - You have built layouts (Chapter 5) and can bind data (Chapter 3's data templates). Chapter 8 will deepen bindings further.

1. Set up a sample project

```
dotnet new avalonia.mvvm -o ControlsShowcase
cd ControlsShowcase
```

We will extend `Views/MainWindow.axaml` with multiple sections backed by `MainWindowViewModel`.

2. Form inputs and validation basics

```
<StackPanel Spacing="16">
  <TextBlock Classes="h1" Text="Customer profile"/>

  <Grid ColumnDefinitions="Auto,*" RowDefinitions="Auto,Auto,Auto" RowSpacing="8" ColumnSpacing="12">
    <TextBlock Text="Name:"/>
    <TextBox Grid.Column="1" Text="{Binding Customer.Name}" Watermark="Full name"/>

    <TextBlock Grid.Row="1" Text="Email:"/>
    <TextBox Grid.Row="1" Grid.Column="1" Text="{Binding Customer.Email}"/>

    <TextBlock Grid.Row="2" Text="Phone:"/>
    <MaskedTextBox Grid.Row="2" Grid.Column="1" Mask="(000) 000-0000" Value="{Binding Customer.Phone}"/
  </Grid>

  <StackPanel Orientation="Horizontal" Spacing="12">
    <NumericUpDown Width="120" Minimum="0" Maximum="20" Value="{Binding Customer.Seats}" Header="Seats"
    <DatePicker SelectedDate="{Binding Customer.RenewalDate}" Header="Renewal"/>
  </StackPanel>
</StackPanel>
```

Notes: - `MaskedTextBox` lives in `Avalonia.Controls` (see `MaskedTextBox.cs`) and enforces input patterns.
- `NumericUpDown` (from `NumericUpDown.cs`) provides spinner buttons and numeric formatting.

3. Toggles, options, and commands

```
<GroupBox Header="Plan options" Padding="12">
  <StackPanel Spacing="8">
    <ToggleSwitch Header="Enable auto-renew" IsChecked="{Binding Customer.AutoRenew}"/>

    <StackPanel Orientation="Horizontal" Spacing="12">
      <CheckBox Content="Include analytics" IsChecked="{Binding Customer.IncludeAnalytics}"/>
      <CheckBox Content="Priority support" IsChecked="{Binding Customer.IncludeSupport}"/>
```

```
    </StackPanel>

    <StackPanel Orientation="Horizontal" Spacing="12">
      <RadioButton GroupName="Plan" Content="Starter" IsChecked="{Binding Customer.IsStarter}"/>
      <RadioButton GroupName="Plan" Content="Growth" IsChecked="{Binding Customer.IsGrowth}"/>
      <RadioButton GroupName="Plan" Content="Enterprise" IsChecked="{Binding Customer.IsEnterprise}"/>
    </StackPanel>

    <Button Content="Save" HorizontalAlignment="Left" Command="{Binding SaveCommand}"/>
  </StackPanel>
</GroupBox>
```

- `ToggleSwitch` gives a Fluent-styled toggle. Implementation: `ToggleSwitch.cs`.
- RadioButtons share state via `GroupName` or `IsChecked` bindings.

4. Selection lists with templating

```
<GroupBox Header="Teams" Padding="12">
  <ListBox Items="{Binding Teams}" SelectedItem="{Binding SelectedTeam}" Height="160">
    <ListBox.ItemTemplate>
      <DataTemplate>
        <StackPanel Orientation="Horizontal" Spacing="12">
          <Ellipse Width="24" Height="24" Fill="{Binding Color}"/>
          <TextBlock Text="{Binding Name}" FontWeight="SemiBold"/>
        </StackPanel>
      </DataTemplate>
    </ListBox.ItemTemplate>
  </ListBox>
</GroupBox>
```

- `ListBox` supports selection out of the box. For custom selection logic, use `SelectionModel` (see `SelectionModel.cs`).
- Consider `ListBox.SelectionMode="Multiple"` for multi-select.

Virtualization tip

Large lists should virtualize. Use `ListBox` with the default `VirtualizingStackPanel` or switch panels:

```
<ListBox Items="{Binding ManyItems}" VirtualizingPanel.IsVirtualizing="True" VirtualizingPanel.CacheLeng
```

Controls for virtualization: `VirtualizingStackPanel.cs`.

5. Hierarchical data with `TreeView`

```
<TreeView Items="{Binding Departments}" SelectedItems="{Binding SelectedDepartments}">
  <TreeView.ItemTemplate>
    <TreeDataTemplate ItemsSource="{Binding Teams}">
      <TextBlock Text="{Binding Name}" FontWeight="SemiBold"/>
      <TreeDataTemplate.ItemTemplate>
        <DataTemplate>
          <TextBlock Text="{Binding Name}" Margin="24,0,0,0"/>
        </DataTemplate>
      </TreeDataTemplate.ItemTemplate>
    </TreeDataTemplate>
  </TreeView.ItemTemplate>
</TreeView>
```

- `TreeView` uses `TreeDataTemplate` to describe hierarchical data. Each template can reference a property (`Teams`) for child items.
- Source implementation: `TreeView.cs`.

6. Navigation controls (`TabControl`, `SplitView`, `Expander`)

```
<TabControl SelectedIndex="{Binding SelectedTab}">
  <TabItem Header="Overview">
    <TextBlock Text="Overview content" Margin="12"/>
  </TabItem>
  <TabItem Header="Reports">
    <TextBlock Text="Reports content" Margin="12"/>
  </TabItem>
  <TabItem Header="Settings">
    <TextBlock Text="Settings content" Margin="12"/>
  </TabItem>
</TabControl>

<SplitView DisplayMode="CompactInline"
          IsPaneOpen="{Binding IsPaneOpen}"
          OpenPaneLength="240" CompactPaneLength="56">
  <SplitView.Pane>
    <NavigationViewContent/>
  </SplitView.Pane>
  <SplitView.Content>
    <Frame Content="{Binding ActivePage}"/>
  </SplitView.Content>
</SplitView>

<Expander Header="Advanced filters" IsExpanded="False">
  <StackPanel Margin="12" Spacing="8">
    <ComboBox Items="{Binding FilterSets}" SelectedItem="{Binding SelectedFilter}"/>
    <CheckBox Content="Include archived" IsChecked="{Binding IncludeArchived}"/>
  </StackPanel>
</Expander>
```

- `TabControl` enables tabbed navigation. Tab headers are content–you can template them via `TabControl.ItemTemplate`.
- `SplitView` (from `SplitView.cs`) provides collapsible navigation, useful for sidebars.
- `Expander` collapses/expands content. Implementation: `Expander.cs`.

7. Auto-complete, pickers, and dialogs

```
<StackPanel Spacing="12">
  <AutoCompleteBox Width="240"
                   Items="{Binding Suggestions}"
                   Text="{Binding Query, Mode=TwoWay}">
    <AutoCompleteBox.ItemTemplate>
      <DataTemplate>
        <StackPanel Orientation="Horizontal" Spacing="8">
          <TextBlock Text="{Binding Icon}"/>
          <TextBlock Text="{Binding Title}"/>
        </StackPanel>
      </DataTemplate>
    </AutoCompleteBox.ItemTemplate>
```

```
  </AutoCompleteBox>

  <ColorPicker SelectedColor="{Binding ThemeColor}"/>

  <Button Content="Choose files" Command="{Binding OpenFilesCommand}"/>
</StackPanel>
```

- `AutoCompleteBox` helps with large suggestion lists. Source: `AutoCompleteBox.cs`.
- `ColorPicker` shows palettes, sliders, and input fields (see `ColorPicker.cs`).
- File pickers will use `IStorageProvider` (Chapter 16).

8. Feedback and status

```
<StatusBar>
  <StatusBarItem>
    <StackPanel Orientation="Horizontal" Spacing="8">
      <TextBlock Text="Ready"/>
      <ProgressBar Width="120" IsIndeterminate="{Binding IsBusy}"/>
    </StackPanel>
  </StatusBarItem>
  <StatusBarItem HorizontalAlignment="Right">
    <TextBlock Text="v1.2.0"/>
  </StatusBarItem>
</StatusBar>

<NotificationCard Width="320" IsOpen="{Binding ShowNotification}" Title="Update available" Description=
```

- `StatusBar` and `NotificationCard` (Fluent template) provide feedback surfaces.

9. Styling, classes, and visual states

Use classes (`Classes="primary"`) or pseudo-classes (`:pointerover`, `:pressed`, `:checked`) to style stateful controls:

```
<Button Content="Primary" Classes="primary"/>
```

```
<Style Selector="Button.primary">
  <Setter Property="Background" Value="{DynamicResource AccentBrush}"/>
  <Setter Property="Foreground" Value="White"/>
</Style>

<Style Selector="Button.primary:pointerover">
  <Setter Property="Background" Value="{DynamicResource AccentBrush2}"/>
</Style>
```

Styles live in `App.axaml` or separate resource dictionaries. Control templates are defined under `src/Avalonia.Themes.Fluent`. Inspect `Button.xaml`, `ListBox.xaml`, etc., to understand structure and visual states.

10. ControlCatalog treasure hunt

1. Clone the Avalonia repository and run the ControlCatalog (Desktop) sample: `dotnet run --project samples/ControlCatalog.Desktop/ControlCatalog.Desktop.csproj`.
2. Use the built-in search to find controls. Explore the `Source` tab to jump to relevant XAML or C# files.
3. Compare ControlCatalog pages with the source directory structure:
   - Text input demos map to `src/Avalonia.Controls/TextBox.cs`.

- Collections and virtualization demos map to `VirtualizingStackPanel.cs`.
- Navigation samples map to `SplitView.cs` and `TabControl` templates.

11. Practice exercises

1. Create a "dashboard" page mixing text input, selection lists, tabs, and a collapsible filter panel. Bind every control to a view model.
2. Add an `AutoCompleteBox` that filters as you type. Use DevTools to inspect the generated `ListBox` inside the control.
3. Replace the `ListBox` with a `TreeView` for hierarchical data; add an `Expander` per root item.
4. Customise button states by adding pseudo-class styles. Confirm they match the ControlCatalog defaults.
5. Swap the `WrapPanel` for an `ItemsRepeater` (Chapter 14) to prepare for virtualization scenarios.

Look under the hood (source bookmarks)

- Core controls: `src/Avalonia.Controls`
- Specialized controls: `src/Avalonia.Controls.ColorPicker`, `src/Avalonia.Controls.NumericUpDown`, `src/Avalonia.Controls.AutoCompleteBox`
- Selection framework: `src/Avalonia.Controls/Selection`
- Styles and templates: `src/Avalonia.Themes.Fluent/Controls`

Check yourself

- Which controls would you choose for numeric input, masked input, and auto-completion?
- How do you template `ListBox` items and enable virtualization for large datasets?
- Where do you look to customise the appearance of a `ToggleSwitch`?
- What role does `SelectionModel` play for advanced selection scenarios?
- How can ControlCatalog help you explore a control's API and default styles?

What's next - Next: Chapter 7

7. Fluent theming and styles made simple

Goal - Understand Avalonia's Fluent theme architecture, theme variants, and how theme resources flow through your app. - Organise resources and styles with `ResourceInclude`, `StyleInclude`, `ThemeVariantScope`, and `ControlTheme` for clean reuse. - Override control templates, use pseudo-classes, and scope theme changes to specific regions. - Support runtime theme switching (light/dark/high contrast) and accessibility requirements. - Map the styles you edit to the Fluent source files so you can explore defaults and extend them safely.

Why this matters - Styling controls consistently is the difference between a polished UI and visual chaos. - Avalonia's Fluent theme ships with rich resources; knowing how to extend them keeps your design system maintainable. - Accessibility requirements (contrast, theming per surface) are easier when you understand theme scoping and dynamic resources.

Prerequisites - Comfort editing `App.axaml`, windows, and user controls (Chapters 3-6). - Basic understanding of data binding and commands (Chapters 3, 6).

1. Fluent theme in a nutshell

Avalonia ships with Fluent 2 based resources and templates. The theme lives under `src/Avalonia.Themes.Fluent`. Templates reference resource keys (brushes, thicknesses, typography) that resolve per theme variant.

`App.axaml` typically looks like this:

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

- `RequestedThemeVariant` controls the global variant (`ThemeVariant.Light`, `ThemeVariant.Dark`, `ThemeVariant.HighContrast`).
- `FluentTheme` can be configured with `Mode="Light"`, `Mode="Dark"`, or `Mode="Default"` (auto based on OS hints). Source: `FluentTheme.cs`.

2. Structure resources into dictionaries

Split large resource sets into dedicated files. Create `Styles/Colors.axaml`:

```
<ResourceDictionary xmlns="https://github.com/avaloniaui">
  <Color x:Key="BrandPrimaryColor">#2563EB</Color>
  <Color x:Key="BrandPrimaryHover">#1D4ED8</Color>

  <SolidColorBrush x:Key="BrandPrimaryBrush"
                   Color="{DynamicResource BrandPrimaryColor}"/>
  <SolidColorBrush x:Key="BrandPrimaryHoverBrush"
                   Color="{DynamicResource BrandPrimaryHover}"/>
</ResourceDictionary>
```

Then create `Styles/Controls.axaml`:

```
<Styles xmlns="https://github.com/avaloniaui">
  <Style Selector="Button.primary">
    <Setter Property="Background" Value="{DynamicResource BrandPrimaryBrush}"/>
    <Setter Property="Foreground" Value="White"/>
```

```
    <Setter Property="Padding" Value="14,10"/>
    <Setter Property="CornerRadius" Value="6"/>
  </Style>

  <Style Selector="Button.primary:pointerover">
    <Setter Property="Background" Value="{DynamicResource BrandPrimaryHoverBrush}"/>
  </Style>
</Styles>
```

Include them in `App.axaml`:

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

- `ResourceInclude` expects a `ResourceDictionary` root.
- `StyleInclude` expects `Styles` or a single `Style` root.

3. Static vs dynamic resources

- `StaticResource` resolves once during load. Use it for values that never change (fonts, corner radius constants).
- `DynamicResource` re-evaluates when the resource is replaced at runtime–essential for theme switching.

```
<Border CornerRadius="{StaticResource CornerRadiusMedium}"
        Background="{DynamicResource BrandPrimaryBrush}"/>
```

Resource lookup order: control -> logical parents -> window -> application -> Fluent theme dictionaries. Source: `ResourceDictionary.cs`.

4. Theme variant scope (local theming)

`ThemeVariantScope` lets you apply a specific theme to part of the UI. Implementation: `ThemeVariantScope.cs`.

```
<ThemeVariantScope RequestedThemeVariant="Dark">
  <Border Padding="16">
    <StackPanel>
      <TextBlock Classes="h2" Text="Dark section"/>
      <Button Content="Dark themed button" Classes="primary"/>
    </StackPanel>
  </Border>
</ThemeVariantScope>
```

Everything inside the scope resolves resources as if the app were using `ThemeVariant.Dark`. Useful for popovers or modal sheets.

5. Runtime theme switching

Add a toggle to your main view:

```
<ToggleSwitch Content="Dark mode" IsChecked="{Binding IsDark}"/>
```

In the view model:

```
using Avalonia;
using Avalonia.Styling;

public sealed class ShellViewModel : ObservableObject
{
    private bool _isDark;
    public bool IsDark
    {
        get => _isDark;
        set
        {
            if (SetProperty(ref _isDark, value))
            {
                Application.Current!.RequestedThemeVariant = value ? ThemeVariant.Dark : ThemeVariant.Li
            }
        }
    }
}
```

Because button styles use `DynamicResource`, they respond immediately. For per-window overrides set `RequestedThemeVariant` on the window itself or wrap content in `ThemeVariantScope`.

6. Customizing control templates with `ControlTheme`

`ControlTheme` lets you replace a control's default template and resources without subclassing. Source: `ControlTheme.cs`.

Example: create a pill-shaped toggle button theme in `Styles/ToggleButton.axaml`:

```
<ResourceDictionary xmlns="https://github.com/avaloniaui"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                    xmlns:themes="clr-namespace:Avalonia.Themes.Fluent;assembly=Avalonia.Themes.Fluent"
  <ControlTheme x:Key="PillToggleTheme" TargetType="ToggleButton">
    <Setter Property="Template">
      <ControlTemplate>
        <Border x:Name="PART_Root"
                Background="{TemplateBinding Background}"
                CornerRadius="20"
                Padding="{TemplateBinding Padding}">
          <ContentPresenter HorizontalAlignment="Center"
                            VerticalAlignment="Center"
                            Content="{TemplateBinding Content}"/>
        </Border>
      </ControlTemplate>
    </Setter>
  </ControlTheme>
</ResourceDictionary>
```

Apply it:

```
<ToggleButton Content="Pill" Theme="{StaticResource PillToggleTheme}" padding="12,6"/>
```

To inherit Fluent visual states, you can base your theme on existing resources by referencing `themes:ToggleButtonTheme`. Inspect templates in `src/Avalonia.Themes.Fluent/Controls` for structure and named parts.

7. Working with pseudo-classes and classes

Use pseudo-classes to target interaction states. Example for `ToggleSwitch`:

```
<Style Selector="ToggleSwitch:checked">
  <Setter Property="ThumbBrush" Value="{DynamicResource BrandPrimaryBrush}"/>
</Style>

<Style Selector="ToggleSwitch:checked:focus">
  <Setter Property="BorderBrush" Value="{DynamicResource BrandPrimaryHoverBrush}"/>
</Style>
```

Pseudo-class documentation lives in `Selectors.md` and runtime code under `Selector.cs`.

8. Accessibility and high contrast themes

Fluent ships high contrast resources. Switch by setting `RequestedThemeVariant="HighContrast"`.

- Provide alternative color dictionaries with increased contrast ratios.
- Use `DynamicResource` for all brushes so high contrast palettes propagate automatically.
- Test with screen readers and OS high contrast modes; ensure custom colors respect `ThemeVariant.HighContrast`.

Example dictionary addition:

```
<ResourceDictionary ThemeVariant="HighContrast"
                    xmlns="https://github.com/avaloniaui">
  <SolidColorBrush x:Key="BrandPrimaryBrush" Color="#00AACC"/>
  <SolidColorBrush x:Key="BrandPrimaryHoverBrush" Color="#007C99"/>
</ResourceDictionary>
```

`ThemeVariant`-specific dictionaries override defaults when the variant matches.

9. Debugging styles with DevTools

Press **F12** to open DevTools -> Styles panel: - Inspect applied styles, pseudo-classes, and resources. - Use the palette to modify brushes live and copy the generated XAML. - Toggle the `ThemeVariant` dropdown in DevTools (bottom) to preview Light/Dark/HighContrast variants.

Enable style diagnostics via logging:

```
AppBuilder.Configure<App>()
    .UsePlatformDetect()
    .LogToTrace(LogEventLevel.Debug, new[] { LogArea.Binding, LogArea.Styling })
    .StartWithClassicDesktopLifetime(args);
```

10. Practice exercises

1. **Create a brand palette**: define primary and secondary brushes with theme-specific overrides (light/dark/high contrast) and apply them to buttons and toggles.
2. **Scope a sub-view**: wrap a settings pane in `ThemeVariantScope RequestedThemeVariant="Dark"` to preview dual-theme experiences.
3. **Control template override**: create a `ControlTheme` for `Button` that changes the visual tree (e.g., adds an icon placeholder) and apply it selectively.
4. **Runtime theme switching**: wire a `ToggleSwitch` or menu command to flip between Light/Dark; ensure all custom brushes use `DynamicResource`.
5. **DevTools audit**: use DevTools to inspect pseudo-classes on a `ToggleSwitch` and verify your custom styles apply in `:checked` and `:focus` states.

Look under the hood (source bookmarks)

- Theme variant scoping: `ThemeVariantScope.cs`
- Control themes and styles: `ControlTheme.cs`, `Style.cs`
- Fluent resources and templates: `src/Avalonia.Themes.Fluent/Controls`
- Theme variant definitions: `ThemeVariant.cs`

Check yourself

- How do `ResourceInclude` and `StyleInclude` differ, and what root elements do they expect?
- When should you use `ThemeVariantScope` versus changing `RequestedThemeVariant` on the application?
- What advantages does `ControlTheme` give over subclassing a control?
- Why do you prefer `DynamicResource` for brushes that change with theme switches?
- Where would you inspect the default template for `ToggleSwitch` or `ComboBox`?

What's next - Next: Chapter 8

8. Data binding basics you'll use every day

Goal - Understand the binding engine (DataContext, binding paths, inheritance) and when to use different binding modes. - Work with binding variations (`Binding`, `CompiledBinding`, `MultiBinding`, `PriorityBinding`, `ElementName`, `RelativeSource`). - Connect collections to `ItemsControl`/`ListBox` with data templates and selection models. - Use converters, validation (`INotifyDataErrorInfo`), and asynchronous bindings for real-world scenarios. - Diagnose bindings using Avalonia's DevTools and `BindingDiagnostics` logging.

Why this matters - Bindings keep UI and data in sync, reducing boilerplate and keeping views declarative. - Picking the right binding technique (compiled, multi-value, priority) improves performance and readability. - Diagnostics help track down "binding isn't working" issues quickly.

Prerequisites - You can create a project and run it (Chapters 2-7). - You've seen basic controls and templates (Chapters 3 & 6).

1. The binding engine at a glance

Avalonia's binding engine lives under `src/Avalonia.Base/Data`. Key pieces: - `DataContext`: inherited down the logical tree. Most bindings resolve relative to the current element's DataContext. - `Binding`: describes a path, mode, converter, fallback, etc. - `BindingBase`: base for compiled bindings, multi bindings, priority bindings. - `BindingExpression`: runtime evaluation created for each binding target.

Bindings resolve in this order: 1. Find the source (DataContext, element name, relative source, etc.). 2. Evaluate the path (e.g., `Customer.Name`). 3. Apply converters or string formatting. 4. Update the target property according to the binding mode.

2. Set up the sample project

```
dotnet new avalonia.mvvm -o BindingPlayground
cd BindingPlayground
```

We'll expand `MainWindow.axaml` and `MainWindowViewModel.cs`.

