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.

Create your own architecture sketch showing `Avalonia.Base` at the foundation, `Avalonia.Controls` and `Avalonia.Markup.Xaml` layered above it, theme assemblies such as `Avalonia.Themes.Fluent`, and platform backends surrounding the stack. Keep the diagram handy as you read later chapters.

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.

The MVVM contract inside Avalonia - `AvaloniaObject` and `StyledElement`: every control derives from

`AvaloniaObject`, gaining access to the dependency property system. `StyledElement` adds styling, resources, and the logical tree. These classes live in `Avalonia.Base`. - `AvaloniaLocator`: a lightweight service locator (`AvaloniaLocator.cs`) used by the framework to resolve services (logging, platform implementations). You can register your own singletons during startup when integrating DI containers. - Logical vs visual tree: controls participate in a logical tree (resources, data context inheritance) and a visual tree (rendered elements). Explore helpers such as `LogicalTreeExtensions` and the DevTools tree viewers to see both perspectives. - `ViewLocator`: MVVM projects often map view models to views dynamically. Avalonia ships a default `ViewLocator` in `Avalonia.ReactiveUI`, and you can create your own service that resolves XAML types by naming convention. - Service registration: register singleton services with `AvaloniaLocator.CurrentMutable.Bind<TService>().ToConstant(instance)` during `AppBuilder` configuration so both code-behind and markup extensions can retrieve them.

Data context flow across trees - Data contexts inherit through the logical tree (e.g., `Window` → `Grid` → `TextBlock`). Controls outside that tree, such as popups, will not inherit automatically; explicitly assign contexts when necessary. - The visual tree may contain additional elements introduced by control templates. Bindings resolve by name through the logical tree first, then resource lookups, so understanding both structures keeps bindings predictable. - Use DevTools' Logical/Visual tabs to inspect the tree at runtime and trace resource lookups or data-context changes.

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. - Pick three controls from ControlCatalog (e.g., Button, SplitView, ColorPicker) and map each to the assembly and namespace hosting its implementation. Sketch the relationships in the architecture diagram you created earlier so you can orient yourself quickly when diving into source.

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

SDK matrix at a glance

Avalonia 11 targets .NET 8.0. The official repository pins versions in `global.json`:

Scenario	SDK / Tooling	Notes
Desktop (Windows/macOS/Linux)	.NET SDK `8.0.x`	Use latest LTS; `global.json` ensures consistent builds across machines.
Android	.NET SDK `8.0.x` + `android` workload	Requires Android Studio or Visual Studio mobile workloads.
iOS/macOS Catalyst	.NET SDK `8.0.x` + `ios` workload	Requires Xcode CLI tools and Apple certificates for device deployment.
Browser (WebAssembly)	.NET SDK `8.0.x` + `wasm-tools` workload	Installs Emscripten toolchain for WASM builds.

Run `dotnet --list-sdks` to confirm the expected SDK version is installed. When multiple SDKs coexist, keep a repo-level `global.json` to pin builds to the Avalonia-supported version.

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.
- Native dependencies: Avalonia bundles Skia; keep GPU drivers updated. When shipping self-contained builds, include ANGLE libraries (`libEGL`, `libGLESv2`, `d3dcompiler_47`) for broader GPU compatibility (see Chapter 26).

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.
- Native dependencies: Avalonia uses Skia via Metal/OpenGL; ensure Command Line Tools are installed (`xcode-select --install`).

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.
- Native dependencies: install Mesa/OpenGL drivers (`sudo apt install mesa-utils`) and ICU libraries for globalization support.

  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
dotnet workload install maui           # Optional: Windows tooling support

# Restore workloads declared in a solution (after cloning a repo)
dotnet workload restore
```

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.
- Add tasks in `.vscode/tasks.json` for `dotnet run` / `dotnet watch` to trigger builds with **Ctrl+Shift+B**.
- Set `"avalonia.preview.host"` to `dotnet` in `.vscode/settings.json` so the previewer launches automatically when you open XAML files.

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.

Template quick-reference

Template	Command	When to use
Desktop (code-behind)	`dotnet new avalonia.app -n MyApp`	Small prototypes with code-behind patterns.
MVVM starter	`dotnet new avalonia.mvvm -n MyApp.Mvvm`	Includes a ViewModel base class and sample bindings.
ReactiveUI	`dotnet new avalonia.reactiveui -n MyApp.ReactiveUI`	If you standardise on ReactiveUI for MVVM.
Cross-platform heads	`dotnet new avalonia.app --multiplatform -n MyApp.Multi`	Generates desktop, mobile, and browser heads in one project.
Split head projects	`dotnet new avalonia.xplat -n MyApp.Xplat`	Separate desktop/mobile projects (Visual Studio friendly).
Control library	`dotnet new avalonia.library -n MyApp.Controls`	Create reusable UI/control libraries.

Pair this with `dotnet workload list` to confirm matching workloads are installed for the heads you create.

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.
- `dotnet new avalonia.xplat -o HelloAvalonia.Xplat` -> generates separate head projects (desktop/mobile) suited to Visual Studio.
- `dotnet new avalonia.library -o HelloAvalonia.Controls` -> starts a reusable control/library project.

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`).
- **GPU anomalies**: temporarily disable GPU (`SKIA_SHARP_GPU=0`) to isolate driver issues, then update GPU drivers or include ANGLE fallbacks.
- **Nightly packages**: add `https://www.myget.org/F/avalonia-nightly/api/v3/index.json` to NuGet sources to test nightly builds; pin a stable package before release.

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`. If workloads are missing, run `dotnet workload restore`.
2. Install the Avalonia templates and scaffold each template from the quick-reference table. Capture which commands require additional workloads.
3. Run one generated app from the CLI and another from your IDE, verifying hot reload or the previewer works in both flows.
4. Clone the Avalonia repo, build it (`./build.sh --target=Build` or `.\build.ps1 -Target Build`), and run the ControlCatalog sample.
5. Inspect `samples/ControlCatalog/ControlCatalog.csproj` and map referenced Avalonia packages to their source folders. Update your architecture sketch with these relationships.
6. Set a breakpoint in `App.axaml.cs` (`OnFrameworkInitializationCompleted`) and step through startup to watch the lifetime initialise.
7. Document SDK versions, workloads, and template output in a team README so new developers can reproduce your setup.

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:OrderRow`).
- To import controls from other assemblies, add the prefix defined by their `[XmlnsDefinition]` attribute (for example, `xmlns:fluent="avares://Avalonia.Themes.Fluent"`).

3. How Avalonia loads this XAML

- `InitializeComponent()` in `MainWindow.axaml.cs` invokes `AvaloniaXamlLoader.Load`, wiring the compiled XAML into the partial class defined by `x:Class`.
- During build, Avalonia's MSBuild tasks generate code that registers resources, name scopes, and compiled bindings for the loader (see Chapter 30 for the full pipeline).
- In design-time or hot reload scenarios, the same loader can parse XAML streams when no compiled version exists, so runtime errors usually originate from this method.
- Keep `x:Class` values in sync with your namespace; mismatches result in `XamlLoadException` messages complaining about missing compiled XAML.

4. 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`.

5. 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.

6. 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>
```

7. 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.

8. 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) packages a reusable view with its own XAML and code-behind. Each `UserControl` creates its own `NameScope` so `x:Name` values remain local.
- **`NameScope`** (see NameScope.cs) governs how `x:Name` lookups work. Use `this.FindControl<T>("OrdersList")` or `NameScope.GetNameScope(this)` to resolve names inside the nearest scope.

Example: add `x:Name="OrdersList"` to the `ItemsControl` in `MainWindow.axaml` and access it from code-behind:

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

        var ordersList = this.FindControl<ItemsControl>("OrdersList");
        // Inspect or manipulate generated visuals here if needed.
```

```
    }
}
```

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.

9. 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.

10. 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.

11. Work with resources (`FindResource`)

- Declare brushes, converters, or styles in `Window.Resources` or `Application.Resources`.
- Retrieve them at runtime with `FindResource` or `TryFindResource`:

```
<Window.Resources>
  <SolidColorBrush x:Key="HighlightBrush" Color="#FFE57F"/>
</Window.Resources>
```