3. Core bindings (OneWay, TwoWay, OneTime)

View model implementing `INotifyPropertyChanged`:

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

namespace BindingPlayground.ViewModels;

public class PersonViewModel : INotifyPropertyChanged
{
    private string _firstName = "Ada";
    private string _lastName = "Lovelace";
    private int _age = 36;

    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(nameof
    }

    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 `MainWindow.axaml` set the DataContext:

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

  <Design.DataContext>
    <vm:MainWindowViewModel />
  </Design.DataContext>


</Window>
```

`Design.DataContext` provides design-time data in the previewer.

4. Binding modes in action

```
<Grid ColumnDefinitions="*,*" RowDefinitions="Auto,*" Padding="16" RowSpacing="16" ColumnSpacing="24">
  <TextBlock Grid.ColumnSpan="2" Classes="h1" Text="Binding basics"/>

  <StackPanel Grid.Row="1" Spacing="8">
    <TextBox Watermark="First name" Text="{Binding Person.FirstName, Mode=TwoWay}"/>
    <TextBox Watermark="Last name"  Text="{Binding Person.LastName, Mode=TwoWay}"/>
    <NumericUpDown Minimum="0" Maximum="120" Value="{Binding Person.Age, Mode=TwoWay}"/>
  </StackPanel>

  <StackPanel Grid.Column="1" Grid.Row="1" Spacing="8">
    <TextBlock Text="Live view" FontWeight="SemiBold"/>
    <TextBlock Text="{Binding Person.FullName, Mode=OneWay}" FontSize="20"/>
    <TextBlock Text="{Binding Person.Age, Mode=OneWay}"/>
    <TextBlock Text="{Binding CreatedAt, Mode=OneTime, StringFormat='Created on {0:d}'}"/>
  </StackPanel>
</Grid>
```

`MainWindowViewModel` holds `Person` and other state:

```
using System;
using System.Collections.ObjectModel;

namespace BindingPlayground.ViewModels;

public class MainWindowViewModel : INotifyPropertyChanged
{
    public PersonViewModel Person { get; } = new();
    public DateTime CreatedAt { get; } = DateTime.Now;

    // Additional samples below
}
```

5. ElementName and RelativeSource

ElementName binding

```
<StackPanel Margin="0,24,0,0" Spacing="6">
  <Slider x:Name="VolumeSlider" Minimum="0" Maximum="100" Value="50"/>
  <ProgressBar Minimum="0" Maximum="100" Value="{Binding #VolumeSlider.Value}"/>
</StackPanel>
```

`#VolumeSlider` targets the element with `x:Name="VolumeSlider"`.

RelativeSource binding

Use `RelativeSource` to bind to ancestors:

```
<TextBlock Text="{Binding DataContext.Person.FullName, RelativeSource={RelativeSource AncestorType=Windo
```

This binds to the window's DataContext even if the local control has its own DataContext.

Relative source syntax also supports `Self` (`RelativeSource={RelativeSource Self}`) and `TemplatedParent` for control templates.

6. Compiled bindings

Compiled bindings (`CompiledBinding`) produce strongly-typed accessors with better performance. Require `x:DataType` or `CompiledBindings` namespace:

1. Add namespace to the root element:

```
xmlns:vm="clr-namespace:BindingPlayground.ViewModels"
```

2. Set `x:DataType` on a scope:

```
<StackPanel DataContext="{Binding Person}" x:DataType="vm:PersonViewModel">
  <TextBlock Text="{CompiledBinding FullName}"/>
  <TextBox Text="{CompiledBinding FirstName}"/>
</StackPanel>
```

If `x:DataType` is set, `CompiledBinding` uses compile-time checking and generates binding code. Source: `CompiledBindingExtension.cs`.

7. MultiBinding and PriorityBinding

MultiBinding

Combine multiple values into one target:

```
public sealed class NameAgeFormatter : IMultiValueConverter
{
    public object? Convert(IList<object?> values, Type targetType, object? parameter, CultureInfo cultu
    {
        var name = values[0] as string ?? "";
        var age = values[1] as int? ?? 0;
        return $"{name} ({age})";
    }

    public object? ConvertBack(IList<object?> values, Type targetType, object? parameter, CultureInfo cu
}
```

Register in resources:

```
<Window.Resources>
  <conv:NameAgeFormatter x:Key="NameAgeFormatter"/>
</Window.Resources>
```

Use it:

```
<TextBlock>
  <TextBlock.Text>
    <MultiBinding Converter="{StaticResource NameAgeFormatter}">
      <Binding Path="Person.FullName"/>
      <Binding Path="Person.Age"/>
    </MultiBinding>
  </TextBlock.Text>
</TextBlock>
```

PriorityBinding

Priority bindings try sources in order and use the first that yields a value:

```
<TextBlock>
  <TextBlock.Text>
    <PriorityBinding>
      <Binding Path="OverrideTitle"/>
      <Binding Path="Person.FullName"/>
      <Binding Path="Person.FirstName"/>
      <Binding Path="'Unknown user'"/>
    </PriorityBinding>
  </TextBlock.Text>
</TextBlock>
```

Source: `PriorityBinding.cs`.

8. Lists, selection, and templates

`MainWindowViewModel` exposes collections:

```
public ObservableCollection<PersonViewModel> People { get; } = new()
{
    new PersonViewModel { FirstName = "Ada", LastName = "Lovelace", Age = 36 },
    new PersonViewModel { FirstName = "Grace", LastName = "Hopper", Age = 45 },
    new PersonViewModel { FirstName = "Linus", LastName = "Torvalds", Age = 32 }
};

private PersonViewModel? _selectedPerson;
```

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

Template the list:

```
<ListBox Items="{Binding People}"
         SelectedItem="{Binding SelectedPerson, Mode=TwoWay}"
         Height="180">
  <ListBox.ItemTemplate>
    <DataTemplate x:DataType="vm:PersonViewModel">
      <StackPanel Orientation="Horizontal" Spacing="12">
        <TextBlock Text="{CompiledBinding FullName}" FontWeight="SemiBold"/>
        <TextBlock Text="{CompiledBinding Age}"/>
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Inside the details pane, bind to `SelectedPerson` safely using null-conditional binding (C#) or triggers. XAML automatically handles null (shows blank). Use `x:DataType` for compile-time checks.

`SelectionModel`

For advanced selection (multi-select, range), use `SelectionModel<T>` from `SelectionModel.cs`. Example:

```
public SelectionModel<PersonViewModel> PeopleSelection { get; } = new() { SelectionMode = SelectionMode
```

Bind it:

```
<ListBox Items="{Binding People}" Selection="{Binding PeopleSelection}"/>
```

9. Validation with `INotifyDataErrorInfo`

Implement `INotifyDataErrorInfo` for asynchronous validation.

```
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;

public class ValidatingPersonViewModel : PersonViewModel, INotifyDataErrorInfo
{
    private readonly Dictionary<string, List<string>> _errors = new();

    public bool HasErrors => _errors.Count > 0;

    public event EventHandler<DataErrorsChangedEventArgs>? ErrorsChanged;

    public IEnumerable GetErrors(string? propertyName)
        => propertyName is not null && _errors.TryGetValue(propertyName, out var errors) ? errors : Arra

    protected override void OnPropertyChanged(string? propertyName)
    {
        base.OnPropertyChanged(propertyName);
        Validate(propertyName);
    }
```

```
    private void Validate(string? propertyName)
    {
        if (propertyName is nameof(Age))
        {
            if (Age < 0 || Age > 120)
                AddError(propertyName, "Age must be between 0 and 120");
            else
                ClearErrors(propertyName);
        }
    }

    private void AddError(string propertyName, string error)
    {
        if (!_errors.TryGetValue(propertyName, out var list))
            _errors[propertyName] = list = new List<string>();

        if (!list.Contains(error))
        {
            list.Add(error);
            ErrorsChanged?.Invoke(this, new DataErrorsChangedEventArgs(propertyName));
        }
    }

    private void ClearErrors(string propertyName)
    {
        if (_errors.Remove(propertyName))
            ErrorsChanged?.Invoke(this, new DataErrorsChangedEventArgs(propertyName));
    }
}
```

Bind the validation feedback automatically:

```
<TextBox Text="{Binding ValidatingPerson.FirstName, Mode=TwoWay}"/>
<TextBox Text="{Binding ValidatingPerson.Age, Mode=TwoWay}"/>
<TextBlock Foreground="#B91C1C" Text="{Binding (Validation.Errors)[0].ErrorContent, RelativeSource={Rela
```

Avalonia surfaces validation errors via attached properties. For a full pattern see `Validation`.

10. Asynchronous bindings

Use `Task`-returning properties with `Binding` and `BindingPriority.AsyncLocalValue`. Example view model property:

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

public async Task LoadWeatherAsync()
{
    Weather = "Loading...";
    var result = await _weatherService.GetForecastAsync();
    Weather = result;
```

```
}
```

Bind with fallback until the value arrives:

```
<TextBlock Text="{Binding Weather, FallbackValue='Fetching forecast...'}"/>
```

You can also bind directly to `Task` results using `TaskObservableCollection` or reactive extensions (Chapter 17 covers background work).

11. Binding diagnostics

- **DevTools**: press F12 -> Diagnostics -> Binding Errors tab. Inspect live errors (missing properties, converters failing).
- **Binding logging**: enable via `BindingDiagnostics`.

```
using Avalonia.Diagnostics;

public override void OnFrameworkInitializationCompleted()
{
    BindingDiagnostics.Enable(
        log => Console.WriteLine(log.Message),
        new BindingDiagnosticOptions
        {
            Level = BindingDiagnosticLogLevel.Warning
        });

    base.OnFrameworkInitializationCompleted();
}
```

Source: `BindingDiagnostics.cs`.

Use `TraceBindingFailures` extension to log failures for specific bindings.

12. Practice exercises

1. **Compiled binding sweep**: add `x:DataType` to each data template and replace `Binding` with `CompiledBinding` where possible. Observe compile-time errors when property names are mistyped.
2. **MultiBinding formatting**: create a multi binding that formats `FirstName`, `LastName`, and `Age` into a sentence like "Ada Lovelace is 36 years old." Add a converter parameter for custom formats.
3. **Priority fallback**: allow a user-provided display name to override `FullName`, falling back to initials if names are empty.
4. **Validation UX**: display validation errors inline using `INotifyDataErrorInfo` and highlight inputs (`Style Selector="TextBox:invalid"`).
5. **Diagnostics drill**: intentionally break a binding (typo) and use DevTools and `BindingDiagnostics` to find it. Fix the binding and confirm logs clear.

Look under the hood (source bookmarks)

- Binding implementation: `Binding.cs`, `BindingExpression.cs`
- Compiled bindings: `CompiledBindingExtension.cs`
- Multi/Priority binding: `MultiBinding.cs`, `PriorityBinding.cs`
- Selection model: `SelectionModel.cs`
- Validation: `Validation.cs`
- Diagnostics: `BindingDiagnostics.cs`

Check yourself

- When would you choose `CompiledBinding` over `Binding`? What prerequisites does it have?

- How do `ElementName` and `RelativeSource` differ in resolving binding sources?
- What scenarios call for `MultiBinding` or `PriorityBinding`?
- How does `INotifyDataErrorInfo` surface validation errors to the UI? What attached properties expose them?
- Which tools can you use to debug binding failures during development?

What's next - Next: Chapter 9

9. Commands, events, and user input

Goal - Understand Avalonia's input system: routed events, commands, gesture recognizers, and keyboard navigation. - Choose between MVVM-friendly commands and low-level events effectively. - Wire keyboard shortcuts, pointer gestures, and access keys; capture pointer input for drag scenarios. - Implement asynchronous commands and recycle CanExecute logic with reactive or toolkit helpers. - Diagnose input issues with DevTools (Events view) and logging.

Why this matters - Robust input handling keeps UI responsive and testable. - Commands keep business logic in view models; events cover fine-grained gestures. - Knowing the pipeline (routed events -> gesture recognizers -> commands) helps debug "nothing happened" scenarios.

Prerequisites - Chapters 3-8 (layouts, controls, binding, theming). - Basic MVVM knowledge and an `INotifyPropertyChanged` view model.

1. Input building blocks

Avalonia input pieces live under: - Routed events infrastructure: `src/Avalonia.Interactivity` - Input elements & devices: `src/Avalonia.Base/Input` - Gesture recognizers (tap, pointer, scroll): `src/Avalonia.Base/Input/GestureRecognizers`

Event flow: 1. Input devices raise raw events (`PointerPressed`, `KeyDown`). 2. Routed events bubble/tunnel through the visual tree. 3. Gesture recognizers translate raw input into high-level events (TapGesture, DoubleTapped). 4. Commands may execute via Buttons, KeyBindings, Access keys.

2. Sample project setup

```
dotnet new avalonia.mvvm -o InputPlayground
cd InputPlayground
```

`MainWindowViewModel` exposes commands and state. Add `CommunityToolkit.Mvvm` or implement your own `AsyncRelayCommand` to simplify asynchronous logic. Example below uses a simple `RelayCommand` and `AsyncRelayCommand`.

```
using System;
using System.Threading.Tasks;
using System.Windows.Input;

namespace InputPlayground.ViewModels;

public sealed class MainWindowViewModel : ViewModelBase
{
    private string _status = "Ready";
    public string Status
    {
        get => _status;
        private set => SetProperty(ref _status, value);
    }

    private bool _hasChanges;
    public bool HasChanges
    {
        get => _hasChanges;
        set
        {
            if (SetProperty(ref _hasChanges, value))
            {
```

```
            SaveCommand.RaiseCanExecuteChanged();
        }
    }
}

public RelayCommand SaveCommand { get; }
public RelayCommand DeleteCommand { get; }
public AsyncRelayCommand RefreshCommand { get; }

public MainWindowViewModel()
{
    SaveCommand = new RelayCommand(_ => Save(), _ => HasChanges);
    DeleteCommand = new RelayCommand(item => Delete(item));
    RefreshCommand = new AsyncRelayCommand(RefreshAsync, () => !IsBusy);
}

private bool _isBusy;
public bool IsBusy
{
    get => _isBusy;
    private set
    {
        if (SetProperty(ref _isBusy, value))
        {
            RefreshCommand.RaiseCanExecuteChanged();
        }
    }
}

private void Save()
{
    Status = "Saved";
    HasChanges = false;
}

private void Delete(object? parameter)
{
    Status = parameter is string name ? $"Deleted {name}" : "Deleted item";
    HasChanges = true;
}

private async Task RefreshAsync()
{
    try
    {
        IsBusy = true;
        Status = "Refreshing...";
        await Task.Delay(1500);
        Status = "Data refreshed";
    }
    finally
    {
        IsBusy = false;
    }
```

```
    }
}
```

Supporting command classes (`RelayCommand`, `AsyncRelayCommand`) go in `Commands` folder. You may reuse the ones from CommunityToolkit.Mvvm or ReactiveUI.

3. Commands vs events cheat sheet

Use command when…	Use event when…
You expose an action (Save/Delete) from view model	You need pointer coordinates, delta, or low-level control
You want CanExecute/disable logic	You're implementing custom gestures/drag interactions
The action runs from buttons, menus, shortcuts	Work is purely visual or specific to a view
You plan to unit test the action	Data is transient or you need immediate UI feedback

Most real views mix both: commands for operations, events for gestures.

4. Binding commands in XAML

```
<StackPanel Spacing="12">
  <TextBox Watermark="Name" Text="{Binding SelectedName, Mode=TwoWay}"/>

  <StackPanel Orientation="Horizontal" Spacing="12">
    <Button Content="Save" Command="{Binding SaveCommand}"/>
    <Button Content="Refresh" Command="{Binding RefreshCommand}" IsEnabled="{Binding !IsBusy}"/>
    <Button Content="Delete" Command="{Binding DeleteCommand}"
            CommandParameter="{Binding SelectedName}"/>
  </StackPanel>

  <TextBlock Text="{Binding Status}"/>
</StackPanel>
```

Buttons disable automatically when `SaveCommand.CanExecute` returns false.

5. Keyboard shortcuts and access keys

KeyBinding / KeyGesture

```
<Window ...>
  <Window.InputBindings>
    <KeyBinding Gesture="Ctrl+S" Command="{Binding SaveCommand}"/>
    <KeyBinding Gesture="Ctrl+R" Command="{Binding RefreshCommand}"/>
    <KeyBinding Gesture="Ctrl+Delete" Command="{Binding DeleteCommand}" CommandParameter="{Binding Sele
  </Window.InputBindings>


</Window>
```

`KeyGesture` parsing is handled by `KeyGestureConverter`. For multiple gestures, add more `KeyBinding` entries.

Access keys (mnemonics)

Use `_` to define an access key in headers (e.g., `_Save`). Access keys work when Alt is pressed.

```
<Menu>
  <MenuItem Header="_File">
    <MenuItem Header="_Save" Command="{Binding SaveCommand}" InputGesture="Ctrl+S"/>
  </MenuItem>
</Menu>
```

Access keys are processed via `AccessKeyHandler` (`AccessKeyHandler.cs`).

6. Pointer gestures and recognizers

Avalonia includes built-in gesture recognizers. You can attach them via `GestureRecognizers` collection:

```
<Border Background="#1e293b" Padding="16">
  <Border.GestureRecognizers>
    <TapGestureRecognizer NumberOfTapsRequired="2" Command="{Binding DoubleTapCommand}" CommandParamete
    <ScrollGestureRecognizer CanHorizontallyScroll="True" CanVerticallyScroll="True"/>
  </Border.GestureRecognizers>

  <TextBlock Foreground="White" Text="Double-tap or scroll"/>
</Border>
```

Implementation: `TapGestureRecognizer.cs`.

For custom gestures (drag to reorder), handle `PointerPressed`, call `e.Pointer.Capture(control)` to capture input, and release on `PointerReleased`. Pointer capture ensures subsequent move/press events go to the capture target even if the pointer leaves its bounds.

```
private bool _isDragging;
private Point _dragStart;

private void Card_PointerPressed(object? sender, PointerPressedEventArgs e)
{
    _isDragging = true;
    _dragStart = e.GetPosition((Control)sender!);
    e.Pointer.Capture((IInputElement)sender!);
}

private void Card_PointerMoved(object? sender, PointerEventArgs e)
{
    if (_isDragging && sender is Control control)
    {
        var offset = e.GetPosition(control) - _dragStart;
        Canvas.SetLeft(control, offset.X);
        Canvas.SetTop(control, offset.Y);
    }
}

private void Card_PointerReleased(object? sender, PointerReleasedEventArgs e)
{
    _isDragging = false;
    e.Pointer.Capture(null);
}
```

See `PointerCapture` for details.

7. Text input pipeline (IME & composition)

Text entry flows through `TextInput` events. For IME (Asian languages), Avalonia raises `TextInput` with composition events. To hook into the pipeline, subscribe to `TextInput` or implement `ITextInputMethodClient` in custom controls. Source: `TextInputMethodClient.cs`.

```
<TextBox TextInput="TextBox_TextInput"/>
```

```
private void TextBox_TextInput(object? sender, TextInputEventArgs e)
{
    Debug.WriteLine($"TextInput: {e.Text}");
}
```

In most MVVM apps you rely on `TextBox` handling IME; implement this only when creating custom text editors.

8. Focus management and keyboard navigation

- Set `Focus()` to move focus programmatically.
- Use `Focusable="False"` on non-interactive elements.
- Control tab order with `TabIndex` (lower numbers focus first).
- Create focus scopes with `FocusManager` when using popups or overlays.

```
<StackPanel>
  <TextBox x:Name="First"/>
  <TextBox x:Name="Second"/>
  <Button Content="Focus second" Command="{Binding FocusSecondCommand}"/>
</StackPanel>
```

In the view model, expose a command that raises an event or use a focus service. For small cases, code-behind calling `Second.Focus()` is sufficient.

9. Routed commands and command routing

Avalonia supports routed commands similar to WPF. Define a `RoutedCommand` (`RoutedCommandLibrary.Save`, etc.) and attach handlers via `CommandBinding`.

```
<Window.CommandBindings>
  <CommandBinding Command="{x:Static commands:AppCommands.Save}" Executed="Save_Executed" CanExecute="Sa
</Window.CommandBindings>
```

```
private void Save_Executed(object? sender, ExecutedRoutedEventArgs e)
{
    if (DataContext is MainWindowViewModel vm)
        vm.SaveCommand.Execute(null);
}

private void Save_CanExecute(object? sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = (DataContext as MainWindowViewModel)?.SaveCommand.CanExecute(null) == true;
}
```

Routed commands bubble up the tree if not handled, allowing menu items and toolbars to share command logic.

Source: `RoutedCommand.cs`.

10. Asynchronous commands

Avoid blocking the UI thread. Use `AsyncRelayCommand` or custom `ICommand` that runs `Task`.

```
public sealed class AsyncRelayCommand : ICommand
{
    private readonly Func<Task> _execute;
    private readonly Func<bool>? _canExecute;
    private bool _isExecuting;

    public AsyncRelayCommand(Func<Task> execute, Func<bool>? canExecute = null)
    {
        _execute = execute;
        _canExecute = canExecute;
    }

    public bool CanExecute(object? parameter) => !_isExecuting && (_canExecute?.Invoke() ?? true);

    public async void Execute(object? parameter)
    {
        if (!CanExecute(parameter))
            return;

        try
        {
            _isExecuting = true;
            RaiseCanExecuteChanged();
            await _execute();
        }
        finally
        {
            _isExecuting = false;
            RaiseCanExecuteChanged();
        }
    }

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

11. Diagnostics: watch input live

DevTools (F12) -> **Events** tab let you monitor events (PointerPressed, KeyDown). Select an element, toggle events to watch.

Enable input logging:

```
AppBuilder.Configure<App>()
    .UsePlatformDetect()
    .LogToTrace(LogEventLevel.Debug, new[] { LogArea.Input })
    .StartWithClassicDesktopLifetime(args);
```

`LogArea.Input` (source: `LogArea.cs`) emits detailed input information.

12. Practice exercises

1. Add `Ctrl+Shift+S` for “Save As” (new command) and ensure it’s disabled when nothing is selected.

2. Implement a drag-to-reorder list using pointer capture. Use DevTools to verify pointer events.
3. Add a `TapGestureRecognizer` to a card view that toggles selection; log the event using `LogArea.Input`.
4. Implement asynchronous refresh with a cancellation token (Chapter 17) and tie the cancel command to the Esc key.
5. Use access keys (`_File`, `_Save`) and verify they work on Windows, macOS, and Linux keyboard layouts.

Look under the hood (source bookmarks)

- Commands: `ButtonBase.Command`, `MenuItem.Command`, `KeyBinding`
- Input elements & events: `InputElement.cs`, `PointerGestureRecognizer.cs`
- Access keys: `AccessText`, `AccessKeyHandler`
- Text input pipeline: `TextInputMethodClient.cs`

Check yourself

- What advantages do commands offer over events in MVVM architectures?
- How do you wire Ctrl+S and Ctrl+Shift+S to different commands?
- When do you need pointer capture?
- What pieces are involved in handling a DoubleTap gesture?
- Which tooling surfaces input events and binding? How would you enable verbose input logging?

What's next - Next: Chapter 10

10. Working with resources, images, and fonts

Goal - Master `avares://` URIs, `AssetLoader`, and resource dictionaries so you can bundle assets cleanly. - Display raster and vector images, control caching/interpolation, and brush surfaces with images. - Load custom fonts, configure `FontManagerOptions`, and support fallbacks. - Understand DPI scaling, bitmap interpolation, and how RenderOptions affects quality. - Hook resources into theming (DynamicResource) and diagnose missing assets quickly.

Why this matters - Assets and fonts give your app brand identity; doing it right avoids blurry visuals or missing resources. - Avalonia's resource system mirrors WPF/UWP but with cross-platform packaging; once you know the patterns, you can deploy confidently.

Prerequisites - You can edit `App.axaml`, views, and bind data (Ch. 3-9). - Familiarity with MVVM and theming (Ch. 7) helps when wiring assets dynamically.

1. `avares://` URIs and project structure

Assets live under your project (e.g., `Assets/Images`, `Assets/Fonts`). Include them as `AvaloniaResource` in the `.csproj`:

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

URI structure: `avares://<AssemblyName>/<RelativePath>`.

Example: `avares://InputPlayground/Assets/Images/logo.png`.

`avares://` references the compiled resource stream (not the file system). Use it consistently even within the same assembly to avoid issues with resource lookups.

2. Loading assets in XAML and code

XAML

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

Code using `AssetLoader`

```
using Avalonia.Platform;
using Avalonia.Media.Imaging;

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

`AssetLoader` lives in `Avalonia.Platform`.

Resource dictionaries

```
<ResourceDictionary xmlns="https://github.com/avaloniaui">
  <Bitmap x:Key="LogoBitmap">avares://AssetsDemo/Assets/Images/logo.png</Bitmap>
</ResourceDictionary>
```

You can then `StaticResource` expose `LogoBitmap`. Bitmaps created this way are cached.

3. Raster images and caching

`Image` control displays bitmaps. Performance tips: - Set `Stretch` to avoid unexpected distortions (Uniform, UniformToFill, Fill, None). - Use `RenderOptions.BitmapInterpolationMode` for scaling quality:

```
<Image Source="avares://AssetsDemo/Assets/Images/photo.jpg"
       Width="240" Height="160"
       RenderOptions.BitmapInterpolationMode="HighQuality"/>
```

Interpolation modes defined in `RenderOptions.cs`.

`Bitmap` supports caching and decoding. You can reuse preloaded bitmaps to avoid repeating disk IO.

4. ImageBrush and tiled backgrounds

`ImageBrush` paints surfaces:

```
<Ellipse Width="96" Height="96">
  <Ellipse.Fill>
    <ImageBrush Source="avares://AssetsDemo/Assets/Images/avatar.png"
                Stretch="UniformToFill" AlignmentX="Center" AlignmentY="Center"/>
  </Ellipse.Fill>
</Ellipse>
```

Tile backgrounds:

```
<Border Width="200" Height="120">
  <Border.Background>
    <ImageBrush Source="avares://AssetsDemo/Assets/Images/pattern.png"
                TileMode="Tile"
                Stretch="None"
                Transform="{ScaleTransform 0.5,0.5}"/>
  </Border.Background>
</Border>
```

`ImageBrush` documentation: `ImageBrush.cs`.

5. Vector graphics

Vector art scales with DPI, can adapt to theme colors, and stays crisp.

Inline geometry

```
<Path Data="M2 12 L9 19 L22 4"
      Stroke="{DynamicResource AccentBrush}"
      StrokeThickness="3"
      StrokeLineCap="Round" StrokeLineJoin="Round"/>
```

Store geometry in resources for reuse:

```
<ResourceDictionary xmlns="https://github.com/avaloniaui">
  <Geometry x:Key="IconCheck">M2 12 L9 19 L22 4</Geometry>
</ResourceDictionary>
```

Vector classes live under `Avalonia.Media`.

SVG support

Use the `Avalonia.Svg` community library or convert simple SVG paths manually. For production, bundling vector icons as XAML ensures theme compatibility.

6. Fonts and typography

Place fonts in `Assets/Fonts`. Register them in `App.axaml` via `Global::Avalonia` URI and specify the font face after `#`:

```
<Application.Resources>
  <FontFamily x:Key="HeadingFont">avares://AssetsDemo/Assets/Fonts/Inter.ttf#Inter</FontFamily>
</Application.Resources>
```

Use the font in styles:

```
<Application.Styles>
  <Style Selector="TextBlock.h1">
    <Setter Property="FontFamily" Value="{StaticResource HeadingFont}"/>
    <Setter Property="FontSize" Value="28"/>
    <Setter Property="FontWeight" Value="SemiBold"/>
  </Style>
</Application.Styles>
```

FontManager options

Configure global font settings in `AppBuilder`:

```
AppBuilder.Configure<App>()
    .UsePlatformDetect()
    .With(new FontManagerOptions
    {
        DefaultFamilyName = "avares://AssetsDemo/Assets/Fonts/Inter.ttf#Inter",
        FontFallbacks = new[] { new FontFallback { Family = "Segoe UI" }, new FontFallback { Family = "
    })
    .StartWithClassicDesktopLifetime(args);
```

`FontManagerOptions` lives in `FontManagerOptions.cs`.

Multi-weight fonts

If fonts include multiple weights, specify them with `FontWeight`. If you ship multiple font files (Regular, Bold), ensure the `#Family` name is consistent.

7. DPI scaling, caching, and performance

Avalonia measures layout in DIPs (1 DIP = 1/96 inch). High DPI monitors scale automatically.

- Prefer vector assets or high-resolution bitmaps.
- Use `RenderOptions.BitmapInterpolationMode="None"` for pixel art.
- For expensive bitmaps (charts) consider caching via `RenderTargetBitmap` or `WriteableBitmap`.

`RenderTargetBitmap` and `WriteableBitmap` under `Avalonia.Media.Imaging`.

8. Linking assets into themes

Bind brushes via `DynamicResource` so assets respond to theme changes:

```
<Application.Resources>
  <SolidColorBrush x:Key="AvatarFallbackBrush" Color="#1F2937"/>
</Application.Resources>

<Ellipse Fill="{DynamicResource AvatarFallbackBrush}"/>
```

Switch resources in theme dictionaries (Chapter 7). Example: lighten icons for Dark theme.

9. Diagnostics

- DevTools -> Resources shows resolved resources.
- Missing asset? Check the output logs (`RenderOptions` area) for "not found" messages.
- Use `AssetLoader.Exists(uri)` to verify at runtime:

```
if (!AssetLoader.Exists(uri))
    throw new FileNotFoundException($"Asset {uri} not found");
```

10. Sample "asset gallery"

```
<Grid ColumnDefinitions="Auto,24,Auto" RowDefinitions="Auto,12,Auto">

  <Image Width="160" Height="80" Stretch="Uniform"
         Source="avares://AssetsDemo/Assets/Images/logo.png"/>

  <Rectangle Grid.Column="1" Grid.RowSpan="3" Width="24"/>


  <Ellipse Grid.Column="2" Width="96" Height="96">
    <Ellipse.Fill>
      <ImageBrush Source="avares://AssetsDemo/Assets/Images/avatar.png" Stretch="UniformToFill"/>
    </Ellipse.Fill>
  </Ellipse>

  <Rectangle Grid.Row="1" Grid.ColumnSpan="3" Height="12"/>


  <Canvas Grid.Row="2" Grid.Column="0" Width="28" Height="28">
    <Path Data="M2 14 L10 22 L26 6"
          Stroke="{DynamicResource AccentBrush}"
          StrokeThickness="3" StrokeLineCap="Round" StrokeLineJoin="Round"/>
  </Canvas>

  <TextBlock Grid.Row="2" Grid.Column="2" Classes="h1" Text="Asset gallery"/>
</Grid>
```

11. Practice exercises

1. Package a second font family (italic) and create a style for quotes.
2. Load a user-selected image from disk using `OpenFileDialog` (Chapter 16) and display it via `Bitmap` and `ImageBrush`.
3. Add a vector icon that swaps color based on `ThemeVariant` (use `DynamicResource` to map theme brushes).
4. Experiment with `RenderOptions.BitmapInterpolationMode` to compare pixelated vs crisp scaling.
5. Create a sprite sheet (single PNG) and display multiple sub-regions using `ImageBrush.SourceRect`.

Look under the hood (source bookmarks)

- Asset loader and URIs: `AssetLoader.cs`
- Bitmap and imaging: `Bitmap.cs`
- Brushes: `ImageBrush.cs`
- Fonts & text formatting: `FontManager.cs`, `TextLayout.cs`
- Render options and DPI: `RenderOptions.cs`

Check yourself

- How do you ensure assets are embedded and addressable with `avares://` URIs?
- When would you use `Image` vs `ImageBrush` vs `Path`?
- What steps configure a custom font and fallback chain across platforms?
- How can `RenderOptions.BitmapInterpolationMode` improve image quality at different scales?
- Which tools help verify resources (DevTools, AssetLoader.Exists)?

What's next - Next: Chapter 11

11. MVVM in depth (with or without ReactiveUI)

Goal - Build production-ready MVVM layers using classic `INotifyPropertyChanged`, CommunityToolkit.Mvvm helpers, or ReactiveUI. - Map view models to views with data templates, view locator patterns, and dependency injection. - Compose complex state using property change notifications, derived properties, async commands, and navigation stacks. - Test view models and reactive flows confidently.

Why this matters - MVVM separates concerns so you can scale UI complexity, swap views, and run automated tests. - Avalonia supports multiple MVVM toolkits; understanding their trade-offs lets you choose the right fit per feature.

Prerequisites - Binding basics (Chapter 8) and commands/input (Chapter 9). - Familiarity with resource organization (Chapter 7) for styles and data templates.

1. MVVM recap

Layer	Role	Contains
Model	Core data/domain logic	POCOs, validation, persistence models
ViewModel	Bindable state, commands	`INotifyPropertyChanged`, `ICommand`, services
View	XAML + minimal code-behind	DataTemplates, layout, visuals

Focus on keeping business logic in view models/models; views remain thin.

2. Classic MVVM (manual or CommunityToolkit.Mvvm)

2.1 Property change base class

```
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? propertyName = null)
    {
        if (Equals(field, value))
            return false;

        field = value;
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
        return true;
    }
}
```

CommunityToolkit.Mvvm offers `ObservableObject`, `ObservableProperty` attribute, and `RelayCommand` out of the box. If you prefer built-in solutions, install `CommunityToolkit.Mvvm` and inherit from `ObservableObject` there.

2.2 Commands (`RelayCommand`)

```
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 ?? 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);
}
```

2.3 Sample: People view model

```
using System.Collections.ObjectModel;

public sealed class Person : ObservableObject
{
    private string _firstName;
    private string _lastName;

    public Person(string first, string last)
    {
        _firstName = first;
        _lastName = last;
    }

    public string FirstName
    {
        get => _firstName;
        set => SetProperty(ref _firstName, value);
    }

    public string LastName
    {
        get => _lastName;
        set => SetProperty(ref _lastName, value);
    }

    public override string ToString() => $"{FirstName} {LastName}";
}

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

    public ObservableCollection<Person> People { get; } = new();
    public RelayCommand AddCommand { get; }
    public RelayCommand RemoveCommand { get; }
```

```
    public PeopleViewModel(IPersonService personService)
    {
        _personService = personService;
        AddCommand = new RelayCommand(_ => AddPerson());
        RemoveCommand = new RelayCommand(_ => RemovePerson(), _ => Selected is not null);

        LoadInitialPeople();
    }

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

    private void LoadInitialPeople()
    {
        foreach (var person in _personService.GetInitialPeople())
            People.Add(person);
    }

    private void AddPerson()
    {
        var newPerson = _personService.CreateNewPerson();
        People.Add(newPerson);
        Selected = newPerson;
    }

    private void RemovePerson()
    {
        if (Selected is null)
            return;

        _personService.DeletePerson(Selected);
        People.Remove(Selected);
        Selected = null;
    }
}
```

`IPersonService` represents data access. Inject it via DI in `App.axaml.cs` (see Section 4).

2.4 Mapping view models to views via DataTemplates

```
<Application xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:views="clr-namespace:MyApp.Views"
             xmlns:viewmodels="clr-namespace:MyApp.ViewModels"
             x:Class="MyApp.App">
  <Application.DataTemplates>
```

```
    <DataTemplate DataType="{x:Type viewmodels:PeopleViewModel}">
      <views:PeopleView />
    </DataTemplate>
  </Application.DataTemplates>
</Application>
```

In `MainWindow.axaml`:

```
<ContentControl Content="{Binding CurrentViewModel}"/>
```

`CurrentViewModel` property determines which view to display. This is the ViewModel-first approach: DataTemplates map VM types to Views automatically.

2.5 Navigation service (classic MVVM)

```
public interface INavigationService
{
    void NavigateTo<TViewModel>() where TViewModel : class;
}

public sealed class NavigationService : ObservableObject, INavigationService
{
    private readonly IServiceProvider _services;
    private object? _currentViewModel;

    public object? CurrentViewModel
    {
        get => _currentViewModel;
        private set => SetProperty(ref _currentViewModel, value);
    }

    public NavigationService(IServiceProvider services)
    {
        _services = services;
    }

    public void NavigateTo<TViewModel>() where TViewModel : class
    {
        var vm = _services.GetRequiredService<TViewModel>();
        CurrentViewModel = vm;
    }
}
```

Register navigation service via dependency injection (next section). View models call `navigationService.NavigateTo<Peopl` to swap views.

3. Dependency injection and composition

Use your favorite DI container. Example with Microsoft.Extensions.DependencyInjection in `App.axaml.cs`:

```
using Microsoft.Extensions.DependencyInjection;

public partial class App : Application
{
    private IServiceProvider? _services;

    public override void OnFrameworkInitializationCompleted()
```

```
    {
        _services = ConfigureServices();

        if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
        {
            desktop.MainWindow = _services.GetRequiredService<MainWindow>();
        }

        base.OnFrameworkInitializationCompleted();
    }

    private static IServiceProvider ConfigureServices()
    {
        var services = new ServiceCollection();
        services.AddSingleton<MainWindow>();
        services.AddSingleton<INavigationService, NavigationService>();
        services.AddTransient<PeopleViewModel>();
        services.AddTransient<HomeViewModel>();
        services.AddSingleton<IPersonService, PersonService>();
        return services.BuildServiceProvider();
    }
}
```

Inject `INavigationService` into view models to drive navigation.

4. Testing classic MVVM view models

A unit test using xUnit:

```
[Fact]
public void RemovePerson_Disables_When_No_Selection()
{
    var service = Substitute.For<IPersonService>();
    var vm = new PeopleViewModel(service);

    vm.Selected = vm.People.First();
    Assert.True(vm.RemoveCommand.CanExecute(null));

    vm.Selected = null;
    Assert.False(vm.RemoveCommand.CanExecute(null));
}
```

Testing ensures command states and property changes behave correctly.

5. ReactiveUI approach

ReactiveUI provides `ReactiveObject`, `ReactiveCommand`, `WhenAnyValue`, and routing/interaction helpers. Source: `Avalonia.ReactiveUI`.

5.1 Reactive object and derived state

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

public sealed class PersonViewModelRx : ReactiveObject
{
```

```
    private string _firstName = "Ada";
    private string _lastName = "Lovelace";

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

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

    public string FullName => $"{FirstName} {LastName}";

    public PersonViewModelRx()
    {
        this.WhenAnyValue(x => x.FirstName, x => x.LastName)
            .Select(_ => Unit.Default)
            .Subscribe(_ => this.RaisePropertyChanged(nameof(FullName)));
    }
}
```

`WhenAnyValue` observes properties and recomputes derived values.

5.2 ReactiveCommand and async workflows

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

public sealed class PeopleViewModelRx : ReactiveObject
{
    private PersonViewModelRx? _selected;

    public ObservableCollection<PersonViewModelRx> People { get; } = new()
    {
        new PersonViewModelRx { FirstName = "Ada", LastName = "Lovelace" },
        new PersonViewModelRx { FirstName = "Grace", LastName = "Hopper" }
    };

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

    public ReactiveCommand<Unit, Unit> AddCommand { get; }
    public ReactiveCommand<PersonViewModelRx, Unit> RemoveCommand { get; }
    public ReactiveCommand<Unit, IReadOnlyList<PersonViewModelRx>> LoadCommand { get; }

    public PeopleViewModelRx(IPersonService service)
    {
        AddCommand = ReactiveCommand.Create(() =>
```

```
        {
            var vm = new PersonViewModelRx { FirstName = "New", LastName = "Person" };
            People.Add(vm);
            Selected = vm;
        });

        var canRemove = this.WhenAnyValue(x => x.Selected).Select(selected => selected is not null);
        RemoveCommand = ReactiveCommand.Create<PersonViewModelRx>(person => People.Remove(person), canR

        LoadCommand = ReactiveCommand.CreateFromTask(async () =>
        {
            var people = await service.FetchPeopleAsync();
            People.Clear();
            foreach (var p in people)
                People.Add(new PersonViewModelRx { FirstName = p.FirstName, LastName = p.LastName });
            return People.ToList();
        });

        LoadCommand.ThrownExceptions.Subscribe(ex => {/* handle errors */});
    }
}
```

`ReactiveCommand` exposes `IsExecuting`, `ThrownExceptions`, and ensures asynchronous flows stay on the UI thread.

5.3 `ReactiveUserControl` and activation

```
using ReactiveUI;
using System.Reactive.Disposables;

public partial class PeopleViewRx : ReactiveUserControl<PeopleViewModelRx>
{
    public PeopleViewRx()
    {
        InitializeComponent();

        this.WhenActivated(disposables =>
        {
            this.Bind(ViewModel, vm => vm.Selected, v => v.PersonList.SelectedItem)
                .DisposeWith(disposables);
            this.BindCommand(ViewModel, vm => vm.AddCommand, v => v.AddButton)
                .DisposeWith(disposables);
        });
    }
}
```

`WhenActivated` manages subscriptions. `Bind`/`BindCommand` reduce boilerplate. Source: `ReactiveUserControl.cs`.

5.4 View locator

ReactiveUI auto resolves views via naming conventions. Register `IViewLocator` in DI or implement your own to map view models to views. Avalonia.ReactiveUI includes `ViewLocator` class you can override.

```
public class AppViewLocator : IViewLocator
{
    public IViewFor? ResolveView<T>(T viewModel, string? contract = null) where T : class
```

```
    {
        var name = viewModel.GetType().FullName.Replace("ViewModel", "View");
        var type = Type.GetType(name ?? string.Empty);
        return type is null ? null : (IViewFor?)Activator.CreateInstance(type);
    }
}
```

Register it:

```
services.AddSingleton<IViewLocator, AppViewLocator>();
```

5.5 Routing and navigation

Routers manage stacks of `IRoutableViewModel` instances. Example shell view model shown earlier. Use `<rxui:RoutedViewHost Router="{Binding Router}"/>` to display the current view.

ReactiveUI navigation supports back/forward, parameter passing, and async transitions.

6. Interactions and dialogs

Use `Interaction<TInput,TOutput>` to request UI interactions from view models.

```
public Interaction<string, bool> ConfirmDelete { get; } = new();

DeleteCommand = ReactiveCommand.CreateFromTask(async () =>
{
    if (Selected is null)
        return;

    var ok = await ConfirmDelete.Handle($"Delete {Selected.FullName}?");
    if (ok)
        People.Remove(Selected);
});
```

In the view:

```
this.WhenActivated(d =>
{
    d(ViewModel!.ConfirmDelete.RegisterHandler(async ctx =>
    {
        var dialog = new ConfirmDialog(ctx.Input);
        var result = await dialog.ShowDialog<bool>(this);
        ctx.SetOutput(result);
    }));
});
```

7. Testing ReactiveUI view models

Use `TestScheduler` from `ReactiveUI.Testing` to control time:

```
[Test]
public void LoadCommand_PopulatesPeople()
{
    var scheduler = new TestScheduler();
    var service = Substitute.For<IPersonService>();
    service.FetchPeopleAsync().Returns(Task.FromResult(new[] { new Person("Alan", "Turing") }));

    var vm = new PeopleViewModelRx(service);
```

```
    vm.LoadCommand.Execute().Subscribe();

    scheduler.Start();

    Assert.Single(vm.People);
}
```

8. Choosing between toolkits

Toolkit	Pros	Cons
Manual / CommunityToolkit.Mvvm	Minimal dependencies, familiar, great for straightforward forms	More boilerplate for async flows, manual derived state
ReactiveUI	Powerful reactive composition, built-in routing/interaction, great for complex async state	Learning curve, more dependencies

Mixing is common: use classic MVVM for most pages; ReactiveUI for reactive-heavy screens.

9. Practice exercises

1. Convert the People example from classic to CommunityToolkit.Mvvm using `[ObservableProperty]` and `[RelayCommand]` attributes.
2. Add async loading with cancellation (Chapter 17) and unit-test cancellation for both MVVM styles.
3. Implement a view locator that resolves views via DI rather than naming convention.
4. Extend ReactiveUI routing with a modal dialog page and test navigation using `TestScheduler`.
5. Compare command implementations by profiling UI responsiveness when commands run long operations.

Look under the hood (source bookmarks)

- Avalonia + ReactiveUI integration: `Avalonia.ReactiveUI`
- Data templates & view mapping: `DataTemplate.cs`
- Reactive command implementation: `ReactiveCommand.cs`
- Interaction pattern: `Interaction.cs`

Check yourself

- What benefits does a view locator provide compared to manual view creation?
- How do `ReactiveCommand` and classic `RelayCommand` differ in async handling?
- Why is DI helpful when constructing view models? How would you register services in Avalonia?
- Which scenarios justify ReactiveUI's routing over simple `ContentControl` swaps?

What's next - Next: Chapter 12

12. Navigation, windows, and lifetimes

Goal - Understand how Avalonia lifetimes (desktop, single-view, browser) drive app startup and shutdown. - Manage windows: main, owned, modal, dialogs; persist placement; respect multiple screens. - Implement navigation patterns (content swapping, navigation services, transitions) that work across platforms. - Leverage `TopLevel` services (clipboard, storage, screens) from view models via abstractions.

Why this matters - Predictable navigation and windowing keep apps maintainable on desktop, mobile, and web. - Lifetimes differ per platform; knowing them prevents "works on Windows, fails on Android" surprises. - Services like file pickers or clipboard should be accessible through MVVM-friendly patterns.

Prerequisites - Chapter 4 (AppBuilder and lifetimes), Chapter 11 (MVVM patterns), Chapter 16 (storage) is referenced later.

1. Lifetimes recap

Lifetime	Use case	Entry method
`ClassicDesktopStyleApplicationLifetime`	Windows/macOS/Linux windowed apps	`StartWithClassicDesktopLifetime(args)`
`SingleViewApplicationLifetime`	Mobile (Android/iOS), embedded	`StartWithSingleViewLifetime(view)`
`BrowserSingleViewLifetime`	WebAssembly	`BrowserAppBuilder` setup

`App.OnFrameworkInitializationCompleted` should handle all lifetimes:

```
public override void OnFrameworkInitializationCompleted()
{
    var services = ConfigureServices();

    if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
    {
        var shell = services.GetRequiredService<MainWindow>();
        desktop.MainWindow = shell;

        // optional: intercept shutdown
        desktop.ShutdownMode = ShutdownMode.OnLastWindowClose;
    }
    else if (ApplicationLifetime is ISingleViewApplicationLifetime singleView)
    {
        singleView.MainView = services.GetRequiredService<ShellView>();
    }

    base.OnFrameworkInitializationCompleted();
}
```

When targeting browser, use `BrowserAppBuilder` with `SetupBrowserApp`.

2. Desktop windows in depth

2.1 Creating a main window with MVVM

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

```
        Opened += (_, _) => RestorePlacement();
        Closing += (_, e) => SavePlacement();
    }

    private const string PlacementKey = "MainWindowPlacement";

    private void RestorePlacement()
    {
        if (LocalSettings.TryReadWindowPlacement(PlacementKey, out var placement))
        {
            Position = placement.Position;
            Width = placement.Size.Width;
            Height = placement.Size.Height;
        }
    }

    private void SavePlacement()
    {
        LocalSettings.WriteWindowPlacement(PlacementKey, new WindowPlacement
        {
            Position = Position,
            Size = new Size(Width, Height)
        });
    }
}
```

`LocalSettings` is a simple persistence helper (file or user settings). Persisting placement keeps UX consistent.

2.2 Owned windows, modal vs modeless

```
public sealed class AboutWindow : Window
{
    public AboutWindow()
    {
        Title = "About";
        Width = 360;
        Height = 200;
        WindowStartupLocation = WindowStartupLocation.CenterOwner;
        Content = new TextBlock { Margin = new Thickness(16), Text = "My App v1.0" };
    }
}

// From main window or service
public Task ShowAboutDialogAsync(Window owner)
    => new AboutWindow { Owner = owner }.ShowDialog(owner);
```

Modeless window:

```
var tool = new ToolWindow { Owner = this };
tool.Show();
```

Always set `Owner` so modal blocks correctly and centering works.

2.3 Multiple screens & placement

Use `Screens` service from `TopLevel`:

```
var topLevel = TopLevel.GetTopLevel(this);
if (topLevel?.Screens is { } screens)
{
    var screen = screens.ScreenFromPoint(Position);
    var workingArea = screen.WorkingArea;
    Position = new PixelPoint(workingArea.X, workingArea.Y);
}
```

`Screens` live under `Avalonia.Controls/Screens.cs`.

2.4 Prevent closing with unsaved changes

```
Closing += async (sender, e) =>
{
    if (DataContext is ShellViewModel vm && vm.HasUnsavedChanges)
    {
        var confirm = await MessageBox.ShowAsync(this, "Unsaved changes", "Exit without saving?", Messa
        if (!confirm)
            e.Cancel = true;
    }
};
```

Implement `MessageBox` yourself or using Avalonia.MessageBox community package.

3. Navigation patterns

3.1 Content control navigation (shared for desktop & mobile)

```
public sealed class NavigationService : INavigationService
{
    private readonly IServiceProvider _services;
    private object? _current;

    public object? Current
    {
        get => _current;
        private set => _current = value;
    }

    public NavigationService(IServiceProvider services)
        => _services = services;

    public void NavigateTo<TViewModel>() where TViewModel : class
        => Current = _services.GetRequiredService<TViewModel>();
}
```

`ShellViewModel` coordinates navigation:

```
public sealed class ShellViewModel : ObservableObject
{
    private readonly INavigationService _navigationService;
    public object? Current => _navigationService.Current;

    public RelayCommand GoHome { get; }
    public RelayCommand GoSettings { get; }

    public ShellViewModel(INavigationService navigationService)
```

```
    {
        _navigationService = navigationService;
        GoHome = new RelayCommand(_ => _navigationService.NavigateTo<HomeViewModel>());
        GoSettings = new RelayCommand(_ => _navigationService.NavigateTo<SettingsViewModel>());
        _navigationService.NavigateTo<HomeViewModel>();
    }
}
```

Bind in view:

```
<DockPanel>
  <StackPanel DockPanel.Dock="Top" Orientation="Horizontal" Spacing="8">
    <Button Content="Home" Command="{Binding GoHome}"/>
    <Button Content="Settings" Command="{Binding GoSettings}"/>
  </StackPanel>
  <TransitioningContentControl Content="{Binding Current}">
    <TransitioningContentControl.Transitions>
      <PageSlide Transition="{Transitions:Slide FromRight}" Duration="0:0:0.2"/>
    </TransitioningContentControl.Transitions>
  </TransitioningContentControl>
</DockPanel>
```

`TransitioningContentControl` (from `Avalonia.Controls`) adds page transitions. Source: `TransitioningContentControl`

3.2 View mapping via DataTemplates

Register view-model-to-view templates (Chapter 11 showed details). Example snippet:

```
<Application.DataTemplates>
  <DataTemplate DataType="{x:Type vm:HomeViewModel}">
    <views:HomeView />
  </DataTemplate>
  <DataTemplate DataType="{x:Type vm:SettingsViewModel}">
    <views:SettingsView />
  </DataTemplate>
</Application.DataTemplates>
```

3.3 Dialog service abstraction

Expose a dialog API from view models without referencing `Window`:

```
public interface IDialogService
{
    Task<bool> ShowConfirmationAsync(string title, string message);
}

public sealed class DialogService : IDialogService
{
    private readonly Window _owner;
    public DialogService(Window owner) => _owner = owner;

    public async Task<bool> ShowConfirmationAsync(string title, string message)
    {
        var dialog = new ConfirmationWindow(title, message) { Owner = _owner };
        return await dialog.ShowDialog<bool>(_owner);
    }
}
```

Register a per-window dialog service in DI. For single-view scenarios, use `TopLevel.GetTopLevel(control)` to retrieve the root and use `StorageProvider` or custom dialogs.

4. Single-view navigation (mobile/web)

For `ISingleViewApplicationLifetime`, use a root `UserControl` (e.g., `ShellView`) with the same `TransitioningContentControl` pattern. Keep navigation inside that control.