```
private void OnHighlight(object? sender, RoutedEventArgs e)
{
    if (FindResource("HighlightBrush") is IBrush brush)
    {
        Background = brush;
    }
}
```

- `FindResource` walks the logical tree first, then escalates to application resources, mirroring how the XAML parser resolves `StaticResource`.
- Resources defined inside a `UserControl` or `DataTemplate` are scoped; use `this.Resources` to override per-view resources without affecting the rest of the app.

12. 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. Name the `ItemsControl` (`x:Name="OrdersList"`) and call `this.FindControl<ItemsControl>("OrdersList")` in code-behind to verify name scoping.
4. Override `HighlightBrush` in `MainWindow.Resources` and use `FindResource` to swap the window background at runtime (e.g., from a button click).
5. Add a `ListBox` instead of `ItemsControl` and observe how selection adds visual states in the visual tree.
6. Use DevTools to inspect both logical and visual trees for `OrderRow`. Toggle the Namescope overlay to see how scopes nest.

Look under the hood (source bookmarks)

- XAML loader: src/Markup/Avalonia.Markup.Xaml/AvaloniaXamlLoader.cs
- Content control composition: src/Avalonia.Controls/ContentControl.cs
- User controls and name scopes: src/Avalonia.Controls/UserControl.cs
- `NameScope` implementation: src/Avalonia.Base/Styling/NameScope.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`, `ShutdownRequested`, `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: `AvaloniaHeadlessApplicationLifetime.cs`

Desktop lifetime flow

- `MainWindow` must be assigned before `base.OnFrameworkInitializationCompleted()` or no window will appear.
- `ShutdownMode` controls when the app exits (`OnLastWindowClose`, `OnMainWindowClose`, or `OnExplicitShutdown`).
- Subscribe to `ShutdownRequested` to cancel shutdown (e.g., unsaved document prompt). Call `e.Cancel = true` to keep the app running.
- Additional windows can be opened by tracking them in a collection and calling `Show()` / `Close()`.

Single view and browser lifetimes

- Provide a root `Control` via `MainView`. Navigation stacks switch the child content instead of opening new windows.
- For Android/iOS, the host platform handles navigation/back events; forward them to view models via commands.
- Browser lifetime initialises asynchronously—await long-running startup logic before assigning `MainView`.

Headless lifetime notes

- `StartWithHeadless` disables rendering but still runs the dispatcher. Use it for integration tests.
- Combine with `Avalonia.Headless.XUnit` or `Avalonia.Headless.NUnit` to drive UI interactions programmatically.

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`

Purpose	Typical targets	Key members	
`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`, `TaskScheduler.UnobservedTaskE` and `Dispatcher.UIThread.UnhandledException` to capture failures before they tear down the dispatcher. - `IControlledApplicationLifetime` (`ApplicationLifetime`) exposes `Exit` and `Shutdown()` so you can close gracefully after logging or prompting the user.

Example:

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

    Dispatcher.UIThread.UnhandledException += (_, e) =>
    {
        LogFatal(e.Exception);
        e.Handled = true; // optionally keep the app alive after logging
    };

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

`ClassicDesktopStyleApplicationLifetime` exposes `ShutdownMode`, `ShutdownRequested`, and `Shutdown()` so you can decide whether to exit on last window close, on main window close, or only when you call `Shutdown()` explicitly.

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`, subscribe to `ShutdownRequested`, and record when the app exits.
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. Wire `Dispatcher.UIThread.UnhandledException` and verify that handled exceptions keep the app alive while unhandled ones terminate.
6. 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`
- Controlled lifetime interface (`IControlledApplicationLifetime`): `src/Avalonia.Controls/ApplicationLifetimes/`
- Dispatcher unhandled exception hook: `src/Avalonia.Base/Threading/Dispatcher.cs`

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. Layout invalidation and diagnostics

- Call `InvalidateMeasure()` when a control's desired size changes (for example, text content updates).
- Call `InvalidateArrange()` when position changes but desired size remains the same. Panels do this when children move without resizing.
- `LayoutManager` batches these requests; inspect timings via `LayoutPassTiming` or DevTools -> Layout tab.
- Enable DevTools layout overlays (F12 -> Layout) to visualise measure/arrange bounds. Combine with `RendererDebugOverlays.LayoutTimeGraph` to profile layout costs.
- For custom panels, avoid calling `InvalidateMeasure` from inside `MeasureOverride`; schedule work via `Dispatcher` if you must recalc asynchronously.

3. 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.

4. 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.

5. 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>
```

6. 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.

7. 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`.

8. 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.

9. 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.
6. **Layout logging**: Enable `LogArea.Layout` and capture a trace of `Measure/Arrange` calls when resizing. Inspect `LayoutManager.Instance.LayoutPassTiming.LastLayoutTime` to correlate with DevTools overlays.

Look under the hood (source bookmarks)

- Base layout contract: `Layoutable.cs`
- Layout manager: `LayoutManager.cs`
- Layout pass timing & diagnostics: `LayoutPassTiming.cs`, `RendererDebugOverlays.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. Control overview matrix

Scenario	Key controls	Highlights	Source snapshot
Text & numeric input	`TextBox`, `MaskedTextBox`, `NumericUpDown`, `DatePicker`	Validation-friendly inputs with watermarks, masks, spinner buttons, culture-aware dates	`TextBox.cs`, `MaskedTextBox`, `NumericUpDown`, `DatePicker`
Toggles & commands	`ToggleSwitch`, `CheckBox`, `RadioButton`, `Button`	MVVM-friendly toggles and grouped options with automation peers	`ToggleSwitch.cs`
Lists & selection	`ListBox`, `TreeView`, `SelectionModel`, `ItemsRepeater`	Single/multi-select, hierarchical data, virtualization	`SelectionModel`, `TreeView`
Navigation surfaces	`TabControl`, `SplitView`, `Expander`, `TransitioningContentControl`	Tabbed pages, collapsible panes, animated transitions	`SplitView`, `TransitioningContentControl`
Search & pickers	`AutoCompleteBox`, `ComboBox`, `ColorPicker`, `FilePicker` dialogs	Suggest-as-you-type, palette pickers, storage providers	`AutoCompleteBox`, `ColorPicker`
Command surfaces	`SplitButton`, `Menu`, `ContextMenu`, `Toolbar`	Primary/secondary actions, keyboard shortcuts, flyouts	`SplitButton`, `Menu`
Refresh & feedback	`RefreshContainer`, `RefreshVisualizer`, `WindowNotificationManager`, `StatusBar`, `NotificationCard`	Pull-to-refresh gestures, toast notifications, status indicators	`RefreshContainer`, `WindowNotificationManager`

Use this table as a map while exploring ControlCatalog; each section below dives into exemplars from these categories.

3. 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. - Accessibility: provide spoken labels via `AutomationProperties.Name` or `HelpText` on inputs so screen readers identify the fields correctly.

4. 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.

5. 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.CacheLen
```

Controls for virtualization: `VirtualizingStackPanel.cs`.

6. 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`.

7. 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`.

8. Auto-complete, pickers, and dialogs

9. Command surfaces and flyouts

```
<StackPanel Spacing="12">
  <SplitButton Content="Export" Command="{Binding ExportAllCommand}">
    <SplitButton.Flyout>
      <MenuFlyout>
        <MenuItem Header="Export CSV" Command="{Binding ExportCsvCommand}"/>
        <MenuItem Header="Export JSON" Command="{Binding ExportJsonCommand}"/>
        <MenuItem Header="Export PDF" Command="{Binding ExportPdfCommand}"/>
      </MenuFlyout>
    </SplitButton.Flyout>
  </SplitButton>

  <Menu>
    <MenuItem Header="File">
      <MenuItem Header="New" Command="{Binding NewCommand}"/>
      <MenuItem Header="Open..." Command="{Binding OpenCommand}"/>
      <Separator/>
      <MenuItem Header="Exit" Command="{Binding ExitCommand}"/>
    </MenuItem>
    <MenuItem Header="Help" Command="{Binding ShowHelpCommand}"/>
  </Menu>

  <StackPanel Orientation="Horizontal" Spacing="8">
    <Button Content="Copy" Command="{Binding CopyCommand}" HotKey="Ctrl+C"/>
    <Button Content="Paste" Command="{Binding PasteCommand}" HotKey="Ctrl+V"/>
  </StackPanel>
</StackPanel>
```

Notes: - `SplitButton` exposes a primary command and a flyout for secondary options. Automation peers surface both the button and flyout; see `SplitButton.cs`. - `Menu`/`ContextMenu` support keyboard navigation and `AutomationProperties.AcceleratorKey` so shortcuts are announced to assistive tech.

Implementation: `Menu.cs`. - Flyouts can host any control (`MenuFlyout`, `Popup`, `FlyoutBase`). Use `FlyoutBase.ShowAttachedFlyout` to open context actions from command handlers.

```
<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).

10. Refresh gestures and feedback

```
<Window xmlns:ptr="clr-namespace:Avalonia.Controls;assembly=Avalonia.Controls"
        xmlns:notifications="clr-namespace:Avalonia.Controls.Notifications;assembly=Avalonia.Controls"
        ...>
  <Grid>
    <ptr:RefreshContainer RefreshRequested="OnRefreshRequested">
      <ptr:RefreshContainer.Visualizer>
        <ptr:RefreshVisualizer Orientation="TopToBottom"
                               Content="Pull to refresh"/>
      </ptr:RefreshContainer.Visualizer>
      <ScrollViewer>
        <ItemsControl Items="{Binding Orders}"/>
      </ScrollViewer>
    </ptr:RefreshContainer>
  </Grid>
</Window>
```

```
private async void OnRefreshRequested(object? sender, RefreshRequestedEventArgs e)
{
    using var deferral = e.GetDeferral();
    await ViewModel.ReloadAsync();
}
```

- `RefreshContainer` + `RefreshVisualizer` implement pull-to-refresh on any scrollable surface. Source: `RefreshContainer`.
- Always provide an alternate refresh action (button, keyboard) for desktop scenarios.

```
var notifications = new WindowNotificationManager(this)
{
    Position = NotificationPosition.TopRight,
```

```
    MaxItems = 3
};
notifications.Show(new Notification("Update available", "Restart to apply updates.", NotificationType.S
```

- `WindowNotificationManager` displays toast notifications layered over the current window; combine with inline `NotificationCard` or `InfoBar` for longer-lived messages. Sources: `WindowNotificationManager`, `NotificationCard`.
- Mark status changes with `AutomationProperties.LiveSetting="Polite"` so assistive technologies announce them.

```
<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>
```

- `StatusBar` hosts persistent indicators (connection status, progress). Implementation: `StatusBar`.