```
<UserControl xmlns="https://github.com/avaloniaui" x:Class="MyApp.Views.ShellView">
  <TransitioningContentControl Content="{Binding Current}"/>
</UserControl>
```

From view models, use `INavigationService` as before; the lifetime determines whether a window or root view hosts the content.

5. TopLevel services: clipboard, storage, screens

`TopLevel.GetTopLevel(control)` returns the hosting top-level (Window or root). Useful for services.

5.1 Clipboard

```
var topLevel = TopLevel.GetTopLevel(control);
if (topLevel?.Clipboard is { } clipboard)
{
    await clipboard.SetTextAsync("Copied text");
}
```

Clipboard API defined in `IClipboard`.

5.2 Storage provider

Works in both desktop and single-view (browser has OS limitations):

```
var topLevel = TopLevel.GetTopLevel(control);
if (topLevel?.StorageProvider is { } sp)
{
    var file = (await sp.OpenFilePickerAsync(new FilePickerOpenOptions
    {
        AllowMultiple = false,
        FileTypeFilter = new[] { FilePickerFileTypes.TextPlain }
    })).FirstOrDefault();
}
```

5.3 Screens info

`topLevel!.Screens` provides monitor layout. Use for placing dialogs on active monitor or respecting working area.

6. Browser (WebAssembly) considerations

Use `BrowserAppBuilder` and `BrowserSingleViewLifetime`:

```
public static void Main(string[] args)
    => BuildAvaloniaApp().SetupBrowserApp("app");
```

Use `TopLevel.StorageProvider` for limited file access (via JavaScript APIs). Use JS interop for features missing from storage provider.

7. Practice exercises

1. Persist window placement, including maximized state, and restore on startup.
2. Implement a navigation history stack (back/forward) using a `Stack<object>` alongside `Current` binding.
3. Create a dialog service that exposes file open dialogs via `StorageProvider` and unit test the abstraction.
4. Detect the active screen and center modals on that screen, even when the main window spans monitors.
5. Implement transitions that differ per platform (e.g., slide on mobile, fade on desktop) by injecting transition providers.

Look under the hood (source bookmarks)

- Window management: `Window.cs`
- Lifetimes: `ClassicDesktopStyleApplicationLifetime.cs`, `SingleViewApplicationLifetime.cs`
- TopLevel services: `TopLevel.cs`
- Transitioning content: `TransitioningContentControl.cs`

Check yourself

- How does `ClassicDesktopStyleApplicationLifetime` differ from `SingleViewApplicationLifetime` when showing windows?
- When should you use `Show` vs `ShowDialog`? Why set `Owner`?
- How do `TransitioningContentControl` and DataTemplates enable platform-neutral navigation?
- Which `TopLevel` service would you use to access the clipboard or file picker from a view model?

What's next - Next: Chapter 13

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

Goal - Build desktop-friendly menus (in-window and native), wire accelerators, and update menu state dynamically. - Provide dialogs through MVVM-friendly services (file pickers, confirmation dialogs, message boxes) that run on desktop and single-view lifetimes. - Integrate system tray icons/notifications responsibly and respect platform nuances (Windows, macOS, Linux). - Access `TopLevel` services (IStorageProvider, Clipboard, Screens) through abstractions.

Why this matters - Menus/tray icons are expected on desktop apps; implementing them cleanly keeps UI testable and idiomatic. - Dialog flows should not couple view models to windows; service abstractions allow unit testing and platform reuse. - Platform-specific APIs (macOS menu bar, Windows tray icons) need awareness to avoid glitches.

Prerequisites - Chapter 9 (commands/input), Chapter 11 (MVVM patterns), Chapter 12 (lifetimes/navigation).

1. Menus and accelerators

1.1 In-window menu (cross-platform)

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="MyApp.MainWindow"
        Title="My App" Width="900" Height="600">
  <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"/>
        <MenuItem Header="_Save" Command="{Binding SaveCommand}" InputGestureText="Ctrl+S"/>
        <Separator/>
        <MenuItem Header="E_xit" Command="{Binding ExitCommand}"/>
      </MenuItem>
      <MenuItem Header="_Edit">
        <MenuItem Header="_Undo" Command="{Binding UndoCommand}" InputGestureText="Ctrl+Z"/>
        <MenuItem Header="_Redo" Command="{Binding RedoCommand}" InputGestureText="Ctrl+Y"/>
      </MenuItem>
      <MenuItem Header="_Help">
        <MenuItem Header="_About" Command="{Binding ShowAboutCommand}"/>
      </MenuItem>
    </Menu>

    <ContentControl Content="{Binding Current}"/>
  </DockPanel>
</Window>
```

Add `KeyBinding` entries (Chapter 9) so shortcuts invoke commands everywhere:

```
<Window.InputBindings>
  <KeyBinding Gesture="Ctrl+N" Command="{Binding NewCommand}"/>
  <KeyBinding Gesture="Ctrl+O" Command="{Binding OpenCommand}"/>
</Window.InputBindings>
```

1.2 Native menu bar (macOS/global menu)

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

```
        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>
  </DockPanel>
</Window>
```

Use NativeMenuBar on platforms that support global menus (macOS). In-window Menu remains for Windows/Linux.

1.3 Dynamic menu updates

Bag commands that toggle state and call `RaiseCanExecuteChanged()`. Example: enabling “Save” only when there are changes.

```
public bool CanSave => HasChanges;
public RelayCommand SaveCommand { get; }

private void OnDocumentChanged()
{
    HasChanges = true;
    SaveCommand.RaiseCanExecuteChanged();
}
```

Menu item automatically disables when `CanExecute` returns false.

2. Context menus and flyouts

2.1 Context menu per control

```
<ListBox Items="{Binding Documents}" SelectedItem="{Binding SelectedDocument}">
  <ListBox.ItemContainerTheme>
    <ControlTheme TargetType="ListBoxItem">
      <Setter Property="ContextMenu">
        <ContextMenu>
          <MenuItem Header="Rename" Command="{Binding DataContext.RenameCommand, RelativeSource={Relati
          <MenuItem Header="Delete" Command="{Binding DataContext.DeleteCommand, RelativeSource={Relati
        </ContextMenu>
      </Setter>
    </ControlTheme>
  </ListBox.ItemContainerTheme>
</ListBox>
```

- `RelativeSource AncestorType=ListBox` lets the item access commands on the parent view model.

2.2 Flyout for custom UI

```
<Button Content="More">
  <Button.Flyout>
    <Flyout>
      <StackPanel Margin="8" Spacing="8">
        <TextBlock Text="Quick actions"/>
        <ToggleSwitch Content="Enable feature" IsChecked="{Binding IsFeatureEnabled}"/>
        <Button Content="Open settings" Command="{Binding OpenSettingsCommand}"/>
      </StackPanel>
    </Flyout>
  </Button.Flyout>
</Button>
```

Flyouts support arbitrary content; use `MenuItem` when you only need command lists.

3. Dialog patterns

3.1 ViewModel-friendly dialog service

```
public interface IDialogService
{
    Task<bool> ShowConfirmationAsync(string title, string message);
    Task<string?> ShowOpenFilePickerAsync();
}

public sealed class DialogService : IDialogService
{
    private readonly Window _owner;

    public DialogService(Window owner) => _owner = owner;

    public async Task<bool> ShowConfirmationAsync(string title, string message)
    {
        var dialog = new ConfirmationDialog(title, message) { Owner = _owner };
        return await dialog.ShowDialog<bool>(_owner);
    }

    public async Task<string?> ShowOpenFilePickerAsync()
    {
        var ofd = new OpenFileDialog
        {
            AllowMultiple = false,
            Filters = { new FileDialogFilter { Name = "Documents", Extensions = { "txt", "md" } } }
        };
        var files = await ofd.ShowAsync(_owner);
        return files?.FirstOrDefault();
    }
}
```

Register per window in DI:

```
services.AddScoped<IDialogService>(sp =>
{
    var window = sp.GetRequiredService<MainWindow>();
    return new DialogService(window);
```

```
});
```

Provide `IDialogService` to `ShellViewModel`. For single-view apps, implement the same interface using `TopLevel.GetTopLevel(view)` to access storage provider.

3.2 Storage provider (cross-platform)

```
public sealed class CrossPlatformDialogService : IDialogService
{
    private readonly TopLevel _topLevel;

    public CrossPlatformDialogService(TopLevel topLevel) => _topLevel = topLevel;

    public Task<bool> ShowConfirmationAsync(string title, string message)
        => MessageBox.ShowAsync(_topLevel, title, message, MessageBoxButtons.YesNo);

    public async Task<string?> ShowOpenFilePickerAsync()
    {
        if (_topLevel.StorageProvider is null)
            return null;

        var result = await _topLevel.StorageProvider.OpenFilePickerAsync(new FilePickerOpenOptions
        {
            AllowMultiple = false,
            FileTypeFilter = new[] { FilePickerFileTypes.TextPlain }
        });
        var file = result.FirstOrDefault();
        return file is null ? null : file.Path.LocalPath;
    }
}
```

4. Message boxes and notifications

Avalonia doesn't ship a default message box, but community packages (`Avalonia.MessageBox`) or custom windows work. A simple custom message box window:

```
public sealed class MessageBoxWindow : Window
{
    public MessageBoxWindow(string title, string message)
    {
        Title = title;
        WindowStartupLocation = WindowStartupLocation.CenterOwner;
        var ok = new Button { Content = "OK", IsDefault = true };
        ok.Click += (_, __) => Close(true);
        Content = new StackPanel
        {
            Margin = new Thickness(16),
            Spacing = 12,
            Children = { new TextBlock { Text = message }, ok }
        };
    }
}
```

5. Tray icons and notifications

```
public sealed class TrayIconService : IDisposable
{
    private readonly IClassicDesktopStyleApplicationLifetime _lifetime;
    private readonly TrayIcon _trayIcon;

    public TrayIconService(IClassicDesktopStyleApplicationLifetime lifetime)
    {
        _lifetime = lifetime;

        var showItem = new NativeMenuItem("Show");
        showItem.Click += (_, _) => _lifetime.MainWindow?.Show();

        var exitItem = new NativeMenuItem("Exit");
        exitItem.Click += (_, _) => _lifetime.Shutdown();

        _trayIcon = new TrayIcon
        {
            ToolTipText = "My App",
            Icon = new WindowIcon("avares://MyApp/Assets/AppIcon.ico"),
            Menu = new NativeMenu { Items = { showItem, exitItem } }
        };
        _trayIcon.Show();
    }

    public void Dispose() => _trayIcon.Dispose();
}
```

Register the service in `App.OnFrameworkInitializationCompleted` when using desktop lifetime; dispose on exit. Tray icons are not supported on mobile/web.

Notifications

Avalonia's `Avalonia.Controls.Notifications` package provides in-app notifications or Windows toast integrations. Example in-app notification manager:

```
using Avalonia.Controls.Notifications;

var manager = new WindowNotificationManager(MainWindow)
{
    Position = NotificationPosition.TopRight,
    MaxItems = 3
};

manager.Show(new Notification("Saved", "Document saved successfully", NotificationType.Success));
```

6. Accessing system services via `TopLevel`

6.1 Clipboard service

```
public interface IClipboardService
{
    Task SetTextAsync(string text);
    Task<string?> GetTextAsync();
}
```

```
public sealed class ClipboardService : IClipboardService
{
    private readonly TopLevel _topLevel;
    public ClipboardService(TopLevel topLevel) => _topLevel = topLevel;

    public Task SetTextAsync(string text) => _topLevel.Clipboard?.SetTextAsync(text) ?? Task.CompletedTa
    public Task<string?> GetTextAsync() => _topLevel.Clipboard?.GetTextAsync() ?? Task.FromResult<strin
}
```

Include this service in DI so view models request clipboard operations without referencing controls.

6.2 Drag and drop / system features

Drag-and-drop uses `DragDrop` APIs (Chapter 16). System features like power notifications or window effects are platform-specific–wrap them in services like the dialog example.

7. Platform guidance

- **macOS**: use `NativeMenuBar`, ensure About/Quit live under the first menu. Tray icons appear in the status bar; `WindowIcon` must be sized to `NSImage` standards.
- **Windows**: `Menu` inside window is standard. Tray icons appear in the notification area; wrap `TrayIcon` show/hide in a service.
- **Linux**: Menus vary per environment; in-window `Menu` works everywhere. Tray icons depend on desktop environment (GNOME may require extensions).
- **Mobile/Web**: skip menus/tray icons; use flyouts, toolbars, and bottom sheets.

8. Practice exercises

1. Add menu commands that update their text or visibility when application state changes, verifying `PropertyChanged` triggers menu updates.
2. Implement a dialog service interface that supports open/save dialogs via `IStorageProvider` and falls back to message boxes when unsupported.
3. Add context menus to list items with enable/disable states reflecting `CanExecute`.
4. Create a tray icon that toggles a “compact mode”, minimizing the window when closing and restoring on double-click.
5. Build a notification manager that displays toast-like overlays using `WindowNotificationManager` and ensure they hide on navigation.

Look under the hood (source bookmarks)

- Menus/Native menus: `Menu.cs`, `NativeMenuBar.cs`
- Context menu & flyouts: `ContextMenu.cs`, `Flyout.cs`
- Tray icons: `TrayIcon.cs`
- Notifications: `WindowNotificationManager.cs`
- Storage provider: `IStorageProvider`

Check yourself

- When do you prefer `NativeMenuBar` vs in-window `Menu`? How do you attach shortcuts to the same command?
- How do you expose dialogs to view models without referencing `Window`?
- What should you consider before adding a tray icon (platform support, lifecycle)?
- Which `TopLevel` services help with clipboard or file picking?

What’s next - Next: Chapter 14

14. Lists, virtualization, and performance

Goal - Choose the right list control (ItemsControl, ListBox, DataGrid, TreeView, ItemsRepeater) for large data sets. - Understand virtualization internals (`VirtualizingStackPanel`, `ItemsPresenter`, recycling) and how to tune them. - Implement incremental loading, selection patterns, and grouped/hierarchical lists efficiently. - Diagnose list performance with DevTools, logging, and profiling.

Why this matters - Lists power dashboards, log viewers, chat apps, tables, and trees. Poorly configured lists freeze apps. - Virtualization keeps memory and CPU usage manageable even with hundreds of thousands of rows.

Prerequisites - Binding/commands (Chapters 8-9), MVVM patterns (Chapter 11).

1. Choosing the right control

Control	When to use	Notes
`ItemsControl`	Simple, read-only lists with custom layout	No selection built in; good for dashboards/badges
`ListBox`	Lists with selection, keyboard navigation	Virtualizes by default when using `VirtualizingStackPanel`
`ItemsRepeater`	High-performance custom layouts, virtualization	Requires manual layout definition; power users only
`DataGrid`	Tabular data with columns, sorting, editing	Virtualizes rows; define columns explicitly
`TreeView`	Hierarchical data	Virtualizes expanded nodes; heavy trees need cautious design

2. Virtualization internals

- `VirtualizingStackPanel` implements `ILogicalScrollable`. It creates visuals only for items near the viewport.
- `ItemsPresenter` hosts the items panel (`ItemsPanelTemplate`). Changing the panel can enable/disable virtualization.
- `ScrollViewer` orchestrates scroll offsets; virtualization works when `ScrollViewer` contains the items host directly.

Ensure virtualization stays active: - Use `ItemsPanelTemplate` with `VirtualizingStackPanel` (or custom panel implementing `IVirtualizingPanel` soon). - Avoid wrapping the items panel in another scroll viewer. - Keep item visuals lightweight; container recycling reuses them to avoid allocations.

3. ListBox with virtualization

```
<ListBox Items="{Binding People}"
         SelectedItem="{Binding Selected}" Height="360">
  <ListBox.ItemsPanel>
    <ItemsPanelTemplate>
      <VirtualizingStackPanel IsVirtualizing="True"
                              Orientation="Vertical"
                              AreHorizontalSnapPointsRegular="True"/>
    </ItemsPanelTemplate>
  </ListBox.ItemsPanel>
  <ListBox.ItemTemplate>
    <DataTemplate x:DataType="vm:PersonViewModel">
      <Grid ColumnDefinitions="Auto,*,Auto" Height="40" Margin="2">
```

```
        <TextBlock Grid.Column="0" Text="{CompiledBinding Id}" Width="48" HorizontalAlignment="Right"/>
        <StackPanel Grid.Column="1" Orientation="Vertical" Spacing="2" Margin="8,0">
          <TextBlock Text="{CompiledBinding FullName}" FontWeight="SemiBold"/>
          <TextBlock Text="{CompiledBinding Email}" FontSize="12" Foreground="#6B7280"/>
        </StackPanel>
        <Button Grid.Column="2"
                Content="Open"
                Command="{Binding DataContext.OpenCommand, RelativeSource={RelativeSource AncestorType=
                CommandParameter="{Binding}"/>
      </Grid>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Tips: - Fixed item height (40) helps virtualization predict layout quickly. - Use `CompiledBinding` to avoid runtime reflection overhead.

4. ItemsRepeater for custom layouts

`ItemsRepeater` (namespace `Avalonia.Controls`) allows custom layout algorithms.

```
<ItemsRepeater Items="{Binding Photos}" xmlns:controls="clr-namespace:Avalonia.Controls;assembly=Avalon
  <ItemsRepeater.Layout>
    <controls:UniformGridLayout Orientation="Vertical" MinItemWidth="200"/>
  </ItemsRepeater.Layout>
  <ItemsRepeater.ItemTemplate>
    <DataTemplate x:DataType="vm:PhotoViewModel">
      <Border Margin="6" Padding="6" Background="#111827" CornerRadius="6">
        <StackPanel>
          <Image Source="{CompiledBinding ThumbnailSource}" Width="188" Height="120" Stretch="UniformTo
          <TextBlock Text="{CompiledBinding Title}" Margin="0,6,0,0"/>
        </StackPanel>
      </Border>
    </DataTemplate>
  </ItemsRepeater.ItemTemplate>
</ItemsRepeater>
```

`ItemsRepeater` virtualization is handled by the layout. Use `UniformGridLayout`, `StackLayout`, or custom layout.

5. SelectionModel for advanced scenarios

`SelectionModel<T>` enables multi-select, anchor selection, and virtualization-friendly selection.

```
public SelectionModel<PersonViewModel> PeopleSelection { get; } = new() { SelectionMode = SelectionMode
```

Bind to `ListBox`:

```
<ListBox Items="{Binding People}" Selection="{Binding PeopleSelection}" Height="360"/>
```

`SelectionModel` lives in `Avalonia.Controls/Selection/SelectionModel.cs`.

6. Incremental loading pattern

View model

```
public sealed class LogViewModel : ObservableObject
{
```

```
    private readonly ILogService _service;
    private readonly ObservableCollection<LogEntryViewModel> _entries = new();
    private bool _isLoading;

    public ReadOnlyObservableCollection<LogEntryViewModel> Entries { get; }
    public RelayCommand LoadMoreCommand { get; }

    private int _pageIndex;
    private const int PageSize = 500;

    public LogViewModel(ILogService service)
    {
        _service = service;
        Entries = new ReadOnlyObservableCollection<LogEntryViewModel>(_entries);
        LoadMoreCommand = new RelayCommand(async () => await LoadMoreAsync(), () => !_isLoading);
        _ = LoadMoreAsync();
    }

    private async Task LoadMoreAsync()
    {
        if (_isLoading) return;
        _isLoading = true;
        LoadMoreCommand.RaiseCanExecuteChanged();
        try
        {
            var batch = await _service.GetEntriesAsync(_pageIndex, PageSize);
            foreach (var entry in batch)
                _entries.Add(new LogEntryViewModel(entry));

            _pageIndex++;
            HasMore = batch.Count == PageSize;
        }
        finally
        {
            _isLoading = false;
            LoadMoreCommand.RaiseCanExecuteChanged();
        }
    }

    public bool HasMore { get; private set; } = true;
}
```

XAML

```
<ListBox Items="{Binding Entries}" Height="480" ScrollViewer.ScrollChanged="ListBox_ScrollChanged">
  <ListBox.ItemTemplate>
    <DataTemplate x:DataType="vm:LogEntryViewModel">
      <TextBlock Text="{CompiledBinding Message}" FontFamily="Consolas"/>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

In code-behind, trigger `LoadMoreCommand` near bottom:

```
private void ListBox_ScrollChanged(object? sender, ScrollChangedEventArgs e)
```

```
{
    if (DataContext is LogViewModel vm && vm.HasMore)
    {
        var scroll = e.Source as ScrollViewer;
        if (scroll is not null && scroll.Offset.Y + scroll.Viewport.Height >= scroll.Extent.Height - 20
        {
            if (vm.LoadMoreCommand.CanExecute(null))
                vm.LoadMoreCommand.Execute(null);
        }
    }
}
```

7. DataGrid performance

- Set `EnableRowVirtualization="True"` (default) and `EnableColumnVirtualization="True"` if width changes are minimal.
- Define columns manually:

```
<DataGrid Items="{Binding People}" AutoGenerateColumns="False" IsReadOnly="True">
  <DataGrid.Columns>
    <DataGridTextColumn Header="Name" Binding="{Binding FullName}" Width="*"/>
    <DataGridTextColumn Header="Email" Binding="{Binding Email}" Width="2*"/>
    <DataGridTextColumn Header="Status" Binding="{Binding Status}" Width="Auto"/>
  </DataGrid.Columns>
</DataGrid>
```

- Use `DataGridTemplateColumn` sparingly; prefer text columns for speed.
- For huge datasets, consider server-side paging and virtualization; DataGrid can handle ~100k rows efficiently with virtualization enabled.

8. Grouping and hierarchical data

Grouping with `CollectionView`

```
var collectionView = new CollectionViewSource(People)
{
    GroupDescriptions = { new PropertyGroupDescription("Department") }
}.View;
```

Bind to `ItemsControl` with `GroupStyle`. Group headers should be minimal to keep virtualization efficient.

TreeView virtualization

- Virtualizes expanded nodes only.
- Keep templates thin; consider lazy loading children.

```
<TreeView Items="{Binding Departments}" SelectedItems="{Binding SelectedDepartments}">
  <TreeView.ItemTemplate>
    <TreeDataTemplate ItemsSource="{Binding Teams}" x:DataType="vm:DepartmentViewModel">
      <TextBlock Text="{CompiledBinding Name}" FontWeight="SemiBold"/>
      <TreeDataTemplate.ItemTemplate>
        <DataTemplate x:DataType="vm:TeamViewModel">
          <TextBlock Text="{CompiledBinding Name}" Margin="24,0,0,0"/>
        </DataTemplate>
      </TreeDataTemplate.ItemTemplate>
    </TreeDataTemplate>
```

```
  </TreeView.ItemTemplate>
</TreeView>
```

Defer loading large subtrees until expanded (bind to command that fetches children on demand).

9. Diagnostics and profiling

- DevTools -> **Visual Tree**: see realized items count.
- DevTools -> **Events**: watch scroll events and virtualization events.
- Enable layout/render logs:

```
AppBuilder.Configure<App>()
    .UsePlatformDetect()
    .LogToTrace(LogEventLevel.Debug, new[] { LogArea.Layout, LogArea.Rendering })
    .StartWithClassicDesktopLifetime(args);
```

- Use .NET memory profilers or `dotnet-counters` to monitor GC activity while scrolling.

10. Practice exercises

1. Create a log viewer with `ListBox + VirtualizingStackPanel` that streams 100k log lines; ensure smooth scroll and provide "Pause autoscroll".
2. Replace an `ItemsControl` dashboard with `ItemsRepeater` using `UniformGridLayout` for better virtualization.
3. Implement `SelectionModel` for multi-select email list and bind to checkboxes inside the template.
4. Add grouping to a `CollectionView`, showing group headers while keeping virtualization intact.
5. Profile a virtualized vs non-virtualized DataGrid with 200k rows and report memory usage.

Look under the hood (source bookmarks)

- Virtualizing panels: `VirtualizingStackPanel.cs`
- Selection model: `SelectionModel.cs`
- ItemsRepeater layouts: `UniformGridLayout.cs`
- DataGrid internals: `Avalonia.Controls.DataGrid`
- Tree virtualization: `TreeView.cs`

Check yourself

- Which panels support virtualization and how do you enable them in ListBox/ItemsControl?
- How does `SelectionModel` improve multi-select scenarios compared to `SelectedItems`?
- What strategies keep DataGrid fast with huge datasets?
- How can you detect when virtualization is broken?

What's next - Next: Chapter 15

15. Accessibility and internationalization

Goal - Deliver interfaces that are usable with keyboard, screen readers, and high-contrast themes. - Localize content, formats, and layout direction for multiple cultures. - Implement automation metadata (AutomationProperties, custom peers) and test accessibility.

Why this matters - Accessibility ensures compliance (WCAG/ADA) and a better experience for keyboard and assistive technology users. - Internationalization widens reach and avoids culture-specific bugs.

Prerequisites - Keyboard/commands (Chapter 9), resources (Chapter 10), MVVM (Chapter 11), navigation (Chapter 12).

1. Keyboard accessibility

1.1 Focus order and tab stops

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

  <TextBlock Text="_Password" RecognizesAccessKey="True"/>
  <PasswordBox x:Name="Password" TabIndex="1"/>

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

  <StackPanel Orientation="Horizontal" Spacing="8">
    <Button TabIndex="3">
      <AccessText Text="_Sign in"/>
    </Button>
    <Button TabIndex="4">
      <AccessText Text="_Cancel"/>
    </Button>
  </StackPanel>
</StackPanel>
```

- `KeyboardNavigation.TabNavigation="Cycle"` wraps focus within container.
- Use `IsTabStop="False"` or `Focusable="False"` for decorative elements.
- Access keys (underscore) require `AccessText` or `RecognizesAccessKey="True"`.

1.2 Keyboard navigation helpers

`KeyboardNavigation` class (source: `KeyboardNavigation.cs`) supports: - `DirectionalNavigation="Cycle"` for arrow-key traversal (menus, grids). - `TabNavigation` modes: `Continue`, `Once`, `Local`, `Cycle`, `None`.

2. Screen reader semantics

2.1 AutomationProperties essentials

```
<StackPanel Spacing="10">
  <TextBlock x:Name="EmailLabel" Text="Email"/>
  <TextBox Text="{Binding Email}" AutomationProperties.LabeledBy="{Binding #EmailLabel}"/>

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

Properties: - `AutomationProperties.Name`: accessible label if no visible label exists. - `AutomationProperties.HelpText`: extra instructions. - `AutomationProperties.AutomationId`: stable ID for UI tests. - `AutomationProperties.ControlType`: override role in rare cases. - `AutomationProperties.LabeledBy`: link to label element.

2.2 Announcing updates

For live regions (status bars, chat messages):

```
<TextBlock AutomationProperties.LiveSetting="Polite" Text="{Binding Status}"/>
```

`Polite` vs `Assertive` determines urgency.

2.3 Custom automation peers

When creating custom controls, override `OnCreateAutomationPeer` (source: `ControlAutomationPeer.cs`):

```
public class ProgressBadge : TemplatedControl
{
    protected override AutomationPeer? OnCreateAutomationPeer()
        => new ProgressBadgeAutomationPeer(this);
}

public sealed class ProgressBadgeAutomationPeer : ControlAutomationPeer
{
    public ProgressBadgeAutomationPeer(ProgressBadge owner) : base(owner) { }

    protected override string? GetNameCore()
        => (Owner as ProgressBadge)?.Text;

    protected override AutomationControlType GetAutomationControlTypeCore()
        => AutomationControlType.Text;
}
```

Register peers for custom controls to describe their role/names to screen readers.

3. High contrast & color considerations

- Provide sufficient contrast (WCAG 2.1 suggests 4.5:1 for text).
- Use theme resources instead of hard-coded colors. For high contrast, include variant dictionaries:

```
<ResourceDictionary ThemeVariant="HighContrast">
  <SolidColorBrush x:Key="AccentBrush" Color="#00FF00"/>
</ResourceDictionary>
```

Test high contrast by toggling `RequestedThemeVariant` (Chapter 7) and using OS settings.

4. Internationalization (i18n)

4.1 Resource management with RESX

Create `Resources.resx` (default) and `Resources.{culture}.resx`. Example localizer:

```
public sealed class Loc : INotifyPropertyChanged
{
    private CultureInfo _culture = CultureInfo.CurrentUICulture;
    private readonly ResourceManager _resources = Resources.ResourceManager;

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

```
    public void SetCulture(CultureInfo culture)
    {
        if (!_culture.Equals(culture))
        {
            _culture = culture;
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(null));
        }
    }

    public event PropertyChangedEventHandler? PropertyChanged;
}
```

Register in `App.axaml`:

```
<Application.Resources>
  <local:Loc x:Key="Loc"/>
</Application.Resources>
```

Use in XAML via indexer binding:

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

Switch culture at runtime:

```
var loc = (Loc)Application.Current!.Resources["Loc"];
loc.SetCulture(new CultureInfo("fr-FR"));
CultureInfo.CurrentCulture = CultureInfo.CurrentUICulture = new CultureInfo("fr-FR");
```

Reassigning `CurrentCulture` ensures format strings (`{0:C}`) use new culture.

4.2 Culture-aware formatting

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

Round-trip parsing uses `CultureInfo.CurrentCulture`. For manual conversions, pass `CultureInfo.CurrentCulture` to `TryParse`.

4.3 FlowDirection for RTL languages

```
<Window FlowDirection="RightToLeft">
  <StackPanel>
    <TextBlock Text="{Binding [Hello], Source={StaticResource Loc}}"/>
    <TextBox FlowDirection="LeftToRight" Text="{Binding Input}"/>
  </StackPanel>
</Window>
```

- RTL flips layout for panels and default icons. Use `FlowDirection.LeftToRight` for controls that should remain LTR (e.g., numbers).
- `FlowDirection` is defined in `Avalonia.Visuals/FlowDirection.cs`.

4.4 Input Method Editors (IME)

Text input (Asian languages) uses IME. Controls like `TextBox` handle IME automatically. When building custom text surfaces, implement `ITextInputMethodClient` (source: `TextInputMethodClient.cs`).

5. Fonts and fallbacks

Use fonts with wide Unicode coverage (Noto Sans, Segoe UI, Roboto). Set defaults via `FontManagerOptions` (Chapter 7). For script-specific fonts, add fallback chain:

```
AppBuilder.Configure<App>()
    .UsePlatformDetect()
    .With(new FontManagerOptions
    {
        DefaultFamilyName = "Noto Sans",
        FontFallbacks = new[]
        {
            new FontFallback { Family = "Noto Sans Arabic" },
            new FontFallback { Family = "Noto Sans CJK SC" }
        }
    })
    .LogToTrace();
```

Embed fonts for branding or to ensure glyph coverage. Use `FontFamily="avares://MyApp/Assets/Fonts/NotoSans.ttf#No` `Sans"` in styles.

6. Testing accessibility

- Manual: Tab through UI, run screen reader (Narrator, VoiceOver, Orca).
- Automated: Use UI test frameworks (Avalonia.Headless, Chapter 21) combined with `AutomationId` to verify accessibility properties.
- Tools: Contrast analyzers (Color Oracle, Stark), `Accessibility Insights` for Windows to inspect accessibility tree.

6.1 Inspecting automation tree

Avalonia DevTools (F12) -> Automation tab displays automation peers and properties. Confirm names, roles, help text.

7. Accessibility checklist

Keyboard - All interactive elements reachable via Tab/Shift+Tab. - Visible focus indicator (use styles to highlight `:focus` pseudo-class). - Access keys for primary commands.

Screen readers - Provide `AutomationProperties.Name/LabeledBy` for inputs. - Use `AutomationProperties.HelpText` for guidance. - Broadcast status updates via `AutomationProperties.LiveSetting`.

High contrast - Colors bound to theme resources; text meets contrast ratios. - Check `RequestedThemeVariant=HighContrast` for readability.

Internationalization - All strings from resources. - `CultureInfo.CurrentCulture/CurrentUICulture` update when switching language. - Layout supports `FlowDirection` changes. - Fonts cover required scripts.

8. Practice exercises

1. Add access keys and keyboard navigation for a form; verify focus order matches the spec.
2. Add `AutomationProperties.Name`, `HelpText`, and `AutomationId` to controls in a settings screen and test with Narrator or VoiceOver.
3. Localize UI strings into two additional cultures (e.g., es-ES, ar-SA), provide culture switching, and confirm RTL layout in Arabic.
4. Configure a default font fallback chain and verify glyph rendering for accented Latin, Cyrillic, Arabic, and CJK text.

5. Build an automated test (Avalonia.Headless) that finds elements via `AutomationId` and asserts localized content changes with culture.

Look under the hood (source bookmarks)

- Keyboard navigation: `KeyboardNavigation.cs`
- Access text: `AccessText.cs`
- Automation properties: `AutomationProperties.cs`
- Automation peers: `ControlAutomationPeer.cs`
- Flow direction: `FlowDirection.cs`
- Font manager options: `FontManagerOptions.cs`

Check yourself

- How do you connect a TextBox to its label so screen readers announce them together?
- Which property enables live region updates for status text?
- How do you switch UI language at runtime and refresh all localized bindings?
- Where do you configure font fallbacks to support multiple scripts?
- What steps ensure your UI handles high-contrast settings correctly?

What's next - Next: Chapter 16

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

Goal - Use Avalonia's storage provider to open, save, and enumerate files/folders across desktop, mobile, and browser. - Abstract file dialogs behind services so MVVM view models remain testable. - Handle drag-and-drop data (files, text, custom formats) and initiate drags from your app. - Work with the clipboard safely, including multi-format payloads.

Why this matters - Users expect native pickers, drag/drop, and clipboard support. Implementing them well keeps experiences consistent across platforms. - Proper abstractions keep storage logic off the UI thread and ready for unit testing.

Prerequisites - Chapter 9 (commands/input), Chapter 11 (MVVM), Chapter 12 (TopLevel services).

1. Storage provider fundamentals

All pickers live on `TopLevel.StorageProvider` (Window, control, etc.). The storage provider is an abstraction over native dialogs and sandbox rules.

```
var topLevel = TopLevel.GetTopLevel(control);
if (topLevel?.StorageProvider is { } storage)
{
    // storage.OpenFilePickerAsync(...)
}
```

If `StorageProvider` is null, ensure the control is attached (e.g., call after `Loaded`/`Opened`).

1.1 Service abstraction for MVVM

```
public interface IFileDialogService
{
    Task<IReadOnlyList<IStorageFile>> OpenFilesAsync(FilePickerOpenOptions options);
    Task<IStorageFile?> SaveFileAsync(FilePickerSaveOptions options);
    Task<IStorageFolder?> PickFolderAsync(FolderPickerOpenOptions options);
}

public sealed class FileDialogService : IFileDialogService
{
    private readonly TopLevel _topLevel;
    public FileDialogService(TopLevel topLevel) => _topLevel = topLevel;

    public Task<IReadOnlyList<IStorageFile>> OpenFilesAsync(FilePickerOpenOptions options)
        => _topLevel.StorageProvider?.OpenFilePickerAsync(options) ?? Task.FromResult<IReadOnlyList<ISt

    public Task<IStorageFile?> SaveFileAsync(FilePickerSaveOptions options)
        => _topLevel.StorageProvider?.SaveFilePickerAsync(options) ?? Task.FromResult<IStorageFile?>(nu

    public async Task<IStorageFolder?> PickFolderAsync(FolderPickerOpenOptions options)
    {
        if (_topLevel.StorageProvider is null)
            return null;
        var folders = await _topLevel.StorageProvider.OpenFolderPickerAsync(options);
        return folders.FirstOrDefault();
    }
}
```

Register the service per window (in DI) so view models request dialogs via `IFileDialogService` without touching UI types.

2. Opening files (async streams)

```
public async Task<string?> ReadTextFileAsync(IStorageFile file, CancellationToken ct)
{
    await using var stream = await file.OpenReadAsync();
    using var reader = new StreamReader(stream, Encoding.UTF8, detectEncodingFromByteOrderMarks: true);
    return await reader.ReadToEndAsync(ct);
}
```

- Always wrap streams in `using`/`await using`.
- Pass `CancellationToken` to long operations.
- For binary files, use `BinaryReader` or direct `Stream` APIs.

2.1 Remote or sandboxed locations

On Android/iOS/Browser the returned stream might be virtual (no direct file path). Always rely on stream APIs; avoid `LocalPath` if `Path` is null.

2.2 File type filters

```
var options = new FilePickerOpenOptions
{
    Title = "Open images",
    AllowMultiple = true,
    SuggestedStartLocation = await storage.TryGetWellKnownFolderAsync(WellKnownFolder.Pictures),
    FileTypeFilter = new[]
    {
        new FilePickerFileType("Images")
        {
            Patterns = new[] { "*.png", "*.jpg", "*.jpeg", "*.webp", "*.gif" }
        }
    }
};
```

`TryGetWellKnownFolderAsync` returns common directories when supported (desktop/mobile). Source: `WellKnownFolder.cs`.

3. Saving files

```
var saveOptions = new FilePickerSaveOptions
{
    Title = "Export report",
    SuggestedFileName = $"report-{DateTime.UtcNow:yyyyMMdd}.csv",
    DefaultExtension = "csv",
    FileTypeChoices = new[]
    {
        new FilePickerFileType("CSV") { Patterns = new[] { "*.csv" } },
        new FilePickerFileType("All files") { Patterns = new[] { "*" } }
    }
};

var file = await _dialogService.SaveFileAsync(saveOptions);
if (file is not null)
{
    await using var stream = await file.OpenWriteAsync();
    await using var writer = new StreamWriter(stream, Encoding.UTF8, leaveOpen: false);
```

```
    await writer.WriteLineAsync("Id,Name,Email");
    foreach (var row in rows)
        await writer.WriteLineAsync($"{row.Id},{row.Name},{row.Email}");
}
```

- `OpenWriteAsync` truncates the existing file. Use `OpenReadWriteAsync` for editing.
- Some platforms prompt for confirmation when writing to previously granted locations.

4. Enumerating folders

```
var folder = await storage.TryGetFolderFromPathAsync(new Uri("file:///C:/Logs"));
if (folder is not null)
{
    await foreach (var item in folder.GetItemsAsync())
    {
        switch (item)
        {
            case IStorageFile file:
                // Process file
                break;
            case IStorageFolder subfolder:
                // Recurse or display
                break;
        }
    }
}
```

`GetItemsAsync()` returns an async sequence; iterate with `await foreach` on .NET 7+. Use `GetFilesAsync`/`GetFoldersAsync` to filter.

5. Platform notes

Platform	Storage provider	Considerations
Windows/macOS/Linux	Native dialogs; file system access	Standard read/write. Some Linux desktops require portals (Flatpak/Snap).
Android/iOS	Native pickers; sandboxed URIs	Streams may be content URIs; persist permissions if needed.
Browser (WASM)	File System Access API	Requires user gestures; may return handles that expire when page reloads.

Wrap storage calls in try/catch to handle permission denials or canceled dialogs gracefully.

6. Drag-and-drop: receiving data

```
<Border AllowDrop="True"
        DragOver="OnDragOver"
        Drop="OnDrop"
        Background="#111827" Padding="12">
  <TextBlock Text="Drop files or text" Foreground="#CBD5F5"/>
</Border>
```

```
private void OnDragOver(object? sender, DragEventArgs e)
{
    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)
{
    var files = await e.Data.GetFilesAsync();
    if (files is not null)
    {
        foreach (var item in files.OfType<IStorageFile>())
        {
            await using var stream = await item.OpenReadAsync();
            // import
        }
        return;
    }

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

- `GetFilesAsync()` returns storage items; check for `IStorageFile`.
- Inspect `e.KeyModifiers` to adjust behavior (e.g., Ctrl for copy).

6.1 Initiating drag-and-drop

```
private async void DragSource_PointerPressed(object? sender, PointerPressedEventArgs e)
{
    if (sender is not Control control)
        return;

    var data = new DataObject();
    data.Set(DataFormats.Text, "Example text");

    var effects = await DragDrop.DoDragDrop(e, data, DragDropEffects.Copy | DragDropEffects.Move);
    if (effects.HasFlag(DragDropEffects.Move))
    {
        // remove item
    }
}
```

`DataObject` supports multiple formats (text, files, custom types). For custom data, both source and target must agree on a format string.

7. Clipboard operations

```
public interface IClipboardService
{
```

```
    Task SetTextAsync(string text);
    Task<string?> GetTextAsync();
    Task SetDataObjectAsync(IDataObject dataObject);
    Task<IReadOnlyList<string>> GetFormatsAsync();
}

public sealed class ClipboardService : IClipboardService
{
    private readonly TopLevel _topLevel;
    public ClipboardService(TopLevel topLevel) => _topLevel = topLevel;

    public Task SetTextAsync(string text) => _topLevel.Clipboard?.SetTextAsync(text) ?? Task.CompletedTa
    public Task<string?> GetTextAsync() => _topLevel.Clipboard?.GetTextAsync() ?? Task.FromResult<string
    public Task SetDataObjectAsync(IDataObject dataObject) => _topLevel.Clipboard?.SetDataObjectAsync(da
    public Task<IReadOnlyList<string>> GetFormatsAsync() => _topLevel.Clipboard?.GetFormatsAsync() ?? Ta
}
```

7.1 Multi-format clipboard payload

```
var dataObject = new DataObject();
dataObject.Set(DataFormats.Text, "Plain text");
dataObject.Set("text/html", "<strong>Bold</strong>");
dataObject.Set("application/x-myapp-item", myItemId);

await clipboardService.SetDataObjectAsync(dataObject);
var formats = await clipboardService.GetFormatsAsync();
```

Browser restrictions: clipboard APIs require user gesture and may only allow text formats.

8. Error handling & async patterns

- Wrap storage operations in try/catch for `IOException`, `UnauthorizedAccessException`.
- Offload heavy parsing to background threads with `Task.Run` (keep UI thread responsive).
- Use `Progress<T>` to report progress to view models.

```
var progress = new Progress<int>(value => ImportProgress = value);
await _importService.ImportAsync(file, progress, cancellationToken);
```

9. Diagnostics

- Log storage/drag errors with `LogArea.Platform` or custom logger.
- DevTools -> Events tab shows drag/drop events.
- On Linux portals (Flatpak/Snap), check console logs for portal errors.

10. Practice exercises

1. Implement `IFileDialogService` and expose commands for Open, Save, and Pick Folder; update the UI with results.
2. Build a log viewer that watches a folder, importing new files via drag-and-drop or Open dialog.
3. Create a clipboard history panel that stores the last N text snippets using the `IClipboard` service.
4. Add drag support from a list to the OS shell (export files) and confirm the OS receives them.
5. Implement cancellation for long-running file imports and confirm resources are disposed when canceled.

Look under the hood (source bookmarks)

- Storage provider: `IStorageProvider`

- File/folder abstractions: `IStorageFile`, `IStorageFolder`
- Picker options: `FilePickerOpenOptions`, `FilePickerSaveOptions`
- Drag/drop: `DragDrop.cs`, `DataObject.cs`
- Clipboard: `IClipboard`

Check yourself

- How do you obtain an `IStorageProvider` when you only have a view model?
- What are the advantages of using asynchronous streams (`await using`) when reading/writing files?
- How can you detect which drag/drop formats are available during a drop event?
- Which APIs let you enumerate well-known folders cross-platform?
- What restrictions exist for clipboard and storage operations on browser/mobile?

What's next - Next: Chapter 17

17. Background work and networking

Goal - Keep the UI responsive while doing heavy or long-running tasks using async/await, Task.Run, and progress reporting. - Surface status, progress, and cancellation to users. - Call web APIs with HttpClient, handle retries/timeouts, and stream downloads/upload. - Respond to connectivity changes and test background logic predictably.

Why this matters - Real apps load data, crunch files, and hit APIs. Blocking the UI thread ruins UX. - Async-first code scales across desktop, mobile, and browser with minimal changes.

Prerequisites - Chapters 8-9 (binding & commands), Chapter 11 (MVVM), Chapter 16 (file IO).

1. The UI thread and Dispatcher

Avalonia has a single UI thread managed by `Dispatcher.UIThread`. UI elements and bound properties must be updated on this thread.

Rules of thumb: - Prefer async I/O (await network/file operations). - For CPU-bound work, use `Task.Run` to offload to a thread pool thread. - Use `Dispatcher.UIThread.Post/InvokeAsync` to marshal back to the UI thread if needed (though `Progress<T>` usually keeps you on the UI thread).

```
await Dispatcher.UIThread.InvokeAsync(() => Status = "Ready");
```

2. Async workflow pattern (ViewModel)

```
public sealed class WorkViewModel : ObservableObject
{
    private CancellationTokenSource? _cts;
    private double _progress;
    private string _status = "Idle";
    private bool _isBusy;

    public double Progress { get => _progress; set => SetProperty(ref _progress, value); }
    public string Status { get => _status; set => SetProperty(ref _status, value); }
    public bool IsBusy { get => _isBusy; set => SetProperty(ref _isBusy, value); }

    public RelayCommand StartCommand { get; }
    public RelayCommand CancelCommand { get; }

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

    private async Task StartAsync()
    {
        IsBusy = true;
        _cts = new CancellationTokenSource();
        var progress = new Progress<double>(value => Progress = value * 100);

        try
        {
            Status = "Processing...";
            await FakeWorkAsync(progress, _cts.Token);
            Status = "Completed";
```

```
        }
        catch (OperationCanceledException)
        {
            Status = "Canceled";
        }
        catch (Exception ex)
        {
            Status = $"Error: {ex.Message}";
        }
        finally
        {
            IsBusy = false;
            _cts = null;
        }
    }

    private static async Task FakeWorkAsync(IProgress<double> progress, CancellationToken ct)
    {
        const int total = 1000;
        await Task.Run(async () =>
        {
            for (int i = 0; i < total; i++)
            {
                ct.ThrowIfCancellationRequested();
                await Task.Delay(2, ct).ConfigureAwait(false);
                progress.Report((i + 1) / (double)total);
            }
        }, ct);
    }
}
```

`Task.Run` offloads CPU work to the thread pool; `ConfigureAwait(false)` keeps the inner loop on the background thread. `Progress<T>` marshals results back to UI thread automatically.

3. UI binding (XAML)

```
<StackPanel Spacing="12">
  <ProgressBar Minimum="0" Maximum="100" Value="{Binding Progress}" IsIndeterminate="{Binding IsBusy}"/>
  <TextBlock Text="{Binding Status}"/>
  <StackPanel Orientation="Horizontal" Spacing="8">
    <Button Content="Start" Command="{Binding StartCommand}"/>
    <Button Content="Cancel" Command="{Binding CancelCommand}"/>
  </StackPanel>
</StackPanel>
```

4. HTTP networking patterns

4.1 HttpClient lifetime

Reuse HttpClient (per host/service) to avoid socket exhaustion. Inject or hold static instance.

```
public static class ApiClient
{
    public static HttpClient Instance { get; } = new HttpClient
    {
        Timeout = TimeSpan.FromSeconds(30)
```

```
    };
}
```

4.2 GET + JSON

```
public async Task<T?> GetJsonAsync<T>(string url, CancellationToken ct)
{
    using var resp = await ApiClient.Instance.GetAsync(url, HttpCompletionOption.ResponseHeadersRead, c
    resp.EnsureSuccessStatusCode();
    await using var stream = await resp.Content.ReadAsStreamAsync(ct);
    return await JsonSerializer.DeserializeAsync<T>(stream, cancellationToken: ct);
}
```

4.3 POST JSON with retry

```
public async Task PostWithRetryAsync<T>(string url, T payload, CancellationToken ct)
{
    var policy = Policy
        .Handle<HttpRequestException>()
        .Or<TaskCanceledException>()
        .WaitAndRetryAsync(3, attempt => TimeSpan.FromSeconds(Math.Pow(2, attempt))); // exponential ba

    await policy.ExecuteAsync(async token =>
    {
        using var response = await ApiClient.Instance.PostAsJsonAsync(url, payload, token);
        response.EnsureSuccessStatusCode();
    }, ct);
}
```

Use `Polly` or custom retry logic. Timeouts and cancellation tokens help stop hanging requests.

4.4 Download with progress

```
public async Task DownloadAsync(Uri uri, IStorageFile destination, IProgress<double> progress, Cancella
{
    using var response = await ApiClient.Instance.GetAsync(uri, HttpCompletionOption.ResponseHeadersRea
    response.EnsureSuccessStatusCode();

    var contentLength = response.Content.Headers.ContentLength;
    await using var httpStream = await response.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(totalRead / (double)contentLength.Value);
    }
}
```

5. Connectivity awareness

Avalonia doesn't ship built-in connectivity events; rely on platform APIs or ping endpoints.

- Desktop: use `System.Net.NetworkInformation.NetworkChange` events.
- Mobile: Xamarin/MAUI style libraries or platform-specific checks.
- Browser: `navigator.onLine` via JS interop.

Expose a service to signal connectivity changes to view models; keep offline caching in mind.

6. Background services & scheduled work

For periodic tasks, use `DispatcherTimer` on UI thread or `Task.Run` loops with delays.

```
var timer = new DispatcherTimer(TimeSpan.FromMinutes(5), DispatcherPriority.Background, (_, _) => Refres
timer.Start();
```

Long-running background work should check `CancellationToken` frequently, especially when app might suspend (mobile).