11. 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.

12. 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.

13. Practice exercises

1. Create a "dashboard" page mixing text input, selection lists, tabs, a `SplitButton`, 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 and verify automation names.
3. Replace the `ListBox` with a `TreeView` for hierarchical data; add an `Expander` per root item.
4. Wire up a `RefreshContainer` around a scrollable list and implement the `RefreshRequested` deferal pattern. Provide a fallback refresh button for keyboard users.
5. Register a singleton `WindowNotificationManager`, show a toast when the refresh completes, and style inline `NotificationCard` messages for success and error states.
6. Customise button states by adding pseudo-class styles and confirm they match the ControlCatalog defaults.
7. 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`
- Command & navigation surfaces: `src/Avalonia.Controls/SplitButton`, `src/Avalonia.Controls/SplitView`
- Refresh & notifications: `src/Avalonia.Controls/PullToRefresh`, `src/Avalonia.Controls/Notifications`
- 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 and merges it into the resource lookup chain. Use it for brushes, colors, converters, and typography resources.
- `StyleInclude` expects `Styles` (or a single `Style`) and registers selectors. Use `avares://Assembly/Path.axaml` URIs to include styles from other assemblies (for example, `avares://Avalonia.Themes.Fluent/Controls/Button.xam`
- When you rename assemblies or move resource files, update the `Source` URI; missing includes surface as `XamlLoadException` during startup.

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: 1. Control-local resources (`this.Resources`). 2. Logical tree parents (user controls, windows). 3. `Application.Resources`. 4. Theme dictionaries merged by `FluentTheme` (light/dark/high contrast). 5. System theme fallbacks.

The implementation lives in `ResourceDictionary.cs`. DevTools -> Resources panel shows the chain and which dictionary satisfied a lookup.

4. Theme variant scope (local theming)

5. Migrating and overriding Fluent resources

When you need to change Fluent defaults globally (for example, switch accent colors or typography), supply variant-specific dictionaries. Place these under `Application.Resources` with a `ThemeVariant` attribute so they override the theme-provided value only for matching variants.

```
<Application.Resources>
  <ResourceInclude Source="avares://ThemePlayground/Styles/Colors.axaml"/>
  <ResourceDictionary ThemeVariant="Light">
    <SolidColorBrush x:Key="SystemAccentColor" Color="#2563EB"/>
  </ResourceDictionary>
  <ResourceDictionary ThemeVariant="Dark">
```

```
    <SolidColorBrush x:Key="SystemAccentColor" Color="#60A5FA"/>
  </ResourceDictionary>
</Application.Resources>
```

- Keys that match Fluent resources (`SystemAccentColor`, `SystemControlBackgroundBaseLowBrush`, etc.) override the defaults only for the specified variant.
- Keep overrides minimal: inspect the Fluent source to copy exact keys. Replace `FluentTheme` with `SimpleTheme` if you want the simple default look.
- To migrate an existing design system, split colors/typography into `ResourceDictionary` files and create `ControlTheme` overrides for specific controls rather than editing Fluent templates in place.

`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.

6. 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`.

7. 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.

8. 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	Applies when
`:pointerover`	Pointer hovers over the control
`:pressed`	Pointer is pressed / command triggered
`:checked`	Toggleable control is on (`CheckBox`, `ToggleSwitch`, `RadioButton`)
`:focus` / `:focus-within`	Control (or a descendant) has keyboard focus
`:disabled`	`IsEnabled = false`
`:invalid`	A binding reports validation errors

Pseudo-class documentation lives in `Selectors.md` and runtime code under `Selector.cs`. Combine pseudo-classes with style classes (e.g., `Button.primary:pointerover`) to keep state-specific visuals consistent and

accessible.

9. 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.

10. 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);
```

11. 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`
- Selector engine & pseudo-classes: `Selector.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`) and imperative helpers via `BindingOperations`. - Connect collections to `ItemsControl`/`ListBox` with data templates, `SelectionModel`, and compiled binding expressions. - Use converters, validation (`INotifyDataErrorInfo`), asynchronous bindings, and reactive bridges (`AvaloniaPropertyObservable`). - Bind to attached properties, tune performance with compiled bindings, and diagnose issues using 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. - `BindingOperations`: static helpers to install, remove, or inspect bindings imperatively. - `ExpressionObserver`: low-level observable pipeline underpinning async, compiled, and reactive bindings.

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.

`BindingOperations.SetBinding` mirrors WPF/WinUI and is useful when you need to create bindings from code (for dynamic property names or custom controls). `BindingOperations.ClearBinding` removes them safely, keeping reference tracking intact.

2. Binding scopes and source selection

Binding sources are resolved differently depending on the binding type:

- **`DataContext` inheritance** – `StyledElement.DataContext` flows through the logical tree. Setting `DataContext` on a container automatically scopes child bindings.
- **Element name** – `{Binding ElementName=Root, Path=Value}` uses `NameScope` lookup to find another control.
- **Relative source** – `{Binding RelativeSource={RelativeSource AncestorType=ListBox}}` walks the logical tree to find an ancestor of the specified type.
- **Self bindings** – `{Binding Path=Bounds, RelativeSource={RelativeSource Self}}` is handy when exposing properties of the control itself.
- **Static/CLR properties** – `{Binding Path=(local:ThemeOptions.AccentBrush)}` reads attached or static properties registered as Avalonia properties.

Avalonia also supports multi-level ancestor search and templated parent references:

```
<TextBlock Text="{Binding DataContext.Title, RelativeSource={RelativeSource AncestorType=Window}}"/>

<ContentControl ContentTemplate="{StaticResource CardTemplate}" />

<DataTemplate x:Key="CardTemplate" x:DataType="vm:Card">
  <Border Background="{Binding Source={RelativeSource TemplatedParent}, Path=Background}"/>
</DataTemplate>
```

When creating controls dynamically, use `BindingOperations.SetBinding` so the engine tracks lifetimes and updates `DataContext` inheritance correctly:

```
var binding = new Binding
{
    Path = "Person.FullName",
    Mode = BindingMode.OneWay
};

BindingOperations.SetBinding(nameTextBlock, TextBlock.TextProperty, binding);
```

`BindingOperations.ClearBinding(nameTextBlock, TextBlock.TextProperty)` detaches it. To observe `AvaloniaProperty` values reactively, wrap them with `AvaloniaPropertyObservable.Observe`:

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

var textStream = AvaloniaPropertyObservable.Observe(this, TextBox.TextProperty)
    .Select(value => value as string ?? string.Empty);

var subscription = textStream.Subscribe(text => ViewModel.TextLength = text.Length);
```

`AvaloniaPropertyObservable` lives in `AvaloniaPropertyObservable.cs` and bridges the binding system with `IObservable<T>` pipelines. Dispose the subscription in `OnDetachedFromVisualTree` (or your view's `Dispose` pattern) to avoid leaks.

3. Set up the sample project

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

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

4. 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.

5. 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
}
```

6. 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.

Binding to attached properties

Avalonia registers attached properties (e.g., `ScrollViewer.HorizontalScrollBarVisibilityProperty`) as `AvaloniaProperty`. Bind to them by wrapping the property name in parentheses:

```
<ListBox ItemsSource="{Binding Items}">
  <ListBox.Styles>
    <Style Selector="ListBox">
      <Setter Property="(ScrollViewer.HorizontalScrollBarVisibility)" Value="Disabled"/>
      <Setter Property="(ScrollViewer.VerticalScrollBarVisibility)" Value="Auto"/>
    </Style>
  </ListBox.Styles>
</ListBox>

<Border Background="{Binding (local:ThemeOptions.AccentBrush)}"/>
```

Attached property syntax also works inside `Binding` or `MultiBinding`. When setting them from code, use the generated static accessor (e.g., `ScrollViewer.SetHorizontalScrollBarVisibility(listBox, ScrollBarVisibility.Disabled);`).

7. 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`.

8. 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`.

9. 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}"/>
```

10. 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`.

11. 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).

12. 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.

13. 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. **Runtime binding helpers**: dynamically add a `TextBlock` for each person in a collection, use `BindingOperations.SetBinding` to wire `TextBlock.Text`, then `ClearBinding` when removing the item.
6. **Observable probes**: pipe `TextBox.TextProperty` through `AvaloniaPropertyObservable.Observe` and surface the text length in the UI.
7. **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`
- Binding helpers: `BindingOperations.cs`, `ExpressionObserver.cs`
- Compiled bindings: `CompiledBindingExtension.cs`
- Multi/Priority binding: `MultiBinding.cs`, `PriorityBinding.cs`
- Reactive bridge: `AvaloniaPropertyObservable.cs`
- Selection model: `SelectionModel.cs`
- Validation: `Validation.cs`
- Diagnostics: `BindingDiagnostics.cs`

Check yourself

- When would you choose `CompiledBinding` over `Binding`, and what prerequisites does it have?
- How do `ElementName`, `RelativeSource`, and attached property syntax change the binding source?
- Which scenarios call for `MultiBinding`, `PriorityBinding`, or programmatic calls to `BindingOperations.SetBinding`?
- How does `AvaloniaPropertyObservable.Observe` integrate with the binding engine, and when would you prefer it over classic bindings?
- Which tooling surfaces validation and binding errors during development, and how would you enable the relevant diagnostics?