7. Testing background code

Use `Task.Delay` injection or `ITestScheduler` (ReactiveUI) to control time. For plain async code, wrap delays in an interface to mock in tests.

```
public interface IDelayProvider
{
    Task Delay(TimeSpan time, CancellationToken ct);
}

public sealed class DelayProvider : IDelayProvider
{
    public Task Delay(TimeSpan time, CancellationToken ct) => Task.Delay(time, ct);
}
```

Inject and replace with deterministic delays in tests.

8. Browser (WebAssembly) considerations

- HttpClient uses fetch; CORS applies.
- WebSockets available via `ClientWebSocket` when allowed by browser.
- Long-running loops should yield frequently (`await Task.Yield()`) to avoid blocking JS event loop.

9. Practice exercises

1. Build a data sync command that fetches JSON from an API, parses it, and updates view models without freezing UI.
2. Add cancellation and progress reporting to a file import feature (Chapter 16) using `IProgress<double>`.
3. Implement retry with exponential backoff around a flaky endpoint and show status messages when retries occur.
4. Detect connectivity loss and display an offline banner; queue commands to run when back online.
5. Write a unit test that confirms cancellation stops a long-running operation before completion.

Look under the hood (source bookmarks)

- Dispatcher & UI thread: `Dispatcher.cs`
- Progress reporting: `Progress<T>`
- HttpClient guidance: .NET HttpClient docs

- Cancellation tokens: .NET cancellation docs

Check yourself

- Why does blocking the UI thread freeze the app? How do you keep it responsive?
- How do you propagate cancellation through nested async calls?
- Which HttpClient features help prevent hung requests?
- How can you provide progress updates without touching `Dispatcher.UIThread` manually?
- What adjustments are needed when running the same code on the browser?

What's next - Next: Chapter 18

18. Desktop targets: Windows, macOS, Linux

Goal - Master Avalonia's desktop-specific features: window chrome, transparency, DPI/multi-monitor handling, platform capabilities, and packaging essentials. - Understand per-platform caveats so your desktop app feels native on Windows, macOS, and Linux.

Why this matters - Desktop users expect native window behavior, correct scaling, and integration with OS features (taskbar/dock, notifications). - Avalonia abstracts the basics but you still need to apply platform-specific tweaks.

Prerequisites - Chapter 4 (lifetimes), Chapter 12 (window navigation), Chapter 13 (menus/dialogs), Chapter 16 (storage).

1. Window fundamentals

```
<Window xmlns="https://github.com/avaloniaui"
        x:Class="MyApp.MainWindow"
        Width="1024" Height="720"
        CanResize="True"
        SizeToContent="Manual"
        WindowStartupLocation="CenterScreen"
        ShowInTaskbar="True"
        Topmost="False"
        Title="My App">

</Window>
```

Properties: - `WindowState`: Normal, Minimized, Maximized, FullScreen. - `CanResize`, `CanMinimize`, `CanMaximize` control system caption buttons. - `SizeToContent`: `Manual`, `Width`, `Height`, `WidthAndHeight` (works best before window is shown). - `WindowStartupLocation`: `Manual` (default), `CenterScreen`, `CenterOwner`. - `ShowInTaskbar`: show/hide taskbar/dock icon. - `Topmost`: keep above other windows.

Persist position/size between runs:

```
protected override void OnOpened(EventArgs e)
{
    base.OnOpened(e);
    if (LocalSettings.TryReadWindowPlacement(out var placement))
    {
        Position = placement.Position;
        Width = placement.Width;
        Height = placement.Height;
        WindowState = placement.State;
    }
}

protected override void OnClosing(WindowClosingEventArgs e)
{
    base.OnClosing(e);
    LocalSettings.WriteWindowPlacement(new WindowPlacement
    {
        Position = Position,
        Width = Width,
        Height = Height,
        State = WindowState
    });
}
```

2. Custom title bars and chrome

`SystemDecorations="None"` removes native chrome; use extend-client-area hints for custom title bars.

```
<Window SystemDecorations="None"
        ExtendClientAreaToDecorationsHint="True"
        ExtendClientAreaChromeHints="PreferSystemChrome"
        ExtendClientAreaTitleBarHeightHint="32">
  <Grid>
    <Border Background="#1F2937" Height="32" VerticalAlignment="Top"
            PointerPressed="TitleBar_PointerPressed">
      <StackPanel Orientation="Horizontal" Margin="12,0" VerticalAlignment="Center" Spacing="12">
        <TextBlock Text="My App" Foreground="White"/>

        <Border x:Name="CloseButton" Width="32" Height="24" Background="Transparent"
                PointerPressed="CloseButton_PointerPressed">
          <Path Stroke="White" StrokeThickness="2" Data="M2,2 L10,10 M10,2 L2,10" HorizontalAlignment="(
        </Border>
      </StackPanel>
    </Border>

  </Grid>
</Window>
```

```
private void TitleBar_PointerPressed(object? sender, PointerPressedEventArgs e)
{
    if (e.GetCurrentPoint(this).Properties.IsLeftButtonPressed)
        BeginMoveDrag(e);
}

private void CloseButton_PointerPressed(object? sender, PointerPressedEventArgs e)
{
    Close();
}
```

- Provide hover/pressed styles for buttons.
- Add keyboard/screen reader support (AutomationProperties).

3. Window transparency & effects

```
<Window TransparencyLevelHint="Mica, AcrylicBlur, Blur, Transparent">

</Window>
```

```
TransparencyLevelHint = new[]
{
    WindowTransparencyLevel.Mica,
    WindowTransparencyLevel.AcrylicBlur,
    WindowTransparencyLevel.Blur,
    WindowTransparencyLevel.Transparent
};

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

Platform support summary (subject to OS version, composition mode): - Windows 10/11: Transparent, Blur, AcrylicBlur, Mica (Win11). - macOS: Transparent, Blur (vibrancy). - Linux (compositor dependent):

Transparent, Blur.

Design for fallback: ActualTransparencyLevel may be `None`–ensure backgrounds look good without blur.

4. Screens, DPI, and scaling

- `Screens`: enumerate monitors (`Screens.All`, `Screens.Primary`).
- `Screen.WorkingArea`: available area excluding taskbar/dock.
- `Screen.Scaling`: per-monitor scale.
- `Window.DesktopScaling`: DIP to physical pixel ratio for positioning.
- `TopLevel.RenderScaling`: DPI scaling for rendering (affects pixel alignment).

Center on active screen:

```
protected override void OnOpened(EventArgs e)
{
    base.OnOpened(e);
    var currentScreen = Screens?.ScreenFromWindow(this) ?? Screens?.Primary;
    if (currentScreen is null)
        return;

    var frameSize = PixelSize.FromSize(ClientSize, DesktopScaling);
    var target = currentScreen.WorkingArea.CenterRect(frameSize);
    Position = target.Position;
}
```

Handle scaling changes when moving between monitors:

```
ScalingChanged += (_, _) =>
{
    // Renderer scaling updated; adjust cached bitmaps if necessary.
};
```

5. Platform integration

5.1 Windows

- Taskbar/dock menus: use Jump Lists via `System.Windows.Shell` interop or community packages.
- Notifications: `WindowNotificationManager` or Windows toast (via WinRT APIs).
- Acrylic/Mica: require Windows 10 or 11; fallback on earlier versions.
- System backdrops: set `TransparencyLevelHint` and ensure the OS supports it; consider theme-aware backgrounds.

5.2 macOS

- Menu bar: use `NativeMenuBar` (Chapter 13).
- Dock menu: `NativeMenuBar.Menu` can include items that appear in dock menu.
- Application events (Quit, About): integrate with `AvaloniaNativeMenuCommands` or handle native application events.
- Fullscreen: Mac expects toggle via green traffic-light button; `WindowState.FullScreen` works, but ensure custom chrome still accessible.

5.3 Linux

- Variety of window managers; test SystemDecorations/ExtendClientArea on GNOME/KDE.
- Transparency requires compositor (e.g., Mutter, KWin). Provide fallback.
- Fractional scaling support varies; check `RenderScaling` for the active monitor.
- Packaging (Flatpak, Snap, AppImage) may affect file dialog behavior (portal APIs).

6. Packaging & deployment overview

- Windows: `dotnet publish -r win-x64 --self-contained` or MSIX via `dotnet publish /p:PublishTrimmed=false /p:WindowsPackageType=msix`.
- macOS: `.app` bundle; codesign and notarize for distribution (`dotnet publish -r osx-x64 --self-contained` followed by bundle packaging via Avalonia templates or scripts).
- Linux: produce .deb/.rpm, AppImage, or Flatpak; ensure dependencies (GTK, Skia) available.

Reference docs: Avalonia publishing guide (docs/publish.md).

7. Multiple window management tips

- Track open windows via `ApplicationLifetime.Windows` (desktop only).
- Use `IClassicDesktopStyleApplicationLifetime.Exit` to exit the app.
- Owner/child relationships ensure modality, centering, and Z-order (Chapter 12).
- Provide "Move to Next Monitor" command by cycling through `Screens.All` and setting `Position` accordingly.

8. Troubleshooting

Issue	Fix
Window blurry on high DPI	Use vector assets; adjust RenderScaling; ensure `UseCompositor` is default
Transparency ignored	Check ActualTransparencyLevel; verify OS support; remove conflicting settings
Custom chrome drag fails	Ensure `BeginMoveDrag` only on left button down; avoid starting drag from interactive controls
Incorrect monitor on startup	Set `WindowStartupLocation` or compute position using `Screens` before showing window
Linux packaging fails	Include `libAvaloniaNative.so` dependencies; use Avalonia Debian/RPM packaging scripts

9. Practice exercises

1. Build a window with custom title bar, including minimize, maximize, close, and move/resize handles.
2. Request Mica/Acrylic, detect fallback, and apply theme-specific backgrounds for each transparency level.
3. Implement a "Move to Next Monitor" command cycling through available screens.
4. Persist window placement (position/size/state) to disk and restore on startup.
5. Create deployment artifacts: MSIX (Windows), .app (macOS), and AppImage/AppImage (Linux) for a simple app.

Look under the hood (source bookmarks)

- Window & TopLevel: `Window.cs`, `TopLevel.cs`
- Transparency enums: `WindowTransparencyLevel.cs`
- Screens API: `Screens.cs`
- Extend client area hints: `Window.cs` lines around ExtendClientArea properties
- Desktop lifetime: `ClassicDesktopStyleApplicationLifetime.cs`

Check yourself

- How do you request and detect the achieved transparency level on each platform?
- What steps are needed to build a custom title bar that supports drag and resize?

- How do you center a window on the active monitor using `Screens` and scaling info?
- What packaging options are available per desktop platform?

What's next - Next: Chapter 19

19. Mobile targets: Android and iOS

Goal - Configure, build, and run Avalonia apps on Android and iOS using the single-project workflow. - Understand single-view lifetimes, navigation patterns, safe areas, and mobile services (storage, clipboard, permissions). - Integrate platform-specific features (back button, app icons, splash screens) while keeping shared MVVM architecture.

Why this matters - Mobile devices have different UI expectations (single window, touch, safe areas, OS-managed lifecycle). - Avalonia lets you share code across desktop and mobile, but you must adjust windowing, navigation, and services.

Prerequisites - Chapter 12 (lifetimes/navigation), Chapter 16 (storage provider), Chapter 17 (async/networking).

1. Projects and workload setup

Install .NET workloads and mobile SDKs:

```
# Android
sudo dotnet workload install android

# iOS (macOS only)
sudo dotnet workload install ios

# Optional: wasm-tools for browser
sudo dotnet workload install wasm-tools
```

Check workloads with `dotnet workload list`.

Project structure: - Shared project (e.g., `MyApp`): Avalonia cross-platform code. - Platform heads (Android, iOS): host the Avalonia app, provide manifests, icons, metadata.

Avalonia templates (`dotnet new avalonia.app --multiplatform`) create the shared project plus heads (`MyApp.Android`, `MyApp.iOS`).

2. Single-view lifetime

`ISingleViewApplicationLifetime` hosts one root view. Configure in `App.OnFrameworkInitializationCompleted` (Chapter 4 showed desktop branch).

```
public override void OnFrameworkInitializationCompleted()
{
    var services = ConfigureServices();

    if (ApplicationLifetime is ISingleViewApplicationLifetime singleView)
    {
        singleView.MainView = services.GetRequiredService<ShellView>();
    }
    else if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
    {
        desktop.MainWindow = services.GetRequiredService<MainWindow>();
    }

    base.OnFrameworkInitializationCompleted();
}
```

`ShellView` is a `UserControl` with mobile-friendly layout and navigation.

3. Mobile navigation patterns

Use view-model-first navigation (Chapter 12) but ensure a visible Back control.

```
<UserControl xmlns="https://github.com/avaloniaui" x:Class="MyApp.Views.ShellView">
  <Grid RowDefinitions="Auto,*">
    <StackPanel Orientation="Horizontal" Spacing="8" Margin="16">
      <Button Content="Back"
              Command="{Binding BackCommand}"
              IsVisible="{Binding CanGoBack}"/>
      <TextBlock Text="{Binding Title}" FontSize="20" VerticalAlignment="Center"/>
    </StackPanel>
    <TransitioningContentControl Grid.Row="1" Content="{Binding Current}"/>
  </Grid>
</UserControl>
```

`ShellViewModel` keeps a stack of view models and implements `BackCommand`/`NavigateTo`. Hook Android back button (Next section) to `BackCommand`.

4. Safe areas and input insets

Phones have notches and OS-controlled bars. Use `IInsetsManager` to apply safe-area padding.

```
public partial class ShellView : UserControl
{
    public ShellView()
    {
        InitializeComponent();
        this.AttachedToVisualTree += (_, __) =>
        {
            var top = TopLevel.GetTopLevel(this);
            var insets = top?.InsetsManager;
            if (insets is null) return;

            void ApplyInsets()
            {
                RootPanel.Padding = new Thickness(
                    insets.SafeAreaPadding.Left,
                    insets.SafeAreaPadding.Top,
                    insets.SafeAreaPadding.Right,
                    insets.SafeAreaPadding.Bottom);
            }

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

Soft keyboard (IME) adjustments: subscribe to `TopLevel.InputPane.Showing/Hiding` and adjust margins above keyboard.

```
var pane = top?.InputPane;
if (pane is not null)
{
    pane.Showing += (_, args) => RootPanel.Margin = new Thickness(0, 0, 0, args.OccludedRect.Height);
```

```
    pane.Hiding += (_, __) => RootPanel.Margin = new Thickness(0);
}
```

5. Platform head customization

5.1 Android head (MyApp.Android)

- `MainActivity.cs` hosts Avalonia.
- `AndroidManifest.xml`: declare permissions (`INTERNET`, `READ_EXTERNAL_STORAGE`), orientation, minimum SDK.
- App icons/splash: `Resources/mipmap-*`, `Resources/layout` for splash.
- Intercept hardware Back button: override `OnBackPressed` to call service.

```
public override void OnBackPressed()
{
    if (!AvaloniaApp.Current?.TryGoBack() ?? true)
        base.OnBackPressed();
}
```

`TryGoBack` calls into shared navigation service and returns true if you consumed the event.

5.2 iOS head (MyApp.iOS)

- `AppDelegate.cs` sets up Avalonia.
- `Info.plist`: permissions (e.g., camera), orientation, status bar style.
- Launch screen via `LaunchScreen.storyboard` or SwiftUI resources.

Handle universal links or background tasks by bridging to shared services in `AppDelegate`.

6. Permissions & storage

- StorageProvider works but returns sandboxed streams. Request platform permissions:
  - Android: declare `<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>` and use runtime requests.
  - iOS: add entries to Info.plist (e.g., `NSPhotoLibraryUsageDescription`).
- Consider packaging specific data (e.g., from `AppBundle`) instead of relying on arbitrary file system access.

7. Touch and gesture design

- Ensure controls are at least 44x44 DIP.
- Provide ripple/highlight states for buttons (Fluent theme handles this). Avoid hover-only interactions.
- Use `Tapped`/`DoubleTapped` events for simple gestures; `PointerGestureRecognizer` for advanced ones.

8. Performance & profiling

- Keep navigation stacks small; heavy animations may impact lower-end devices.
- Profile with Android Studio's profiler / Xcode Instruments for CPU, memory, GPU.
- When using `Task.Run`, consider battery impact; use async I/O where possible.

9. Packaging and deployment

Android

```
cd MyApp.Android
# Debug build to device
msbuild /t:Run /p:Configuration=Debug
```

```
# Release APK/AAB
msbuild /t:Publish /p:Configuration=Release /p:AndroidPackageFormat=aab
```

Sign with keystore for app store.

iOS

- Use Xcode to build and deploy to simulator/device. `dotnet build -t:Run -f net8.0-ios` works on macOS with Xcode installed.
- Provisioning profiles & certificates required for devices/app store.

10. Browser compatibility (bonus)

Mobile code often reuses single-view logic for WebAssembly. Check `ApplicationLifetime` for `BrowserSingleViewLifetime` and swap to a `ShellView`. Storage/clipboard behave like Chapter 16 with browser limitations.

11. Practice exercises

1. Configure the Android/iOS heads and run the app on emulator/simulator with a shared `ShellView`.
2. Implement a navigation service with back stack and wire Android back button to it.
3. Adjust safe-area padding and keyboard insets for a login screen (Inputs remain visible when keyboard shows).
4. Add file pickers via `StorageProvider` and test on device (consider permission prompts).
5. Package a release build (.aab for Android, .ipa for iOS) and validate icons/splash screens.

Look under the hood (source bookmarks)

- Single-view lifetime: `SingleViewApplicationLifetime.cs`
- Input pane (soft keyboard): `IInputPane`
- Insets manager: `IInsetsManager`
- Android platform project: `src/Android`
- iOS platform project: `src/iOS`
- Mobile samples: `samples/ControlCatalog.Android`, `samples/ControlCatalog.iOS`

Check yourself

- How does the navigation pattern differ between desktop and mobile? How do you surface back navigation?
- How do you ensure inputs remain visible when the on-screen keyboard appears?
- What permission declarations are required for file access on Android/iOS?
- Where in the platform heads do you configure icons, splash screens, and orientation?

What's next - Next: Chapter 20

20. Browser (WebAssembly) target

Goal - Run your Avalonia app in the browser using WebAssembly (WASM) with minimal changes to shared code. - Understand browser-specific lifetimes, hosting options, rendering modes, and platform limitations (files, networking, threading). - Debug, package, and deploy a browser build with confidence.

Why this matters - Web delivery eliminates install friction for demos, tooling, and dashboards. - Browser rules (sandboxing, CORS, user gestures) require tweaks compared to desktop/mobile.

Prerequisites - Chapter 19 (single-view navigation), Chapter 16 (storage provider), Chapter 17 (async/networking).

1. Project structure and setup

Install `wasm-tools` workload:

```
sudo dotnet workload install wasm-tools
```

A multi-target solution has: - Shared project (`MyApp`): Avalonia code. - Browser head (`MyApp.Browser`): hosts the app (Program.cs, index.html, static assets).

Avalonia template (`dotnet new avalonia.app --multiplatform`) can create the browser head for you.

2. Start the browser app

`StartBrowserAppAsync` attaches Avalonia to a DOM element by ID.

```
using Avalonia;
using Avalonia.Browser;

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

    public static Task Main(string[] args)
        => BuildAvaloniaApp()
            .StartBrowserAppAsync("out");
}
```

Ensure host HTML contains `<div id="out"></div>`.

For advanced embedding, use `SetupBrowserAppAsync` to control when/where you attach views.

3. Single view lifetime

Browser uses `ISingleViewApplicationLifetime` (same as mobile). Configure in `App.OnFrameworkInitializationComplete`

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

    base.OnFrameworkInitializationCompleted();
}
```

Navigation patterns from Chapter 19 apply (content control with back stack).

4. Rendering options

Configure `BrowserPlatformOptions` to choose rendering mode and polyfills.

```
await BuildAvaloniaApp().StartBrowserAppAsync(
    "out",
    new BrowserPlatformOptions
    {
        RenderingMode = new[]
        {
            BrowserRenderingMode.WebGL2,
            BrowserRenderingMode.WebGL1,
            BrowserRenderingMode.Software2D
        },
        RegisterAvaloniaServiceWorker = true,
        AvaloniaServiceWorkerScope = "/",
        PreferFileDialogPolyfill = false,
        PreferManagedThreadDispatcher = true
    });
```

- WebGL2: best performance (default when supported).
- WebGL1: fallback for older browsers.
- Software2D: ultimate fallback (slower).
- Service worker: required for save-file polyfill; serve over HTTPS/localhost.
- `PreferManagedThreadDispatcher`: run dispatcher on worker thread when WASM threading enabled (requires server sending COOP/COEP headers).

5. Storage and file dialogs

`IStorageProvider` uses the File System Access API when available; otherwise a polyfill (service worker + download anchor) handles saves.

Limitations: - Browsers require user gestures (click) to open dialogs. - File handles may not persist between sessions; use IDs and re-request access if needed. - No direct file system access outside the user-chosen handles.

Example save using polyfill-friendly code (Chapter 16 shows full pattern). Test with/without service worker to ensure both paths work.

6. Clipboard & drag-drop

Clipboard operations require user gestures and may only support text formats. - `Clipboard.SetTextAsync` works after user interaction (button click). - Advanced formats require clipboard permissions or aren't supported.

Drag/drop from browser to app is supported, but dragging files out of the app is limited by browser APIs.

7. Networking & CORS

- HttpClient uses `fetch`. All requests obey CORS. Configure server with correct `Access-Control-Allow-*` headers.
- WebSockets supported via `ClientWebSocket` if server enables them.
- HTTPS recommended; some APIs (clipboard, file access) require secure context.

8. JavaScript interop

Call JS via `window.JSObject` or `JSRuntime` helpers (Avalonia.Browser exposes interop helpers). Example:

```
using Avalonia.Browser.Interop;

await JSRuntime.InvokeVoidAsync("console.log", "Hello from Avalonia");
```

Use interop to integrate with existing web components or to access Web APIs not wrapped by Avalonia.

9. Hosting in Blazor (optional)

`Avalonia.Browser.Blazor` lets you embed Avalonia controls in a Blazor app. Example sample: `ControlCatalog.Browser.Blazor`. Use when you need Blazor's routing/layout but Avalonia UI inside components.

10. Debugging

- Inspector: use browser devtools (F12). Evaluate DOM, watch console logs.
- Source maps: publish with `dotnet publish -c Debug` to get wasm debugging symbols for supported browsers.
- Logging: `AppBuilder.LogToTrace()` outputs to console.
- Performance: use Performance tab to profile frames, memory, CPU.

11. Deployment

Publish the browser head:

```
cd MyApp.Browser
# Debug
dotnet run
# Release bundle
dotnet publish -c Release
```

Output under `bin/Release/net8.0/browser-wasm/AppBundle`. Serve via static web server (ASP.NET, Node, Nginx, GitHub Pages). Ensure service worker scope matches hosting path.

Remember to enable compression (Brotli) for faster load times.

12. Platform limitations

Feature	Browser behavior
Windows/Dialogs	Single view only; no OS windows, tray icons, native menus
File system	User-selection only via pickers; no arbitrary file access
Threading	Multi-threaded WASM requires server headers (COOP/COEP) and browser support
Clipboard	Requires user gesture; limited formats
Notifications	Use Web Notifications API via JS interop
Storage	LocalStorage/IndexedDB via JS interop for persistence

Design for progressive enhancement: provide alternative flows if feature unsupported.

13. Practice exercises

1. Add a browser head and run the app in Chrome/Firefox, verifying rendering fallbacks.
2. Implement file export via `IStorageProvider` and test save polyfill with service worker enabled/disabled.
3. Add logging to report `BrowserPlatformOptions.RenderingMode` and `ActualTransparencyLevel` (should be `None`).
4. Integrate a JavaScript API (e.g., Web Notifications) via interop and show a notification after user action.
5. Publish a release build and deploy to a static host (GitHub Pages or local web server), verifying service worker scope.

Look under the hood (source bookmarks)

- Browser app builder: `BrowserAppBuilder.cs`
- Browser lifetime: `BrowserSingleViewLifetime.cs`
- Browser storage provider: `BrowserStorageProvider.cs`
- Input pane & insets: `BrowserInputPane.cs`, `BrowserInsetsManager.cs`
- Blazor integration: `Avalonia.Browser.Blazor`

Check yourself

- How do you configure rendering fallbacks for the browser target?
- What limitations exist for file access and how does the polyfill help?
- Which headers or hosting requirements enable WASM multi-threading? Why might you set `PreferManagedThreadDispatcher`?
- How do CORS rules affect HttpClient calls in the browser?
- What deployment steps are required to serve a browser bundle with service worker support?

What's next - Next: Chapter 21

21. Headless and testing

Goal - Test Avalonia UI components without a display server using the headless platform. - Simulate user input, capture rendered frames, and integrate UI tests into CI (xUnit, NUnit, other frameworks). - Organize your test strategy: view models, control-level tests, visual regression, fast feedback.

Why this matters - UI you can't test will regress. Headless testing runs anywhere (CI, Docker) and stays deterministic. - Automated UI tests catch regressions in bindings, styles, commands, and layout quickly.

Prerequisites - Chapter 11 (MVVM patterns), Chapter 17 (async patterns), Chapter 16 (storage) for file-based assertions.

1. Packages and setup

Add packages to your test project: - `Avalonia.Headless` - `Avalonia.Headless.XUnit` or `Avalonia.Headless.NUnit` - `Avalonia.Skia` (only if you need rendered frames)

xUnit setup (`AssemblyInfo.cs`)

```
using Avalonia;
using Avalonia.Headless;
using Avalonia.Headless.XUnit;

[assembly: AvaloniaTestApplication(typeof(TestApp))]

public sealed class TestApp : Application
{
    public static AppBuilder BuildAvaloniaApp() => AppBuilder.Configure<TestApp>()
        .UseHeadless(new AvaloniaHeadlessPlatformOptions
        {
            UseHeadlessDrawing = true, // set false + UseSkia for frame capture
            UseCpuDisabledRenderLoop = true
        })
        .AfterSetup(_ => Dispatcher.UIThread.VerifyAccess());
}
```

`UseHeadlessDrawing = true` skips Skia (fast). For pixel tests, set false and call `.UseSkia()`.

NUnit setup

Use `[AvaloniaTestApp]` attribute (from `Avalonia.Headless.NUnit`) and the provided `AvaloniaTestFixture` base.

2. Writing a simple headless test

```
public class TextBoxTests
{
    [AvaloniaFact]
    public async Task TextBox_Receives_Typed_Text()
    {
        var textBox = new TextBox { Width = 200, Height = 24 };
        var window = new Window { Content = textBox };
        window.Show();

        // Focus on UI thread
        await Dispatcher.UIThread.InvokeAsync(() => textBox.Focus());
```

```
        window.KeyTextInput("Avalonia");
        AvaloniaHeadlessPlatform.ForceRenderTimerTick();

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

Helpers from `Avalonia.Headless` add extension methods to `TopLevel`/`Window` (`KeyTextInput`, `KeyPress`, `MouseDown`, etc.). Always call `ForceRenderTimerTick()` after inputs to flush layout/bindings.

3. Simulating pointer input

```
[ AvaloniaFact ]
public async Task Button_Click_Executes_Command()
{
    var commandExecuted = false;
    var button = new Button
    {
        Width = 100,
        Height = 30,
        Content = "Click me",
        Command = ReactiveCommand.Create(() => commandExecuted = true)
    };

    var window = new Window { Content = button };
    window.Show();

    await Dispatcher.UIThread.InvokeAsync(() => button.Focus());
    window.MouseDown(button.Bounds.Center, MouseButton.Left);
    window.MouseUp(button.Bounds.Center, MouseButton.Left);
    AvaloniaHeadlessPlatform.ForceRenderTimerTick();

    Assert.True(commandExecuted);
}
```

`Bounds.Center` obtains center point from `Control.Bounds`. For container-based coordinates, offset appropriately.

4. Frame capture & visual regression

Configure Skia rendering in test app builder:

```
public static AppBuilder BuildAvaloniaApp() => AppBuilder.Configure<TestApp>()
    .UseSkia()
    .UseHeadless(new AvaloniaHeadlessPlatformOptions
    {
        UseHeadlessDrawing = false,
        UseCpuDisabledRenderLoop = true
    });
```

Capture frames:

```
[ AvaloniaFact ]
public void Border_Renders_Correct_Size()
{
    var border = new Border
    {
```

```
        Width = 200,
        Height = 100,
        Background = Brushes.Red
    };

    var window = new Window { Content = border };
    window.Show();
    AvaloniaHeadlessPlatform.ForceRenderTimerTick();

    using var frame = window.GetLastRenderedFrame();
    Assert.Equal(200, frame.Size.Width);
    Assert.Equal(100, frame.Size.Height);

    // Optional: save to disk for debugging
    // frame.Save("border.png");
}
```

Compare pixels to baseline image using e.g., `ImageMagick` or custom diff with tolerance. Keep baselines per theme/resolution to avoid false positives.

5. Organizing tests

- **ViewModel tests**: no Avalonia dependencies; test commands and property changes (fastest).
- **Control tests**: headless platform; simulate inputs to verify states.
- **Visual regression**: limited number; capture frames and compare.
- **Integration/E2E**: run full app with navigation; keep few due to complexity.

6. Advanced headless scenarios

6.1 VNC mode

For debugging, you can run headless with a VNC server and observe the UI.

```
AppBuilder.Configure<App>()
    .UseHeadless(new AvaloniaHeadlessPlatformOptions { UseVnc = true, UseSkia = true })
    .StartWithClassicDesktopLifetime(args);
```

Connect with a VNC client to view frames and interact.

6.2 Simulating time & timers

Use `AvaloniaHeadlessPlatform.ForceRenderTimerTick()` to advance timers. For `DispatcherTimer` or animations, call it repeatedly.

6.3 File system in tests

For file-based assertions, use in-memory streams or temp directories. Avoid writing to the repo path; tests should be self-cleaning.

7. Testing async flows

- Use `Dispatcher.UIThread.InvokeAsync` for UI updates.
- Await tasks; avoid `.Result` or `.Wait()`.
- To wait for state changes, poll with timeout:

```
async Task WaitForAsync(Func<bool> condition, TimeSpan timeout)
{
```

```
    var deadline = DateTime.UtcNow + timeout;
    while (!condition())
    {
        if (DateTime.UtcNow > deadline)
            throw new TimeoutException("Condition not met");
        AvaloniaHeadlessPlatform.ForceRenderTimerTick();
        await Task.Delay(10);
    }
}
```

8. CI integration

- Headless tests run under `dotnet test` in GitHub Actions/Azure Pipelines/GitLab.
- On Linux CI, no display server required (no `Xvfb`).
- Provide environment variables or test-specific configuration as needed.
- Collect snapshots as build artifacts when tests fail (optional).

9. Practice exercises

1. Write a headless test that types into a TextBox, presses Enter, and asserts a command executed.
2. Simulate a drag-and-drop using `DragDrop` helpers and confirm target list received data.
3. Capture a frame of an entire form and compare to a baseline image stored under `tests/BaselineImages`.
4. Create a test fixture that launches the app's main view, navigates to a secondary page, and verifies a label text.
5. Add headless tests to CI and configure the pipeline to upload snapshot diffs for failing cases.

Look under the hood (source bookmarks)

- Headless platform: `AvaloniaHeadlessPlatform`
- Input extensions: `HeadlessWindowExtensions`
- xUnit integration: `Avalonia.Headless.XUnit`
- NUnit integration: `Avalonia.Headless.NUnit`
- Reference tests: `tests/Avalonia.Headless.UnitTests`

Check yourself

- How do you initialize the headless platform for xUnit? Which attribute is required?
- How do you simulate keyboard and pointer input in headless tests?
- What steps are needed to capture rendered frames? Why might you use them sparingly?
- How can you run the headless platform visually (e.g., via VNC) for debugging?
- How does your test strategy balance view model tests, control tests, and visual regression tests?

What's next - Next: Chapter 22

22. Rendering pipeline in plain words

Goal - Understand how Avalonia turns your visual tree into frames on screen across platforms. - Know the responsibilities of the UI thread, render thread, compositor, renderer, and Skia. - Learn how to tune rendering with `SkiaOptions`, `RenderOptions`, and diagnostics tools.

Why this matters - Smooth, power-efficient UI depends on understanding what triggers redraws and how Avalonia schedules work. - Debugging rendering glitches is easier when you know each component's role.

Prerequisites - Chapter 17 (async/background) for thread awareness, Chapter 18/19 (platform differences).

1. Mental model

1. **UI thread** builds and updates the visual tree (`Visuals`/`Controls`). When properties change, visuals mark themselves dirty (e.g., via `InvalidateVisual`).
2. **Compositor** batches dirty visuals, serializes changes, and schedules a render pass.
3. **Renderer** walks the visual tree, issues drawing commands, and hands them to Skia.
4. **Skia** rasterizes shapes/text/images into GPU textures (or CPU bitmaps).
5. **Platform swapchain** presents the frame in a window or surface.

Avalonia uses a multithreaded architecture: UI thread and render thread. Animation scheduling, input handling, and compositing rely on the UI thread staying responsive.

2. UI thread: creating and invalidating visuals

- `Visuals` have properties (`Bounds`, `Opacity`, `Transform`, etc.) that trigger redraw when changed.
- `InvalidateVisual()` marks a visual dirty. Most controls call this automatically when a property changes.
- Layout changes may also mark visuals dirty (e.g., size change).

3. Render thread and renderer pipeline

- `IRenderer` (see `IRenderer.cs`) exposes methods:
  - `AddDirty(Visual visual)` – mark dirty region.
  - `Paint` – handle paint request (e.g., OS says "redraw now").
  - `Resized` – update when target size changes.
  - `Start`/`Stop` – hook into render loop lifetime.

Avalonia includes `CompositingRenderer` (default) and `DeferredRenderer`. The renderer uses dirty rectangles to redraw minimal regions.

Immediate renderer

`ImmediateRenderer` renders a visual subtree synchronously into a `DrawingContext`. Used for `RenderTargetBitmap`, `VisualBrush`, etc. Not used for normal window presentation.

4. Compositor and render loop

The compositor orchestrates UI -> render thread updates (see `Compositor.cs`).

- Batches (serialized UI tree updates) are committed to render thread.
- `RenderLoop` ticks at platform-defined cadence (vsync/animation timers). When there's dirty content or `CompositionTarget` animations, it schedules a frame.
- Render loop ensures frames draw at stable cadence even if UI thread is busy momentarily.

5. Skia backend

Avalonia uses Skia for cross-platform drawing: - GPU or CPU rendering depending on platform capabilities. - GPU backend chosen automatically (OpenGL, ANGLE, Metal, Vulkan, WebGL, etc.). - `UseSkia(new SkiaOptions { ... })` in `AppBuilder` to tune.

SkiaOptions

```
AppBuilder.Configure<App>()
    .UsePlatformDetect()
    .UseSkia(new SkiaOptions
    {
        MaxGpuResourceSizeBytes = 64L * 1024 * 1024,
        UseOpacitySaveLayer = false
    })
    .LogToTrace();
```

- `MaxGpuResourceSizeBytes`: limit Skia resource cache.
- `UseOpacitySaveLayer`: forces Skia to use save layers for opacity stacking (accuracy vs performance).

6. RenderOptions (per Visual)

`RenderOptions` attached properties influence interpolation and text rendering: - `BitmapInterpolationMode`: Low/Medium/High quality vs default. - `BitmapBlendingMode`: blend mode for images. - `TextRenderingMode`: Default, Antialias, SubpixelAntialias, Aliased. - `EdgeMode`: Antialias vs Aliased for geometry edges. - `RequiresFullOpacityHandling`: handle complex opacity composition.

Example:

```
RenderOptions.SetBitmapInterpolationMode(image, BitmapInterpolationMode.HighQuality);
RenderOptions.SetTextRenderingMode(smallText, TextRenderingMode.Aliased);
```

RenderOptions apply to a visual and flow down to children unless overridden.

7. When does a frame render?

- Property changes on visuals (brush, text, transform).
- Layout updates affecting size/position.
- Animations (composition or binding-driven) schedule continuous frames.
- Input (pointer events) may cause immediate redraw (e.g., ripple effect).
- External events: window resize, DPI change.

Prevent unnecessary redraws: - Avoid toggling properties frequently without change. - Batch updates on UI thread; let binding/animation handle smooth changes. - Free large bitmaps once no longer needed.

8. Profiling & diagnostics

DevTools

- Press `F12` to open DevTools.
- Use `Rendering` panel (if available) to inspect GPU usage, show dirty rectangles.
- `Visual Tree` shows realized visuals; `Events` logs layout/render events.

Logging

```
AppBuilder.Configure<App>()
    .UsePlatformDetect()
    .LogToTrace(LogEventLevel.Debug, new[] { LogArea.Rendering, LogArea.Layout })
    .StartWithClassicDesktopLifetime(args);
```

Render overlays

`RendererDebugOverlays` (see `RendererDebugOverlays.cs`) enable overlays showing dirty rectangles, FPS, layout costs.

```
if (TopLevel is { Renderer: { } renderer })
    renderer.DebugOverlays = RendererDebugOverlays.Fps | RendererDebugOverlays.LayoutTimeGraph;
```

Tools

- Use .NET memory profiler or `dotnet-counters` to monitor GC while animating UI.
- GPU profilers (RenderDoc) can capture Skia GPU commands (advanced scenario).

9. Immediate rendering utilities

RenderTargetBitmap

```
var bitmap = new RenderTargetBitmap(new PixelSize(300, 200), new Vector(96, 96));
await bitmap.RenderAsync(myControl);
bitmap.Save("snapshot.png");
```

Uses `ImmediateRenderer` to render a control off-screen.

Drawing manually

`DrawingContext` allows custom drawing via immediate renderer.

10. Platform-specific notes

- Windows: GPU backend typically ANGLE (OpenGL) or D3D via Skia; transparency support (Mica/Acrylic) may involve compositor-level effects.
- macOS: uses Metal via Skia; retina scaling via `RenderScaling`.
- Linux: OpenGL (or Vulkan) depending on driver; virtualization/backends vary.
- Mobile: OpenGL ES on Android, Metal on iOS; consider battery impact when scheduling animations.
- Browser: WebGL2/WebGL1/Software2D (Chapter 20); one-threaded unless WASM threading enabled.

11. Practice exercises

1. Enable `RendererDebugOverlays.Fps` and animate a control; observe frame rate.
2. Switch `BitmapInterpolationMode` on an image while scaling up/down; compare results.
3. Apply `UseOpacitySaveLayer = true` and stack semi-transparent panels; compare visual results to default.
4. Render a control to `RenderTargetBitmap` using `RenderOptions` tweaks and inspect output.
5. Log render/layout events at `Debug` level and analyze which updates cause frames using DevTools.

Look under the hood (source bookmarks)

- Renderer interface: `IRenderer.cs`
- Compositor: `Compositor.cs`
- Immediate renderer: `ImmediateRenderer.cs`
- Render loop: `RenderLoop.cs`
- Render options: `RenderOptions.cs`
- Skia options and platform interface: `SkiaOptions.cs`, `PlatformRenderInterface.cs`
- Debug overlays: `RendererDebugOverlays.cs`

Check yourself

- What components run on the UI thread vs render thread?
- How does `InvalidateVisual` lead to a new frame?
- When would you adjust `SkiaOptions.MaxGpuResourceSizeBytes` vs `RenderOptions.BitmapInterpolationMode`?
- What tools help you diagnose rendering bottlenecks?

What's next - Next: Chapter 23

23. Custom drawing and custom controls

Goal - Decide when to custom draw (override `Render`) versus build templated controls (pure XAML). - Master `DrawingContext`, invalidation (`AffectsRender`, `InvalidateVisual`), and caching for performance. - Structure a restylable `TemplatedControl`, expose properties, and support theming/accessibility.

Why this matters - Charts, gauges, and other visuals often need custom drawing. Understanding rendering and templating keeps your controls fast and customizable. - Well-structured controls enable reuse and consistent theming.

Prerequisites - Chapter 22 (rendering pipeline), Chapter 15 (accessibility), Chapter 16 (storage for exporting images if needed).

1. Choosing an approach

Scenario	Draw (override `Render`)	Template (`ControlTemplate`)
Pixel-perfect graphics, charts	[x]	
Animations driven by drawing primitives	[x]	
Standard widgets composed of existing controls		[x]
Consumer needs to restyle via XAML		[x]
Complex interaction per element (buttons in control)		[x]

Hybrid: templated control containing a custom-drawn child for performance-critical surface.

2. Invalidation basics

- `InvalidateVisual()` schedules redraw.
- Register property changes via `AffectsRender<TControl>(property1, ...)` in static constructor to auto-invalidate on property change.
- For layout changes, use `InvalidateMeasure` similarly (handled automatically for `StyledPropertys` registered with `AffectsMeasure`).

3. DrawingContext essentials

`DrawingContext` primitives: - `DrawGeometry(brush, pen, geometry)` - `DrawRectangle`/`DrawEllipse` - `DrawImage(image, sourceRect, destRect)` - `DrawText(formattedText, origin)` - `PushClip`, `PushOpacity`, `PushOpacityMask`, `PushTransform` – use in `using` blocks to auto-pop state.

Example pattern:

```
public override void Render(DrawingContext ctx)
{
    base.Render(ctx);
    using (ctx.PushClip(new Rect(Bounds.Size)))
    {
        ctx.DrawRectangle(Brushes.Black, null, Bounds);
        ctx.DrawText(_formattedText, new Point(10, 10));
    }
}
```

4. Example: Sparkline (custom draw)

```
public sealed class Sparkline : Control
{
    public static readonly StyledProperty<IReadOnlyList<double>?> ValuesProperty =
```

```
    AvaloniaProperty.Register<Sparkline, IReadOnlyList<double>?>(nameof(Values));

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

public static readonly StyledProperty<double> StrokeThicknessProperty =
    AvaloniaProperty.Register<Sparkline, double>(nameof(StrokeThickness), 2.0);

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

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

public IBrush Stroke
{
    get => GetValue(StrokeProperty);
    set => SetValue(StrokeProperty, value);
}

public double StrokeThickness
{
    get => GetValue(StrokeThicknessProperty);
    set => SetValue(StrokeThicknessProperty, value);
}

public override void Render(DrawingContext ctx)
{
    base.Render(ctx);
    var values = Values;
    var bounds = Bounds;
    if (values is null || values.Count < 2 || bounds.Width <= 0 || bounds.Height <= 0)
        return;

    double min = values.Min();
    double max = values.Max();
    double range = Math.Max(1e-9, max - min);

    using var geometry = new StreamGeometry();
    using (var gctx = geometry.Open())
    {
        for (int i = 0; i < values.Count; i++)
        {
            double t = i / (double)(values.Count - 1);
            double x = bounds.X + t * bounds.Width;
            double yNorm = (values[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(false);
        }

        var pen = new Pen(Stroke, StrokeThickness);
        ctx.DrawGeometry(null, pen, geometry);
    }
}
```

Usage:

```
<local:Sparkline Width="160" Height="36" Values="3,7,4,8,12" StrokeThickness="2"/>
```

Performance tips

- Avoid allocations inside `Render`. Cache `Pen`, `FormattedText` when possible.
- Use `StreamGeometry` and reuse if values rarely change (rebuild when invalidated).

5. Templated control example: Badge

Create `Badge : TemplatedControl` with properties (`Content`, `Background`, `Foreground`, `CornerRadius`, `MaxWidth`, etc.). Default style in `Styles.axaml`:

```
<Style Selector="local|Badge">
  <Setter Property="Template">
    <ControlTemplate TargetType="local:Badge">
      <Border Background="{TemplateBinding Background}"
              CornerRadius="{TemplateBinding CornerRadius}"
              Padding="6,0"
              MinHeight="16" MinWidth="20"
              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="12"/>
  <Setter Property="HorizontalAlignment" Value="Left"/>
</Style>
```

Consumers can override the template for custom visuals without editing C#.

Control class

```
public sealed class Badge : TemplatedControl
{
    public static readonly StyledProperty<object?> ContentProperty =
        AvaloniaProperty.Register<Badge, object?>(nameof(Content));
```

```
    public object? Content
    {
        get => GetValue(ContentProperty);
        set => SetValue(ContentProperty, value);
    }
}
```

Additional properties (e.g., `CornerRadius`, `Background`) are inherited from `TemplatedControl` base properties or newly registered as needed.

6. Accessibility & input

- Set `Focusable` as appropriate; override `OnPointerPressed`/`OnKeyDown` for interaction.
- Expose automation metadata via `AutomationProperties.Name`, `HelpText`, or custom `AutomationPeer` for drawn controls.
- Implement `OnCreateAutomationPeer` when your control represents a unique semantic (`ProgressBadgeAutomationPeer`

7. Measure/arrange

Custom controls should override `MeasureOverride`/`ArrangeOverride` when size depends on content/drawing.

```
protected override Size MeasureOverride(Size availableSize)
{
    var values = Values;
    if (values is null || values.Count == 0)
        return Size.Empty;
    return new Size(Math.Min(availableSize.Width, 120), Math.Min(availableSize.Height, 36));
}
```

`TemplatedControl` handles measurement via its template (border + content). For custom-drawn controls, define desired size heuristics.

8. Rendering to bitmaps / exporting

Use `RenderTargetBitmap` for saving custom visuals:

```
var rtb = new RenderTargetBitmap(new PixelSize(200, 100), new Vector(96, 96));
await rtb.RenderAsync(sparkline);
await using var stream = File.OpenWrite("spark.png");
await rtb.SaveAsync(stream);
```

Use `RenderOptions` to adjust interpolation for exported graphics if needed.

9. Combining drawing & template (hybrid)

Example: `ChartControl` template contains toolbar (Buttons, ComboBox) and a custom `ChartCanvas` child that handles drawing/selection. - Template XAML composes layout. - Drawn child handles heavy rendering & direct pointer handling. - Chart exposes data/selection via view models.

10. Troubleshooting & best practices

- Flickering or wrong clip: ensure you clip to `Bounds` using `PushClip` when necessary.
- Aliasing issues: adjust `RenderOptions.SetEdgeMode` and align lines to device pixels (e.g., `Math.Round(x) + 0.5` for 1px strokes at 1.0 scale).
- Performance: profile by measuring allocations, consider caching `StreamGeometry`/`FormattedText`.

- Template issues: ensure template names line up with `TemplateBinding`; use DevTools -> `Style Inspector` to check which template applies.

11. Practice exercises

1. Build a `BarGauge` control: custom draw N vertical bars, exposing properties for values/brushes/thickness.
2. Create a `Badge` templated control with alternative styles (e.g., success/warning) using style classes.
3. Add an accessibility peer for `Sparkline` that reports summary (min/max/average) via `AutomationProperties.HelpTex`
4. Export your custom drawing to a PNG using `RenderTargetBitmap` and verify output at multiple DPI.

Look under the hood (source bookmarks)

- Visual/render infrastructure: `Visual.cs`
- DrawingContext API: `DrawingContext.cs`
- StreamGeometry: `StreamGeometryContextImpl`
- Templated control base: `TemplatedControl.cs`
- Control theme infrastructure: `ControlTheme.cs`
- Automation peers: `ControlAutomationPeer.cs`

Check yourself

- When do you override `Render` versus `ControlTemplate`?
- How does `AffectsRender` simplify invalidation?
- What caches can you introduce to prevent allocations in `Render`?
- How do you expose accessibility information for drawn controls?
- How can consumers restyle your templated control without touching C#?

What's next - Next: Chapter 24

24. Performance, diagnostics, and DevTools

Goal - Diagnose and fix Avalonia performance issues using measurement, logging, DevTools, and overlays. - Focus on the usual suspects: non-virtualized lists, layout churn, binding storms, expensive rendering. - Build repeatable measurement habits (Release builds, small reproducible tests).

Why this matters - "UI feels slow" is common feedback. Without data, fixes are guesswork. - Avalonia provides built-in diagnostics (DevTools, overlays) and logging hooks–learn to leverage them.

Prerequisites - Chapter 22 (rendering pipeline), Chapter 17 (async patterns), Chapter 16 (custom controls and lists).

1. Measure before changing anything

- Run in Release (`dotnet run -c Release`). JIT optimizations affect responsiveness.
- Use a small repro: isolate the view or control and reproduce with minimal data before optimizing.
- Use high-resolution timers only around suspect code sections; avoid timing entire app startup on the first pass.
- Change one variable at a time and re-measure to confirm impact.

2. Logging

Enable logging per area using `AppBuilder` extensions (see `LoggingExtensions.cs`).

```
AppBuilder.Configure<App>()
    .UsePlatformDetect()
    .LogToTrace(LogEventLevel.Information, new[] { LogArea.Binding, LogArea.Layout, LogArea.Render, Log
    .StartWithClassicDesktopLifetime(args);
```

- Areas: see `Avalonia.Logging.LogArea` (`Binding`, `Layout`, `Render`, `Property`, `Control`, etc.).
- Reduce noise by lowering level (`Warning`) or limiting areas once you identify culprit.
- Optionally log to file via `LogToTextWriter`.

3. DevTools (F12)

Attach DevTools after app initialization:

```
public override void OnFrameworkInitializationCompleted()
{
    // configure windows/root view
    this.AttachDevTools();
    base.OnFrameworkInitializationCompleted();
}
```

Supports options: `AttachDevTools(new DevToolsOptions { StartupScreenIndex = 1 })` for multi-monitor setups.