What's next - Next: Chapter 9

9. Commands, events, and user input

Goal - Understand how routed events flow through `InputElement` and how gesture recognizers, commands, and keyboard navigation fit together. - Choose between MVVM-friendly commands and low-level events effectively (and bridge them with hotkeys and toolkits). - Wire keyboard shortcuts, pointer gestures, and access keys; capture pointer input for drag scenarios with `HotKeyManager` and pointer capture APIs. - Implement asynchronous commands and recycle CanExecute logic with reactive or toolkit helpers. - Diagnose input issues with DevTools (Events view), logging, and custom event tracing.

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: `Avalonia.Interactivity` defines `RoutedEvent`, event descriptors, and routing strategies. - Core element hierarchy: `InputElement` (inherits `Interactive` → `Visual` → `Animatable`) exposes focus, input, and command helpers that every control inherits. - Devices & state: `Avalonia.Base/Input` provides `Pointer`, `KeyboardDevice`, `KeyGesture`, `PointerPoint`. - Gesture recognizers: `GestureRecognizers` translate raw pointer data into tap, scroll, drag behaviors. - Hotkeys & command sources: `HotkeyManager` walks the visual tree to resolve `KeyGesture`s against `ICommand` targets.

Event flow: 1. Devices raise raw events (`PointerPressed`, `KeyDown`). Each is registered as a `RoutedEvent` with a routing strategy (tunnel, bubble, direct). 2. `InputElement` hosts the event metadata, raising class handlers and instance handlers. 3. Gesture recognizers subscribe to pointer streams and emit semantic events (`Tapped`, `DoubleTapped`, `PointerPressedEventArgs`). 4. Command sources (`Button.Command`, `KeyBinding`, `InputGesture`) execute `ICommand` implementations and update `CanExecute`.

Creating custom events uses the static registration helpers:

```
public static readonly RoutedEvent<RoutedEventArgs> DragStartedEvent =
    RoutedEvent.Register<Control, RoutedEventArgs>(
        nameof(DragStarted),
        RoutingStrategies.Bubble);

public event EventHandler<RoutedEventArgs> DragStarted
{
    add => AddHandler(DragStartedEvent, value);
    remove => RemoveHandler(DragStartedEvent, value);
}
```

`RoutingStrategies` live in `RoutedEvent.cs`; each handler chooses whether the event should travel from root to leaf (tunnel) or leaf to root (bubble).

2. Input playground 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. Hotkeys are attached in XAML using `HotKeyManager.HotKey`, keeping the view model free of UI dependencies.

```
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, KeyGesture, and HotKeyManager

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 `KeyGesture` and `KeyGestureConverter`. For multiple gestures, add more `KeyBinding` entries on the relevant `InputElement`.

`HotKeyManager` attached property

`KeyBinding` only fires while the owning control is focused. To register process-wide hotkeys that stay active as long as a control is in the visual tree, attach a `KeyGesture` via `HotKeyManager.HotKey`:

```
<Window xmlns:controls="clr-namespace:Avalonia.Controls;assembly=Avalonia.Controls">
  <Button Content="Save"
          Command="{Binding SaveCommand}"
          controls:HotKeyManager.HotKey="Ctrl+Shift+S"/>
</Window>
```

`HotKeyManager` walks up to the owning `TopLevel` and injects a `KeyBinding` for you, even when the button is not focused. In code you can call `HotKeyManager.SetHotKey(button, new KeyGesture(Key.S, KeyModifiers.Control | KeyModifiers.Shift));`. Implementation lives in `HotkeyManager.cs`.

Bring `Avalonia.Input` into scope when assigning gestures programmatically so `KeyGesture` and `KeyModifiers` resolve.

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`). Combine them with `HotKeyManager` to offer both menu accelerators and global commands.

6. Pointer gestures, capture, and drag initiation

Avalonia ships gesture recognizers derived from `GestureRecognizer`. Attach them via `GestureRecognizers` to translate raw pointer data into commands:

```
<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 (e.g., 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. Use `PointerEventArgs.GetCurrentPoint` to inspect buttons, pressure, tilt, or contact rectangles for richer interactions.

```
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);
}
```

To cancel capture, call `e.Pointer.Capture(null)` or use `Pointer.Captured`. See `PointerDevice.cs` and `PointerEventArgs.cs` 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. Keyboard focus management and navigation

- Call `Focus()` to move input programmatically. `InputElement.Focus()` delegates to `FocusManager`.
- Use `Focusable="False"` on decorative elements so they are skipped in traversal.
- Control tab order with `TabIndex` (lower numbers focus first); combine with `KeyboardNavigation.TabNavigation` to scope loops.
- Create focus scopes (`Focusable="True"` + `IsTabStop="True"`) for popups/overlays so focus returns to the invoking control when closed.
- Use `TraversalRequest` and `KeyboardNavigationHandler` to implement custom arrow-key navigation for grids or toolbars.

```
<StackPanel KeyboardNavigation.TabNavigation="Cycle" Spacing="8">
  <TextBox x:Name="First" Watermark="First name"/>
  <TextBox x:Name="Second" Watermark="Last name"/>
  <Button Content="Focus second" Command="{Binding FocusSecondCommand}"/>
</StackPanel>
```

```
public void FocusSecond()
{
    var scope = FocusManager.Instance.Current;
    var second = this.FindControl<TextBox>("Second");
    scope?.Focus(second);
}
```

For MVVM-safe focus changes, expose an interaction request (event or `Interaction<T>` from ReactiveUI) and let the view handle it. Keyboard navigation services live under `IKeyboardNavigationHandler`.

9. Bridging commands with MVVM frameworks

- **CommunityToolkit.Mvvm** – `RelayCommand`/`AsyncRelayCommand` implement `ICommand` and expose `CanExecuteChanged`. Use `[RelayCommand]` attributes to generate commands and wrap business logic in partial classes.
- **ReactiveUI** – `ReactiveCommand` exposes `IObservable` execution pipelines, throttling, and cancellation. Bind with `{Binding SaveCommand}` just like any other `ICommand`.
- **Prism / DryIoc** – `DelegateCommand` supports `ObservesCanExecute` and integrates with dependency injection lifetimes.

To unify event-heavy code paths with commands, expose interaction helpers instead of code-behind:

```
public Interaction<Unit, PointerPoint?> StartDragInteraction { get; } = new();

public async Task BeginDragAsync()
{
    var pointerPoint = await StartDragInteraction.Handle(Unit.Default);
    if (pointerPoint is { } point)
    {
        // Use pointer data to seed drag operation
    }
}
```

The example uses `ReactiveUI.Interaction` and `Avalonia.Input.PointerPoint`; adapt the pattern to your MVVM framework of choice.

In XAML, use `Interaction` behaviors (`<interactions:Interaction.Triggers>` or toolkit `EventToCommandBehavior`) to connect events such as `PointerPressed` to `ReactiveCommand`s without writing code-behind. This keeps

event routing logic discoverable while leaving testable command logic in the view model.

10. 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`.

11. Asynchronous command patterns

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);
}
```

12. 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.

13. Practice exercises

1. Extend InputPlayground with a routed event logger: call `AddHandler` for `PointerPressedEvent`/`KeyDownEvent`, display bubbling order, and compare to the DevTools Events tab.
2. Register a global `Ctrl+Shift+S` gesture with `HotKeyManager.HotKey` (in XAML or via `HotKeyManager.SetHotKey`), then toggle the button's `IsEnabled` state and confirm `CanExecute` updates propagate.
3. Build a drag-to-reorder list that uses pointer capture and `PointerPoint.Properties` to track left vs right button drags.
4. Integrate a `ReactiveCommand` or toolkit `AsyncRelayCommand` with a drag `Interaction<T>` so the view model decides when async work starts.
5. Configure `KeyboardNavigation.TabNavigation="Cycle"` on a popup and verify focus returns to the launcher when it closes.

Look under the hood (source bookmarks)

- Routed events: `RoutedEvent.cs`, `RoutingStrategies`
- Commands: `ButtonBase.Command`, `MenuItem.Command`, `KeyBinding`
- Hotkeys: `KeyGesture.cs`, `HotkeyManager.cs`
- Input elements & gestures: `InputElement.cs`, `GestureRecognizer.cs`
- Focus: `FocusManager.cs`, `IKeyboardNavigationHandler`
- Text input pipeline: `TextInputMethodClient.cs`

Check yourself

- What advantages do commands offer over events in MVVM architectures?
- When would you choose `KeyBinding` vs registering a gesture with `HotKeyManager`?
- Which API captures `PointerPoint` data during drag initiation and why does it matter?
- How would you bridge a pointer event to a `ReactiveCommand` or toolkit command without code-behind?
- Which tooling surfaces routed events, and how do you enable verbose input logging?

What's next - Next: Chapter 10

10. Working with resources, images, and fonts

Goal - Master `avares://` URIs, `AssetLoader`/`IAssetLoader`, and `ResourceDictionary` lookup so you can bundle assets cleanly. - Display raster and vector images, control caching/interpolation, and brush surfaces with images (including SVG pipelines). - Load custom fonts, configure `FontManagerOptions`, and swap font families at runtime. - Understand resource fallback order, dynamic `ResourceDictionary` updates, and diagnostics when a lookup fails. - Tune DPI scaling, bitmap interpolation, and responsive asset strategies that scale across devices.

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. Resource dictionaries and lookup order

`ResourceDictionary` derives from `ResourceProvider` and implements `IResourceProvider`. When you request `{StaticResource}` or call `TryGetResource`, Avalonia walks this chain:

1. The requesting `IResourceHost` (control, style, or application).
2. Parent styles (`<Style.Resources>`), control templates, and data templates.
3. Theme dictionaries (`ThemeVariantScope`, `Application.Styles`, `Application.Resources`).
4. Merged dictionaries (`<ResourceDictionary.MergedDictionaries>` or `<ResourceInclude>`).
5. Global application resources and finally platform defaults (`SystemResources`).

`ResourceDictionary.cs` and `ResourceNode.cs` coordinate this traversal. Use `TryGetResource` when retrieving values from code:

```
if (control.TryGetResource("AccentBrush", ThemeVariant.Dark, out var value) && value is IBrush brush)
{
    control.Background = brush;
}
```

`ThemeVariant` lets you request a variant-specific value; pass `ThemeVariant.Default` to follow the same logic as `{DynamicResource}`.

Merge dictionaries to break assets into reusable packs:

```
<ResourceDictionary>
  <ResourceDictionary.MergedDictionaries>
    <ResourceInclude Source="avares://AssetsDemo/Assets/Colors.axaml"/>
    <ResourceInclude Source="avares://AssetsDemo/Assets/Icons.axaml"/>
```

```
  </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
```

Each merged dictionary is loaded lazily via `IAssetLoader`, so make sure the referenced file is marked as `AvaloniaResource`.

3. Loading assets in XAML and code

XAML

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

Code using `AssetLoader`

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

var uri = new Uri("avares://AssetsDemo/Assets/Images/logo.png");
var assetLoader = AvaloniaLocator.Current.GetRequiredService<IAssetLoader>();

await using var stream = assetLoader.Open(uri);
LogoImage.Source = new Bitmap(stream);
```

`AssetLoader` is a static helper over the same `IAssetLoader` service. Prefer the interface when unit testing or when you need to mock resource access. Both live in `Avalonia.Platform`.

Need to probe for optional assets? Use `assetLoader.TryOpen(uri)` or `AssetLoader.Exists(uri)` to avoid exceptions.

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.

4. Raster images, decoders, and caching

`Image` renders `Avalonia.Media.Imaging.Bitmap`. Decode streams once and keep the bitmap alive when the pixels are reused, instead of calling `new Bitmap(stream)` for every render. 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`.

Decode oversized images to a target width/height to save memory:

```
await using var stream = assetLoader.Open(uri);
using var decoded = Bitmap.DecodeToWidth(stream, 512);
PhotoImage.Source = decoded;
```

`Bitmap` and decoder helpers live in `Bitmap.cs`. Avalonia picks the right codec (PNG, JPEG, WebP, BMP, GIF) using Skia; for unsupported formats supply a custom `IBitmapDecoder`.

5. 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`.

6. 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`.

`StreamGeometryContext` for programmatic icons

Generate vector shapes in code when you need to compose icons dynamically or reuse geometry logic:

```
var geometry = new StreamGeometry();

using (var ctx = geometry.Open())
{
    ctx.BeginFigure(new Point(2, 12), isFilled: false);
    ctx.LineTo(new Point(9, 19));
    ctx.LineTo(new Point(22, 4));
    ctx.EndFigure(isClosed: false);
}

IconPath.Data = geometry;
```

`StreamGeometry` and `StreamGeometryContext` live in `StreamGeometryContext.cs`. Remember to freeze geometry instances or share them via resources to reduce allocations.

SVG support

Install the `Avalonia.Svg.Skia` package to render SVG assets natively:

```
<svg:SvgImage xmlns:svg="clr-namespace:Avalonia.Svg.Controls;assembly=Avalonia.Svg.Skia"
              Source="avares://AssetsDemo/Assets/Images/logo.svg"
              Stretch="Uniform" />
```

SVGs stay sharp at any DPI and can adapt colors if you parameterize them (e.g., replace fill attributes at build time). For simple icons, converting the path data into XAML keeps dependencies minimal.

7. 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.

Runtime font swaps and custom collections

You can inject fonts at runtime without restarting the app. Register an embedded collection and update resources:

```
using Avalonia.Media;
using Avalonia.Media.Fonts;

var baseUri = new Uri("avares://AssetsDemo/Assets/BrandFonts/");
var collection = new EmbeddedFontCollection(new Uri("fonts:brand"), baseUri);

FontManager.Current.AddFontCollection(collection);

Application.Current!.Resources["BodyFont"] = new FontFamily("fonts:brand#Brand Sans");
```

`EmbeddedFontCollection` pulls all font files under the provided URI using `IAssetLoader`. Removing the collection via `FontManager.Current.RemoveFontCollection(new Uri("fonts:brand"))` detaches it again.

8. 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`.

9. Dynamic resources, theme variants, and runtime updates

Bind brushes via `DynamicResource` so assets respond to theme changes. When a dictionary entry changes, `ResourceDictionary.ResourcesChanged` notifies every subscriber and controls update automatically:

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

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

At runtime you can swap assets:

```
Application.Current!.Resources["AvatarFallbackBrush"] = new SolidColorBrush(Color.Parse("#3B82F6"));
```

To scope variants, wrap content in a `ThemeVariantScope` and supply dictionaries per variant:

```
<ThemeVariantScope RequestedThemeVariant="Dark">
  <ThemeVariantScope.Resources>
    <SolidColorBrush x:Key="AvatarFallbackBrush" Color="#E5E7EB"/>
  </ThemeVariantScope.Resources>
  <ContentPresenter Content="{Binding}"/>
</ThemeVariantScope>
```

`ThemeVariantScope` relies on `IResourceHost` to merge dictionaries in order (scope → parent scope → application). To inspect all merged resources in DevTools, open **Resources** and observe how `RequestedThemeVariant` switches dictionaries.

10. 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");
```

- Subscribe to `Application.Current.Resources.ResourcesChanged` (or scope-specific hosts) to log when dictionaries update, especially when debugging `DynamicResource` refreshes.

11. 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>
```

12. Practice exercises

1. Move brand colors into `Assets/Brand.axaml`, include it with `<ResourceInclude Source="avares://AssetsDemo/Asse` and verify lookups succeed from a control in another assembly.
2. Build an image component that prefers SVG (`SvgImage`) but falls back to a PNG `Bitmap` on platforms where the SVG package is missing.
3. Decode a high-resolution photo with `Bitmap.DecodeToWidth` and compare memory usage against eagerly loading the original stream.
4. Register an `EmbeddedFontCollection` at runtime and swap your typography resources by updating `Application.Current.Resources["BodyFont"]`.
5. Toggle `ThemeVariantScope.RequestedThemeVariant` at runtime and confirm `DynamicResource`-bound brushes and images update without recreating controls.

Look under the hood (source bookmarks)

- Resource system: `ResourceProvider.cs`, `ResourceDictionary.cs`
- Asset loader and URIs: `AssetLoader.cs`, `ResourceInclude.cs`
- Bitmap and imaging: `Bitmap.cs`
- Vector geometry: `StreamGeometryContext.cs`, `Path.cs`
- Fonts & text formatting: `FontManager.cs`, `EmbeddedFontCollection.cs`
- Theme variants and resources: `ThemeVariantScope.cs`, `ResourcesChangedHelper.cs`
- Render options and DPI: `RenderOptions.cs`

Check yourself

- What order does Avalonia search when resolving `{StaticResource}` and `{DynamicResource}`?
- When do you reach for `IAssetLoader` instead of the static `AssetLoader` helper?
- How would you build a responsive icon pipeline that prefers `StreamGeometry`/SVG but falls back to a bitmap?
- Which APIs let you swap font families at runtime without restarting the app?
- How can you confirm that dynamic resource updates propagated after changing `Application.Current.Resources`?

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 3).

2.4 Binding notifications and validation

Bindings surface both conversion errors and validation failures through `BindingNotification` and the `DataValidationException` payload. Listening to those notifications helps you surface validation summaries in the UI and quickly diagnose binding issues during development.

```
public sealed class AccountViewModel : ObservableValidator
{
    private string _email = string.Empty;
```

```
    public ObservableCollection<string> ValidationMessages { get; } = new();

    [Required(ErrorMessage = "Email is required")]
    [EmailAddress(ErrorMessage = "Enter a valid email address")]
    public string Email
    {
        get => _email;
        set => SetProperty(ref _email, value, true);
    }
}
```

`ObservableValidator` lives in CommunityToolkit.Mvvm and combines property change notification with `INotifyDataErrorInfo` support. Expose `ValidationMessages` (e.g., an `ObservableCollection<string>`) to feed summaries or inline hints.

```
<TextBox x:Name="EmailBox"
         Text="{Binding Email, Mode=TwoWay, ValidatesOnNotifyDataErrors=True, UpdateSourceTrigger=Prope
<ItemsControl ItemsSource="{Binding ValidationMessages}"/>
```

```
var subscription = EmailBox.GetBindingObservable(TextBox.TextProperty)
    .Subscribe(result =>
    {
        if (result.HasError && result.Error is BindingNotification notification)
        {
            if (notification.Error is ValidationException validation)
                ValidationMessages.Add(validation.Message);
            else
                Logger.LogError(notification.Error, "Binding failure for Email");
        }
    });

DataValidationErrors.GetObservable(EmailBox)
    .Subscribe(args => ValidationMessages.Add(args.Error.Content?.ToString() ?? string.Empty));
```

`BindingNotification` distinguishes between binding errors and data validation errors (`BindingErrorType`). Validation failures arrive as `DataValidationException` instances on the notification, exposing the offending property and message. Use Avalonia's `DataValidationErrors` helper to observe validation changes and feed a summary control or toast.