DevTools tour

- **Visual Tree**: inspect hierarchy, properties, pseudo-classes, and layout bounds.
- **Logical Tree**: understand DataContext/template relationships.
- **Layout Explorer**: measure/arrange info, constraints, actual sizes.
- **Events**: view event flow; detect repeated pointer/keyboard events.
- **Styles & Resources**: view applied styles/resources; test pseudo-class states.
- **Hotkeys/Settings**: adjust F12 gesture.

Use the target picker to select elements on screen and inspect descendants/ancestors.

4. Debug overlays (`RendererDebugOverlays`)

Access via DevTools “Diagnostics” pane or programmatically:

```
if (this.ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
{
    desktop.MainWindow.AttachedToVisualTree += (_, __) =>
    {
        if (desktop.MainWindow?.Renderer is { } renderer)
            renderer.DebugOverlays = RendererDebugOverlays.Fps | RendererDebugOverlays.DirtyRects;
    };
}
```

Overlays include: - `Fps` – frames per second. - `DirtyRects` – regions redrawn each frame. - `LayoutTimeGraph` – layout duration per frame. - `RenderTimeGraph` – render duration per frame.

Interpretation: - Large dirty rects = huge redraw areas; find what invalidates entire window. - LayoutTime spikes = heavy measure/arrange; check Layout Explorer to spot bottleneck. - RenderTime spikes = expensive drawing (big bitmaps, custom rendering).

5. Performance checklist

Lists & templates - Use virtualization (`VirtualizingStackPanel`) for list controls. - Keep item templates light; avoid nested panels and convert heavy converters to cached data. - Pre-compute value strings/colors in view models to avoid per-frame conversion.

Layout & binding - Minimize property changes that re-trigger layout of large trees. - Avoid swapping entire templates when simple property changes suffice. - Watch for binding storms (log `LogArea.Binding`). Debounce or use state flags.

Rendering - Use vector assets where possible; for bitmaps, match display resolution. - Set `RenderOptions.BitmapInterpolati` for scaling to avoid blurry or overly expensive scaling. - Cache expensive geometries (`StreamGeometry`), `FormattedText`, etc.

Async & threading - Move heavy work off UI thread (async/await, `Task.Run` for CPU-bound tasks). - Use `IProgress<T>` to report progress instead of manual UI thread dispatch.

Profiling - Use `.NET` profilers (dotTrace, PerfView, dotnet-trace) to capture CPU/memory. - For GPU, use platform tools if necessary (RenderDoc for GL/DirectX when supported).

6. Considerations per platform

- Windows: ensure GPU acceleration enabled; check drivers. Acrylic/Mica can cost extra GPU time.
- macOS: retina scaling multiplies pixel counts; ensure vector assets and efficient drawing.
- Linux: varying window managers/compositors. If using software rendering, expect lower FPS–optimize accordingly.
- Mobile & Browser: treat CPU/GPU resources as more limited; avoid constant redraw loops.

7. Automation & CI

- Combine unit tests with headless UI tests (Chapter 21).
- Create regression tests for performance-critical features (measure time for known operations, fail if above threshold).
- Capture baseline metrics (FPS, load time) and compare across commits; tools like BenchmarkDotNet can help (for logic-level measurements).

8. Workflow summary

1. Reproduce in Release with logging disabled -> measure baseline.
2. Enable DevTools overlays (FPS, dirty rects, layout/render graphs) -> identify pattern.
3. Enable targeted logging (Binding/Layout/Render) -> correlate with overlays.
4. Apply fix (virtualization, caching, reducing layout churn)
5. Re-measure with overlays/logs to confirm improvements.
6. Capture notes and, if beneficial, automate tests for future regressions.

9. Practice exercises

1. Attach DevTools to your app, enable `RendererDebugOverlays.Fps`, and record FPS before/after virtualizing a long list.
2. Log `Binding/Property` areas and identify recurring property changes; batch or throttle updates.
3. Measure layout time via overlay before/after simplifying a nested panel layout; compare results.
4. Add a unit/UITest that asserts a time-bound operation completes under a threshold (e.g., load 1,000 items). Use Release build to verify.
5. Capture a profile with `dotnet-trace` or `dotnet-counters` during a slow interaction; interpret CPU/memory graphs.

Look under the hood (source bookmarks)

- DevTools attach helpers: `DevToolsExtensions.cs`
- DevTools view models (toggling overlays): `MainViewModel.cs`
- Renderer overlays: `RendererDebugOverlays.cs`
- Logging infrastructure: `LogArea`
- RenderOptions (quality settings): `RenderOptions.cs`
- Layout diagnostics: `LayoutHelper`

Check yourself

- Why must performance measurements be done in Release builds?
- Which overlay would you enable to track layout time spikes? What about render time spikes?
- How do DevTools and logging complement each other?
- List three common causes of UI lag and their fixes.
- How would you automate detection of a performance regression?

What's next - Next: Chapter 25

25. Design-time tooling and the XAML Previewer

Goal - Use Avalonia's XAML Previewer (designer) effectively in VS, Rider, and VS Code. - Feed realistic sample data and preview styles/resources without running your full backend. - Understand design mode plumbing, avoid previewer crashes, and sharpen your design workflow.

Why this matters - Fast iteration on UI keeps you productive. The previewer drastically reduces build/run cycles if you set it up correctly. - Design-time data prevents "black boxes" in the previewer and reveals layout problems early.

Prerequisites - Familiarity with XAML bindings (Chapter 8) and templates (Chapter 23).

1. How the previewer works

IDE hosts spawn a preview process that loads your view or resource dictionary. Avalonia signals design mode via `Design.IsDesignMode` and applies design-time properties (`Design.*`).

Key components (see `Design.cs`): - `Design.IsDesignMode`: true inside previewer; branch code to avoid real services. - `Design.DataContext`, `Design.Width/Height`, `Design.DesignStyle`, `Design.PreviewWith`: attached properties injected at design time and removed from runtime. - XAML transformer (`AvaloniaXamlIlDesignPropertiesTransformer`) strips `Design.*` in compiled output.

2. Design-time DataContext & sample data

Provide lightweight POCOs or design view models for preview.

Sample POCO:

```
namespace MyApp.Design;

public sealed class SamplePerson
{
    public string Name { get; set; } = "Ada Lovelace";
    public string Email { get; set; } = "ada@example.com";
    public int Age { get; set; } = 37;
}
```

Usage in XAML:

```
<UserControl xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:design="clr-namespace:Avalonia.Controls;assembly=Avalonia.Controls"
             xmlns:samples="clr-namespace:MyApp.Design" x:Class="MyApp.Views.ProfileView">
  <design:Design.DataContext>
    <samples:SamplePerson/>
  </design:Design.DataContext>

  <StackPanel Spacing="12" Margin="16">
    <TextBlock Classes="h1" Text="{Binding Name}"/>
    <TextBlock Text="{Binding Email}"/>
    <TextBlock Text="Age: {Binding Age}"/>
  </StackPanel>
</UserControl>
```

At runtime the transformer removes `Design.DataContext`; real view models take over. For complex forms, expose design view models with stub services but avoid heavy logic.

Design.IsDesignMode checks

Guard expensive operations:

```
if (Design.IsDesignMode)
    return; // skip service setup, timers, network
```

Place guards in view constructors, `OnApplyTemplate`, or view model initialization.

3. Design.Width/Height & DesignStyle

Set design canvas size:

```
<StackPanel design:Design.Width="320"
            design:Design.Height="480"
            design:Design.DesignStyle="{StaticResource DesignOutlineStyle}">

</StackPanel>
```

`DesignStyle` can add dashed borders or backgrounds for preview only (define style in resources).

Example design style:

```
<Style x:Key="DesignOutlineStyle">
  <Setter Property="Border.BorderThickness" Value="1"/>
  <Setter Property="Border.BorderBrush" Value="#808080"/>
</Style>
```

4. Preview resource dictionaries with Design.PreviewWith

Previewing a dictionary or style requires a host control:

```
<ResourceDictionary xmlns="https://github.com/avaloniaui"
                    xmlns:design="clr-namespace:Avalonia.Controls;assembly=Avalonia.Controls"
                    xmlns:views="clr-namespace:MyApp.Views">
  <design:Design.PreviewWith>
    <Border Padding="16" Background="#1f2937">
      <StackPanel Spacing="8">
        <views:Badge Content="1" Classes="success"/>
        <views:Badge Content="Warning" Classes="warning"/>
      </StackPanel>
    </Border>
  </design:Design.PreviewWith>


</ResourceDictionary>
```

`PreviewWith` ensures the previewer renders the host when you open the dictionary alone.

5. IDE-specific tips

Visual Studio

- Ensure “Avalonia Previewer” extension is installed.
- F12 toggles DevTools; `Alt+Space` opens previewer hotkeys.
- If previewer doesn’t refresh, rebuild project; VS sometimes caches the design assembly.

Rider

- Avalonia plugin required; previewer window shows automatically when editing XAML.
- Use the data context drop-down to quickly switch between sample contexts if multiple available.

VS Code

- Avalonia `.vsix` extension supports previewer with dotnet CLI driven host. Ensure `dotnet workload install wasm-tools` (previewer uses WASM).

General - Keep constructors light; heavy constructors crash previewer. - Use `Design.DataContext` to avoid hitting DI container or real services. - Split complex layouts into smaller user controls and preview them individually.

6. Troubleshooting & best practices

Issue	Fix
Previewer blank/crashes	Guard code with `Design.IsDesignMode`; simplify layout; ensure no blocking calls in constructor
Design-only styles appear at runtime	Remember `Design.*` stripped at runtime; if you see them, check build output or ensure property wired correctly
Resource dictionary preview fails	Add `Design.PreviewWith`; ensure resources compiled (check `AvaloniaResource` includes)
Sample data not showing	Confirm namespace mapping correct and sample object constructs without exceptions
Slow preview	Remove animations/effects temporarily; large data sets or virtualization can slow preview host

7. Automation

- Document designer defaults using `README` for your UI project. Include instructions for sample data.&
- Use git hooks/CI to catch accidental runtime usages of `Design.*`. For instance, forbid `Design.IsDesignMode` checks in release-critical code by scanning for patterns if needed.

8. Practice exercises

1. Add `Design.DataContext` to a complex form, providing realistic sample data (names, email, totals). Ensure preview shows formatted values.
2. Set `Design.Width/Height` to 360x720 for a mobile view; use `Design.DesignStyle` to highlight layout boundaries.
3. Create a resource dictionary for badges; use `Design.PreviewWith` to render multiple badge variants side-by-side.
4. Guard service initialization with `if (Design.IsDesignMode)` and confirm preview load improves.
5. Bonus: create a `Design` namespace helper static class that exposes sample models for multiple views; reference it from XAML.

Look under the hood (source bookmarks)

- Design property helpers: `Design.cs`
- Previewer bootstrapping: `RemoteDesignerEntryPoint.cs`
- Design-time property transformer: `AvaloniaXamlIlDesignPropertiesTransformer.cs`
- Previewer window implementation: `PreviewerWindowImpl.cs`
- Samples: ControlCatalog resources demonstrate `Design.PreviewWith` usage (`samples/ControlCatalog/Styles/...`)

Check yourself

- How do you provide sample data without running production services?
- How do you prevent design-only code from running in production?
- When do you use `Design.PreviewWith`?
- What are the most common previewer crashes and how do you avoid them?

What's next - Next: Chapter 26

26. Build, publish, and deploy

Goal - Produce distributable builds for every platform Avalonia supports (desktop, mobile, browser). - Understand .NET publish options (framework-dependent vs self-contained, single-file, ReadyToRun, trimming). - Package and ship your app (MSIX, DMG, AppImage, AAB/IPA, browser bundles) and automate via CI/CD.

Why this matters - Reliable builds avoid "works on my machine" syndrome. - Choosing the right publish options balances size, startup time, and compatibility.

Prerequisites - Chapters 18-20 for platform nuances, Chapter 17 for async/networking (relevant to release builds).

1. Build vs publish

- `dotnet build`: compiles assemblies, typically run for local development.
- `dotnet publish`: creates a self-contained folder/app ready to run on target machines (Optionally includes .NET runtime).
- Always test in `Release` configuration: `dotnet publish -c Release`.

2. Runtime identifiers (RIDs)

Common RIDs: - Windows: `win-x64`, `win-arm64`. - macOS: `osx-x64` (Intel), `osx-arm64` (Apple Silicon), `osx.12-arm64` (specific OS version), etc. - Linux: `linux-x64`, `linux-arm64` (distribution-neutral), or distro-specific RIDs (`linux-musl-x64`). - Android: `android-arm64`, `android-x86`, etc. (handled in platform head). - iOS: `ios-arm64`, `iossimulator-x64`. - Browser (WASM): `browser-wasm` (handled by browser head).

3. Publish configurations

Framework-dependent (requires installed .NET runtime)

```
dotnet publish -c Release -r win-x64 --self-contained false
```

Smaller download; target machine must have matching .NET runtime. Good for enterprise scenarios.

Self-contained (bundled runtime)

```
dotnet publish -c Release -r osx-arm64 --self-contained true
```

Larger download; runs on machines without .NET. Standard for consumer apps.

Single-file

```
dotnet publish -c Release -r linux-x64 /p:SelfContained=true /p:PublishSingleFile=true
```

Creates one executable (plus a few native libraries depending on platform). Avalonia may extract resources native libs to temp; test startup.

ReadyToRun

```
dotnet publish -c Release -r win-x64 /p:SelfContained=true /p:PublishReadyToRun=true
```

Precompiles IL to native code; faster cold start at cost of larger size. Measure before deciding.

Trimming (advanced)

```
dotnet publish -c Release -r osx-arm64 /p:SelfContained=true /p:PublishTrimmed=true
```

Aggressive size reduction; risky because Avalonia/XAML relies on reflection. Requires careful annotation/preservation with `DynamicDependency` or `ILLinkTrim` files. Start without trimming; enable later with thorough testing.

Publish options matrix (example)

Option	Pros	Cons
Framework-dependent	Small	Requires runtime install
Self-contained	Runs anywhere	Larger downloads
Single-file	Simple distribution	Extracts natives; more memory
ReadyToRun	Faster cold start	Larger size
Trimmed	Smaller	Risk of missing types

4. Output directories

Publish outputs to `bin/Release/<TFramework>/<RID>/publish`.

Examples: - `bin/Release/net8.0/win-x64/publish` - `bin/Release/net8.0/linux-x64/publish` - `bin/Release/net8.0/osx-arm64/publish`

Verify resources (images, fonts) present; confirm `AvaloniaResource` includes them (check `.csproj`).

5. Platform packaging

Windows

- Basic distribution: zip the publish folder or single-file EXE.
- MSIX: use `dotnet publish /p:WindowsPackageType=msix` or MSIX packaging tool. Enables automatic updates, store distribution.
- MSI/Wix: for enterprise installs.
- Code signing recommended (Authenticode certificate) to avoid SmartScreen warnings.

macOS

- Create `.app` bundle with `Avalonia.DesktopRuntime.MacOS` packaging scripts.
- Code sign and notarize: use Apple Developer ID certificate, `codesign`, `xcrun altool/notarytool`.
- Provide DMG for distribution.

Linux

- Zip/tarball publish folder with run script.
- AppImage: use `Avalonia.AppTemplate.AppImage` or AppImage tooling to bundle.
- Flatpak: create manifest (flatpak-builder). Ensure dependencies included via `org.freedesktop.Platform` runtime.
- Snap: use `snapcraft.yaml` to bundle.

Android

- Platform head (`MyApp.Android`) builds APK/AAB using Android tooling.
- Publish release AAB and sign with keystore (`./gradlew bundleRelease` or `dotnet publish` using .NET Android tooling).
- Upload to Google Play or sideload.

iOS

- Platform head (`MyApp.iOS`) builds .ipa using Xcode or `dotnet publish -f net8.0-ios -c Release` with workload.
- Requires macOS, Xcode, signing certificates, provisioning profiles.
- Deploy to App Store via Transporter/Xcode.

Browser (WASM)

- `dotnet publish -c Release` in browser head (`MyApp.Browser`). Output in `bin/Release/net8.0/browser-wasm/AppB`
- Deploy to static host (GitHub Pages, S3, etc.). Use service worker for caching if desired.

6. Automation (CI/CD)

- Use GitHub Actions/Azure Pipelines/GitLab CI to run `dotnet publish` per target.
- Example GitHub Actions matrix:

```yaml
jobs:
  publish:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        include:
          - os: windows-latest
            rid: win-x64
          - os: macos-latest
            rid: osx-arm64
          - os: ubuntu-latest
            rid: linux-x64
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-dotnet@v4
        with:
          dotnet-version: '8.0.x'
      - run: dotnet publish src/MyApp/MyApp.csproj -c Release -r ${{ matrix.rid }} --self-contained tru
      - uses: actions/upload-artifact@v4
        with:
          name: myapp-${{ matrix.rid }}
          path: src/MyApp/bin/Release/net8.0/${{ matrix.rid }}/publish
```

- Add packaging steps (MSIX, DMG) via platform-specific actions/tools.
- Sign artifacts in CI where possible (store certificates securely).

7. Verification checklist

- Run published app on real machines/VMs for each RID.
- Check fonts, DPI, plugins, network resources.
- Validate updates to config/resources; ensure relative paths work from publish folder.
- If using trimming, run automated UITests (Chapter 21) and manual smoke tests.
- Run `dotnet publish` with `--self-contained` false/true to compare sizes and startup times; pick best trade-off.

8. Troubleshooting

Problem	Fix
Missing native libs on Linux	Install required packages (`libicu`, `fontconfig`, `libx11`, etc.). Document dependencies.
Startup crash only in Release	Enable logging to file; check for missing assets; ensure `AvaloniaResource` includes.
High CPU at startup	Investigate ReadyToRun vs normal build; pre-load data asynchronously vs synchronously.
Code signing errors (macOS/Windows)	Confirm certificates, entitlements, notarization steps.
Publisher mismatch (store upload)	Align package IDs, manifest metadata with store requirements.

9. Practice exercises

1. Publish self-contained builds for `win-x64`, `osx-arm64`, `linux-x64`. Run each and note size/performance differences.
2. Enable `PublishSingleFile` and `PublishReadyToRun` for one target; compare startup time and size.
3. Experiment with trimming on a small sample; add `ILLink` attributes to preserve necessary types; test thoroughly.
4. Set up a GitHub Actions workflow to publish artifacts per RID and upload them as artifacts.
5. Optional: create MSIX (Windows) or DMG (macOS) packages and run locally to test installation/updates.

Look under the hood (source & docs)

- Avalonia build docs: `docs/build.md`
- Samples for reference packaging: `samples/ControlCatalog`
- .NET publish docs: dotnet publish reference
- App packaging: Microsoft MSIX docs, Apple code signing docs, AppImage/Flatpak/Snap guidelines.

Check yourself

- What's the difference between framework-dependent and self-contained publishes? When do you choose each?
- How do single-file, ReadyToRun, and trimming impact size/performance?
- Which RIDs are needed for your user base?
- What packaging format suits your distribution channel (installer, app store, raw executable)?
- How can CI/CD automate builds and packaging per platform?

What's next - Next: Chapter 27

27. Read the source, contribute, and grow

Goal - Navigate the Avalonia repo confidently, understand how to build/test locally, and contribute fixes, features, docs, or samples. - Step into framework sources while debugging your app, and know how to file issues or PRs effectively. - Stay engaged with the community to keep learning.

Why this matters - Framework knowledge deepens your debugging skills and shapes better app architecture. - Contributions improve the ecosystem and strengthen your expertise.

Prerequisites - Familiarity with Git, .NET tooling (`dotnet build/publish/test`).

1. Repository tour

Avalonia repo: - Core source: `src/` - `Avalonia.Base`, `Avalonia.Controls`, `Avalonia.Markup.Xaml`, `Avalonia.Diagnostics`, platform folders (`Android`, `iOS`, `Browser`, `Skia`). - Tests: `tests/` - Unit/integration/headless tests. Read tests to understand expected behavior and edge cases. - Samples: `samples/` - ControlCatalog, BindingDemo, ReactiveUIDemo, etc. Useful for debugging/regressions. - Docs: `docs/` coupled with the avalonia-docs site. - Contribution guidelines: `CONTRIBUTING.md`, `CODE_OF_CONDUCT.md`.

2. Building the framework locally

Scripts in repo root: - `build.ps1` (Windows), `build.sh` (Unix), `build.cmd`. - These restore NuGet packages, compile, run tests (optionally), and produce packages.

Manual build:

```
# Restore dependencies
dotnet restore Avalonia.sln

# Build core
cd src/Avalonia.Controls
dotnet build -c Debug

# Run tests
cd tests/Avalonia.Headless.UnitTests
dotnet test -c Release

# Run sample
cd samples/ControlCatalog
 dotnet run
```

Follow `docs/build.md` for environment requirements.

3. Reading source with purpose

Common entry points: - Controls/styling: `src/Avalonia.Controls/` (Control classes, templates, themes). - Layout: `src/Avalonia.Base/Layout/` (Measurement/arrange logic). - Rendering: `src/Avalonia.Base/Rendering/`, `src/Skia/Avalonia.Skia/`. - Input: `src/Avalonia.Base/Input/` (Pointer, keyboard, gesture recognizers).

Use IDE features (Go to Definition, Find Usages) to jump between user code and framework internals.

4. Debugging into Avalonia

- Enable symbol loading for Avalonia assemblies (packaged symbols or local build).
- In Visual Studio/Rider: enable “Allow step into external code”. Add `src` folder as source path.
- Set breakpoints in your app, step into framework code to inspect layout/renderer behavior.
- Combine with DevTools overlays to correlate visual state with code paths.

5. Filing issues

Best practice checklist: - Minimal reproducible sample (GitHub repo, .zip, or steps to recreate with ControlCatalog). - Include platform(s), .NET version, Avalonia version, self-contained vs framework-dependent. - Summarize expected vs actual behavior. Provide logs (Binding/Layout/Render) or screenshot/video when relevant. - Tag regression vs new bug; mention if release-only or debug-only.

6. Contributing pull requests

Steps: 1. Check CONTRIBUTING.md for branching/style. 2. Fork repo, create feature branch. 3. Implement change (small, focused scope). 4. Add/update tests under `tests/` (headless tests for controls, unit tests for logic). 5. Run `dotnet build` and `dotnet test` (possibly `build.ps1 -Target Test`). 6. Update docs/samples if behavior changed. 7. Submit PR with clear description, referencing issue IDs/sites. 8. Respond to feedback promptly.

Writing tests

- Use headless tests for visual/interaction behavior (Chapter 21 covers pattern).
- Add regression tests for fixed bugs to prevent future breakage.
- Consider measuring performance (BenchmarkDotNet) if change affects rendering/layout.

7. Docs & sample contributions

- Docs source: avalonia-docs repository.
  - Submit PRs with improved content/instructions/examples.
- Samples: add new sample to `samples/` illustrating advanced patterns or new controls.
- Keep docs in sync with code changes for features/bug fixes.

8. Community & learning

- GitHub discussions: AvaloniaUI discussions.
- Discord community: link in README.
- Follow release notes and blog posts for new features (subscribe to repo releases).
- Speak at meetups, write blog posts, or answer questions to grow visibility and knowledge.

9. Sustainable contribution workflow

Checklist before submitting work: - [ ] Reproduced issue with minimal sample. - [ ] Wrote or updated tests covering change. - [ ] Verified on all affected platforms (Windows/macOS/Linux/Mobile/Browser where applicable). - [ ] Performance measured if relevant. - [ ] Docs/samples updated.

10. Practice exercises

1. Clone Avalonia repo, run `build.ps1` (or `build.sh`), and launch ControlCatalog. Inspect the code for one control you use frequently.
2. Set up symbol/source mapping in your IDE and step into `TextBlock` rendering while running ControlCatalog.
3. File a sample issue in a sandbox repo (practice minimal repro). Outline expected vs actual behavior clearly.
4. Write a headless unit test for a simple control (e.g., verifying a custom control draws expected output) and run it locally.
5. Pick an area of docs that needs improvement (e.g., design-time tooling) and draft a doc update in the avalonia-docs repo.

Look under the hood (source bookmarks)

- Repo root: github.com/AvaloniaUI/Avalonia
- Build scripts: `build.ps1`, `build.sh`
- Issue templates: `.github/ISSUE_TEMPLATE` directory (bug/feature request).
- PR template: `.github/pull_request_template.md`.

Check yourself

- Where do you find tests or samples relevant to a control you're debugging?
- How do you step into Avalonia sources from your app?
- What makes a strong issue/PR description?
- How can you contribute documentation or samples beyond code?
- Which community channels help you stay informed about releases and roadmap?

What's next - Return to Index or revisit topics as needed. Keep exploring and contributing!