2.5 Value converters and formatting

When view and view model types differ, implement `IValueConverter` or `IBindingTypeConverter` to keep view models POCO-friendly.

```
public sealed class TimestampToLocalTimeConverter : IValueConverter
{
    public object? Convert(object? value, Type targetType, object? parameter, CultureInfo culture)
        => value is DateTimeOffset dto ? dto.ToLocalTime().ToString("t", culture) : string.Empty;

    public object? ConvertBack(object? value, Type targetType, object? parameter, CultureInfo culture)
        => DateTimeOffset.TryParse(value as string, culture, DateTimeStyles.AssumeLocal, out var dto) ?
}
```

Register converters in resources and reuse them across DataTemplates:

```
<Window.Resources>
  <local:TimestampToLocalTimeConverter x:Key="LocalTime"/>
</Window.Resources>
```

```
<TextBlock Text="{Binding LastSignIn, Converter={StaticResource LocalTime}}"/>
```

Converters keep view models focused on domain types while views shape presentation. For complex pipelines, combine converters with `Binding.ConverterParameter` or chained bindings.

2.6 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. For advanced scenarios, register an `IGlobalDataTemplates` implementation to provide templates at runtime (e.g., when view models live in feature modules).

```
public sealed class AppDataTemplates : IGlobalDataTemplates
{
    private readonly IServiceProvider _services;

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

    public bool Match(object? data) => data is ViewModelBase;

    public Control Build(object? data)
        => data switch
        {
            HomeViewModel => _services.GetRequiredService<HomeView>(),
            SettingsViewModel => _services.GetRequiredService<SettingsView>(),
            _ => new TextBlock { Text = "No view registered." }
        };
}
```

Register the implementation in `App` or DI container so Avalonia uses it when resolving content.

2.7 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. Composition and state management

3.1 Dependency injection and view model factories

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>();
        }
        else if (ApplicationLifetime is ISingleViewApplicationLifetime singleView)
        {
            singleView.MainView = _services.GetRequiredService<ShellView>();
        }

        base.OnFrameworkInitializationCompleted();
    }

    private static IServiceProvider ConfigureServices()
    {
        var services = new ServiceCollection();
        services.AddSingleton<MainWindow>();
```

```
        services.AddSingleton<ShellView>();
        services.AddSingleton<INavigationService, NavigationService>();
        services.AddTransient<PeopleViewModel>();
        services.AddTransient<HomeViewModel>();
        services.AddSingleton<IPersonService, PersonService>();
        services.AddSingleton<IGlobalDataTemplates, AppDataTemplates>();
        return services.BuildServiceProvider();
    }
}
```

Inject `INavigationService` (or a more opinionated router) into view models to drive navigation. Supplying `IGlobalDataTemplates` from the service provider keeps view discovery aligned with DI—views can request their own dependencies on construction.

3.2 State orchestration with observables

Centralize shared state in dedicated services so view models remain focused on UI coordination:

```
public sealed class DocumentStore : ObservableObject
{
    private readonly ObservableCollection<DocumentViewModel> _documents = new();
    public ReadOnlyObservableCollection<DocumentViewModel> OpenDocuments { get; }

    public DocumentStore()
        => OpenDocuments = new ReadOnlyObservableCollection<DocumentViewModel>(_documents);

    public void Open(DocumentViewModel document)
    {
        if (!_documents.Contains(document))
            _documents.Add(document);
    }

    public void Close(DocumentViewModel document) => _documents.Remove(document);
}
```

Expose commands that call into the store instead of duplicating logic across view models. For undo/redo, track a stack of undoable actions and leverage property observables to record mutations:

```
public interface IUndoableAction
{
    void Execute();
    void Undo();
}

public sealed class UndoRedoManager
{
    private readonly Stack<IUndoableAction> _undo = new();
    private readonly Stack<IUndoableAction> _redo = new();

    public void Do(IUndoableAction action)
    {
        action.Execute();
        _undo.Push(action);
        _redo.Clear();
    }
```

```
    public void Undo() => Execute(_undo, _redo);
    public void Redo() => Execute(_redo, _undo);

    private static void Execute(Stack<IUndoableAction> source, Stack<IUndoableAction> target)
    {
        if (source.TryPop(out var action))
        {
            action.Undo();
            target.Push(action);
        }
    }
}
```

Subscribe to `INotifyPropertyChanged` or use `Observable.FromEventPattern` to capture state snapshots whenever important properties change. This approach works equally well for manual MVVM, CommunityToolkit, or ReactiveUI view models.

3.3 Bridging other MVVM frameworks

- **Prism**: Register `ViewModelLocator.AutoWireViewModel` in XAML and let Prism resolve view models via Avalonia DI. Use Prism's region navigation on top of `ContentControl`-based shells.
- **Caliburn.Micro / Stylet**: Hook their view locator into Avalonia by implementing `IGlobalDataTemplates` or setting `ViewLocator.LocateForModelType` to the framework's resolver.
- **PropertyChanged.Fody / FSharp.ViewModule**: Combine source generators with Avalonia bindings—`BindingNotification` still surfaces validation errors, so logging and diagnostics remain consistent.

The key is to treat Avalonia's property system as the integration point: as long as view models raise property change notifications, you can plug in different MVVM toolkits without rewriting view code.

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.

5.6 Avalonia.ReactiveUI helpers

`Avalonia.ReactiveUI` ships opinionated base classes such as `ReactiveWindow<TViewModel>`, `ReactiveContentControl<TVi` and extension methods that bridge Avalonia's property system with ReactiveUI's `IObservable` pipelines.

```
public partial class ShellWindow : ReactiveWindow<ShellViewModel>
{
    public ShellWindow()
    {
        InitializeComponent();

        this.WhenActivated(disposables =>
        {
            this.OneWayBind(ViewModel, vm => vm.Router, v => v.RouterHost.Router)
                .DisposeWith(disposables);
            this.BindCommand(ViewModel, vm => vm.ExitCommand, v => v.ExitMenuItem)
                .DisposeWith(disposables);
        });
    }
}
```

Activation hooks route `BindingNotification` instances through ReactiveUI's logging infrastructure, so binding failures show up in `RxApp.DefaultExceptionHandler`. Register `ActivationForViewFetcher` when hosting custom controls so ReactiveUI can discover activation semantics:

```
Locator.CurrentMutable.Register(() => new ShellWindow(), typeof(IViewFor<ShellViewModel>));
Locator.CurrentMutable.RegisterConstant(new AvaloniaActivationForViewFetcher(), typeof(IActivationForVi
```

These helpers keep Avalonia bindings, routing, and interactions in sync with ReactiveUI conventions.

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. Compose a multi-view shell that swaps `HomeViewModel`/`SettingsViewModel` via DI-backed `IGlobalDataTemplates` and an `INavigationService`.
2. Extend the account form to surface a validation summary by listening to `DataValidationErrors.GetObservable` and logging `BindingNotification` errors.
3. Author a currency `IValueConverter`, register it in resources, and verify formatting in both classic and ReactiveUI views.
4. Implement an async load pipeline with `ReactiveCommand`, binding `IsExecuting` to a progress indicator and asserting behaviour with `TestScheduler`.
5. Add undo/redo support to the People sample by capturing `INotifyPropertyChanged` via `Observable.FromEventPattern` and replaying changes.

Look under the hood (source bookmarks)

- Binding diagnostics: `BindingNotification.cs`
- Data validation surfaces: `DataValidationErrors.cs`
- Avalonia + ReactiveUI integration: `Avalonia.ReactiveUI`
- Global templates: `IGlobalDataTemplates.cs`
- Value conversion defaults: `DefaultValueConverter.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 `BindingNotification` and `DataValidationErrors` help diagnose problems during binding?
- 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 advantage does `IGlobalDataTemplates` offer over static XAML data templates?

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
`ISingleTopLevelApplicationLifetime`	Single top-level host (preview/embedded scenarios)	Exposed by the runtime; inspect via `ApplicationLifetime as ISingleTopLevelApplicationLifetime`

`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();
}
```

`ISingleTopLevelApplicationLifetime` is currently marked `[PrivateApi]`, but you may see it when Avalonia hosts supply a single `TopLevel`. Treat it as read-only metadata rather than something you implement yourself.

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`.

Subscribe to `screens.Changed` when you need to react to hot-plugging monitors or DPI changes:

```
screens.Changed += (_, _) =>
{
    var active = screens.ScreenFromWindow(this);
    Logger.LogInformation("Monitor layout changed. Active screen: {Bounds}", active.WorkingArea);
};
```

`WindowBase.Screens` always maps to the platform's latest monitor topology, so you can reposition tool windows or popups when displays change.

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?", Messag
        if (!confirm)
            e.Cancel = true;
    }
};
```

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

2.5 Window lifecycle events (`WindowBase`)

`WindowBase` is the shared base type for `Window` and other top-levels. It raises events that fire before layout runs, letting you respond to activation, resizing, and positioning at the window layer:

```
public partial class ToolWindow : Window
{
    public ToolWindow()
    {
        InitializeComponent();
        Activated += (_, _) => StatusBar.Text = "Active";
        Deactivated += (_, _) => StatusBar.Text = "Inactive";
        PositionChanged += (_, e) => Logger.LogInformation("Moved to {Point}", e.Point);
```

```
        Resized += (_, e) => Metrics.Track(e.Size, e.Reason);
        Closed += (_, _) => _subscriptions.Dispose();
    }
}
```

`WindowBase.Resized` reports the reason the platform resized your window (user drag, system DPI change, maximize). Distinguish it from `Control.SizeChanged`, which fires after layout completes. Use `WindowBase.IsActive` to trigger focus-sensitive behaviour such as pausing animations when the window moves to the background.

2.6 Platform-specific window features

Avalonia exposes chrome customisation through `TopLevel` properties:

```
TransparencyLevelHint = new[] { WindowTransparencyLevel.Mica, WindowTransparencyLevel.Acrylic, WindowTra
SystemDecorations = SystemDecorations.None;
ExtendClientAreaToDecorationsHint = true;
ExtendClientAreaChromeHints = ExtendClientAreaChromeHints.SystemChrome | ExtendClientAreaChromeHints.OS
WindowStartupLocation = WindowStartupLocation.CenterScreen;
```

Combine those settings with platform options to unlock OS-specific effects:

- **Windows** (`Win32PlatformOptions`): enable `CompositionBackdrop` or `UseWgl` for specific GPU paths. Set `WindowEffect = new MicaEffect();` to match Windows 11 styling.
- **macOS** (`MacOSPlatformOptions`): toggle `ShowInDock`, `DisableDefaultApplicationMenu`, and `UseNativeMenuBar` per window.
- **Linux/X11** (`X11PlatformOptions`): control `EnableIME`, `EnableTransparency`, and `DisableDecorations` when providing custom chrome.

Always test transparency fallbacks—older GPUs may fall back to `Opaque`. Query `ActualTransparencyLevel` at runtime to reflect final behaviour in the UI.

2.7 Coordinating shutdown with `ShutdownRequestedEventArgs`

`IClassicDesktopStyleApplicationLifetime` exposes a `ShutdownRequested` event. Cancel it when critical work is in progress or when you must prompt the user:

```
if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
{
    desktop.ShutdownRequested += (_, e) =>
    {
        if (_documentStore.HasDirtyDocuments && !ConfirmShutdown())
            e.Cancel = true;

        if (e.IsOSShutdown)
            Logger.LogWarning("OS initiated shutdown");
    };
}
```

Return `true` from `ConfirmShutdown()` only after persisting state or when the user explicitly approves. Pair this with `ShutdownMode` to decide whether closing the main window exits the entire application.

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 SplitView shell navigation

For sidebars or hamburger menus, wrap the navigation service in a `SplitView` so content and commands share a host:

```
<SplitView IsPaneOpen="{Binding IsPaneOpen}"
           DisplayMode="CompactOverlay"
           CompactPaneLength="48"
           OpenPaneLength="200">
  <SplitView.Pane>
    <ItemsControl ItemsSource="{Binding NavigationItems}">
      <ItemsControl.ItemTemplate>
        <DataTemplate>
          <Button Content="{Binding Title}"
                  Command="{Binding NavigateCommand}"/>
        </DataTemplate>
      </ItemsControl.ItemTemplate>
    </ItemsControl>
  </SplitView.Pane>
  <TransitioningContentControl Content="{Binding Current}"/>
</SplitView>
```

Expose `NavigationItems` as view-model descriptors (title + command). Pair with `SplitView.PanePlacement` to adapt between desktop (left rail) and mobile (bottom sheet). Listen to `TopLevel.BackRequested` to collapse the pane when the host (Android, browser, web view) signals a system back gesture.

3.4 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.

5.4 System back navigation

`TopLevel.BackRequested` bubbles up hardware or browser navigation gestures through Avalonia's `ISystemNavigationManagerImpl`. Subscribe to it when embedding in Android, browser, or platform WebView hosts:

```
var topLevel = TopLevel.GetTopLevel(control);
if (topLevel is { })
{
    topLevel.BackRequested += (_, e) =>
    {
        if (_navigation.Pop())
            e.Handled = true;
```

```
    };
}
```

Mark the event as handled when your navigation stack consumes the back action; otherwise Avalonia lets the host perform its default behaviour (e.g., browser history navigation).

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. `TopLevel.BackRequested` maps to the browser's history stack—handle it to keep SPA navigation in sync with the host's back button.

7. Practice exercises

1. Spawn a secondary tool window from the shell, handle `WindowBase.Resized/PositionChanged`, and persist placement per monitor.
2. Hook `ShutdownRequested` to prompt about unsaved documents, cancelling the shutdown when the user declines.
3. Subscribe to `Screens.Changed` and reposition floating windows onto the active display when monitors are hot-plugged.
4. Build a `SplitView` navigation shell that collapses in response to `TopLevel.BackRequested` on Android or the browser.
5. Toggle `TransparencyLevelHint` and `SystemDecorations` per platform and display the resulting `ActualTransparencyLevel` in the UI.

Look under the hood (source bookmarks)

- Window management: `Window.cs`, `WindowBase.cs`
- Lifetimes & shutdown: `ClassicDesktopStyleApplicationLifetime.cs`, `ShutdownRequestedEventArgs.cs`
- Navigation surfaces: `TopLevel.cs`, `SplitView.cs`, `SystemNavigationManagerImpl.cs`
- Screens API: `Screens.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`?
- Which `WindowBase` events fire before layout, and how do they differ from `SizeChanged`?
- How can `TopLevel.BackRequested` improve the experience on Android or the browser?
- What does `ShutdownRequestedEventArgs.IsOSShutdown` tell you, and how would you react to it?
- 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 - Wire desktop menus, context menus, and native menu bars using `Menu`, `MenuItem`, `ContextMenu`, and `NativeMenu`. - Surface dialogs through MVVM-friendly services that switch between `ManagedFileChooser`, `SystemDialog`, and storage providers. - Integrate tray icons, notifications, and app-level commands with the `TrayIcon` API and `TopLevel` services. - Document platform-specific behaviour so menus, dialogs, and tray features degrade gracefully.

Why this matters - Desktop users expect menu bars, keyboard accelerators, and tray icons that follow their OS conventions. - Dialog flows that stay inside services remain unit-testable and work across desktop, mobile, and browser hosts. - System integrations (storage, notifications, clipboard) require a clear view of per-platform capabilities to avoid runtime surprises.

Prerequisites - Chapters 9 (commands and input), 11 (MVVM patterns), and 12 (lifetimes and windowing).

Key namespaces - `Menu.cs` - `MenuItem.cs` - `NativeMenu.cs` - `ContextMenu.cs` - `TrayIcon.cs` - `SystemDialog.cs` - `ManagedFileChooser.cs`

1. Menu surfaces at a glance

1.1 In-window menus (`Menu`/`MenuItem`)

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="MyApp.MainWindow"
        Title="My App" Width="1000" Height="700">
  <DockPanel>
    <Menu DockPanel.Dock="Top">
      <MenuItem Header="_File">
        <MenuItem Header="_New" Command="{Binding AppCommands.New}" HotKey="Ctrl+N"/>
        <MenuItem Header="_Open..." Command="{Binding AppCommands.Open}" HotKey="Ctrl+O"/>
        <MenuItem Header="_Save" Command="{Binding AppCommands.Save}" HotKey="Ctrl+S"/>
        <MenuItem Header="Save _As..." Command="{Binding AppCommands.SaveAs}"/>
        <Separator/>
        <MenuItem Header="E_xit" Command="{Binding AppCommands.Exit}"/>
      </MenuItem>
      <MenuItem Header="_Edit">
        <MenuItem Header="_Undo" Command="{Binding AppCommands.Undo}"/>
        <MenuItem Header="_Redo" Command="{Binding AppCommands.Redo}"/>
      </MenuItem>
      <MenuItem Header="_Help">
        <MenuItem Header="_About" Command="{Binding AppCommands.ShowAbout}"/>
      </MenuItem>
    </Menu>

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

- `MenuItem.HotKey` accepts `KeyGesture` syntax, keeping accelerators in sync with displayed text.
- `AppCommands` is a shared command aggregate in the view model layer; use the same instances for menus, toolbars, and tray commands so `CanExecute` state stays consistent.
- Add `KeyBinding` entries on the window so shortcuts remain active even when focus is inside a text box:

```
<Window.InputBindings>
  <KeyBinding Gesture="Ctrl+N" Command="{Binding AppCommands.New}"/>
```

```
  <KeyBinding Gesture="Ctrl+O" Command="{Binding AppCommands.Open}"/>
</Window.InputBindings>
```

1.2 Native menus and the macOS menu bar

`NativeMenu` exports menu metadata to the host OS when available (macOS, some Linux environments). Attach it to the `TopLevel` so Avalonia's native exporters keep it in sync with window focus.

```
public override void OnFrameworkInitializationCompleted()
{
    if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
    {
        var window = Services.GetRequiredService<MainWindow>();
        desktop.MainWindow = window;

        NativeMenu.SetMenu(window, BuildNativeMenu());
    }

    base.OnFrameworkInitializationCompleted();
}

private static NativeMenu BuildNativeMenu()
{
    var appMenu = new NativeMenu
    {
        new NativeMenuItem("About", (_, _) => Locator.Commands.ShowAbout.Execute(null)),
        new NativeMenuItemSeparator(),
        new NativeMenuItem("Quit", (_, _) => Locator.Commands.Exit.Execute(null))
    };

    var fileMenu = new NativeMenu
    {
        new NativeMenuItem("New", (_, _) => Locator.Commands.New.Execute(null))
        {
            Gesture = new KeyGesture(Key.N, KeyModifiers.Control)
        },
        new NativeMenuItem("Open...", (_, _) => Locator.Commands.Open.Execute(null))
    };

    return new NativeMenu
    {
        new NativeMenuItem("MyApp") { Menu = appMenu },
        new NativeMenuItem("File") { Menu = fileMenu }
    };
}
```

- `NativeMenuItem.Gesture` mirrors `MenuItem.HotKey` and feeds the OS accelerator tables.
- Use `NativeMenuBar` in XAML when you want markup control over the native bar:

```
<native:NativeMenuBar DockPanel.Dock="Top">
  <native:NativeMenuBar.Menu>
    <native:NativeMenu>
      <native:NativeMenuItem Header="My App">
        <native:NativeMenuItem Header="About" Command="{Binding AppCommands.ShowAbout}"/>
      </native:NativeMenuItem>
      <native:NativeMenuItem Header="File">
```

```
        <native:NativeMenuItem Header="New" Command="{Binding AppCommands.New}"/>
      </native:NativeMenuItem>
    </native:NativeMenu>
  </native:NativeMenuBar.Menu>
</native:NativeMenuBar>
```

1.3 Command state and routing

`MenuItem` observes `ICommand.CanExecute`. Use commands that publish notifications (`ReactiveCommand`, `DelegateCommand`) and call `RaiseCanExecuteChanged()` whenever state changes. Keep command instances long-lived (registered in DI or a singleton `AppCommands` class) so every menu, toolbar, context menu, and tray icon reflects the same enable/disable state.

2. Context menus and flyouts

Attach `ContextMenu` to items directly or via styles so each container gets the same commands:

```
<ListBox Items="{Binding Documents}" SelectedItem="{Binding SelectedDocument}">
  <ListBox.Styles>
    <Style Selector="ListBoxItem">
      <Setter Property="ContextMenu">
        <ContextMenu>
          <MenuItem Header="Rename"
                    Command="{Binding DataContext.Rename, RelativeSource={RelativeSource AncestorType=L
                    CommandParameter="{Binding}"/>
          <MenuItem Header="Delete"
                    Command="{Binding DataContext.Delete, RelativeSource={RelativeSource AncestorType=L
                    CommandParameter="{Binding}"/>
        </ContextMenu>
      </Setter>
    </Style>
  </ListBox.Styles>
</ListBox>
```

- `RelativeSource AncestorType=ListBox` bridges from the item container back to the list's data context.
- For richer layouts (toggles, sliders, forms) use `Flyout` or `MenuFlyout` – both live in `Avalonia.Controls` and share placement logic with context menus.
- Remember accessibility: set `MenuItem.InputGestureText` or `HotKey` so screen readers announce shortcuts.

3. Dialog pipelines

3.1 Define a dialog service interface

```
public interface IFileDialogService
{
    Task<IReadOnlyList<FilePickResult>> PickFilesAsync(FilePickerOpenOptions options, CancellationToken
    Task<FilePickResult?> SaveFileAsync(FilePickerSaveOptions options, CancellationToken ct = default);
    Task<IReadOnlyList<FilePickResult>> PickFoldersAsync(FolderPickerOpenOptions options, CancellationT
}

public record FilePickResult(string Path, IStorageItem? Handle);
```

Expose the service through dependency injection so view models request it instead of referencing `Window` or `TopLevel`.

3.2 Choose between `IStorageProvider`, `SystemDialog`, and `ManagedFileChooser`

`TopLevel.StorageProvider` supplies the native picker implementation (`IStorageProvider`). When it is unavailable (custom hosts, limited backends), fall back to the managed dialog stack built on `ManagedFileChooser`. The extension method `OpenFileDialog.ShowManagedAsync` renders the managed UI and is enabled automatically when you call `AppBuilder.UseManagedSystemDialogs()` during startup.

```
using Avalonia.Dialogs;
using Avalonia.Platform.Storage;

public sealed class FileDialogService : IFileDialogService
{
    private readonly TopLevel _topLevel;

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

    public async Task<IReadOnlyList<FilePickResult>> PickFilesAsync(FilePickerOpenOptions options, Canc
    {
        var provider = _topLevel.StorageProvider;
        if (provider is { CanOpen: true })
        {
            var files = await provider.OpenFilePickerAsync(options, ct);
            return files.Select(f => new FilePickResult(f.TryGetLocalPath() ?? f.Name, f)).ToArray();
        }

        if (_topLevel is Window window)
        {
            var dialog = new OpenFileDialog { AllowMultiple = options.AllowMultiple };
            var paths = await dialog.ShowManagedAsync(window, new ManagedFileDialogOptions());
            return paths.Select(p => new FilePickResult(p, handle: null)).ToArray();
        }

        return Array.Empty<FilePickResult>();
    }

    public async Task<FilePickResult?> SaveFileAsync(FilePickerSaveOptions options, CancellationToken c
    {
        var provider = _topLevel.StorageProvider;
        if (provider is { CanSave: true })
        {
            var file = await provider.SaveFilePickerAsync(options, ct);
            return file is null ? null : new FilePickResult(file.TryGetLocalPath() ?? file.Name, file);
        }

        if (_topLevel is Window window)
        {
            var dialog = new SaveFileDialog
            {
                DefaultExtension = options.DefaultExtension,
                InitialFileName = options.SuggestedFileName
            };
            var path = await dialog.ShowAsync(window);
            return path is null ? null : new FilePickResult(path, handle: null);
        }
```

```
        return null;
    }

    public async Task<IReadOnlyList<FilePickResult>> PickFoldersAsync(FolderPickerOpenOptions options, C
    {
        var provider = _topLevel.StorageProvider;
        if (provider is { CanPickFolder: true })
        {
            var folders = await provider.OpenFolderPickerAsync(options, ct);
            return folders.Select(f => new FilePickResult(f.TryGetLocalPath() ?? f.Name, f)).ToArray();
        }

        if (_topLevel is Window window)
        {
            var dialog = new OpenFolderDialog();
            var path = await dialog.ShowAsync(window);
            return path is null
                ? Array.Empty<FilePickResult>()
                : new[] { new FilePickResult(path, handle: null) };
        }

        return Array.Empty<FilePickResult>();
    }
}
```

- `OpenFileDialog`, `SaveFileDialog`, and `OpenFolderDialog` derive from `SystemDialog`. They remain useful when you need to force specific behaviour or when the platform lacks a proper storage provider.
- `AppBuilder.UseManagedSystemDialogs()` configures Avalonia to instantiate `ManagedFileChooser` by default whenever a native dialog is unavailable.
- Treat `FilePickResult.Handle` as optional: on browser/mobile targets you might only receive virtual URIs, while desktop gives full file system access.

4. Tray icons, notifications, and app commands

The tray API exports icons through the `Application`. Add them during application initialization so they follow the application lifetime automatically.

```
public override void Initialize()
{
    base.Initialize();

    if (ApplicationLifetime is not IClassicDesktopStyleApplicationLifetime)
        return;

    var trayIcons = new TrayIcons
    {
        new TrayIcon
        {
            Icon = new WindowIcon("avares://MyApp/Assets/App.ico"),
            ToolTipText = "My App",
            Menu = new NativeMenu
            {
                new NativeMenuItem("Show", (_, _) => Locator.Commands.ShowMain.Execute(null)),
                new NativeMenuItemSeparator(),
                new NativeMenuItem("Exit", (_, _) => Locator.Commands.Exit.Execute(null))
```

```
            }
        }
    };

    TrayIcon.SetIcons(this, trayIcons);
}
```

- Toggle `TrayIcon.IsVisible` in response to `Window` events to implement "minimize to tray". Guard the feature by checking `TrayIcon.SetIcons` only when running with a desktop lifetime.
- `NativeMenu` attached to a tray icon becomes the right-click menu. Reuse the same command implementations that power your primary menu to avoid duplication.
- Detect tray support by asking `AvaloniaLocator.Current.GetService<IWindowingPlatform>()?.CreateTrayIcon()` inside a try/catch before you rely on it.

In-app notifications come from `Avalonia.Controls.Notifications`:

```
using Avalonia.Controls.Notifications;

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

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

5. Top-level services and system integrations

`TopLevel` exposes cross-platform services you should wrap behind interfaces for testability:

```
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
}
```

Other helpful services on `TopLevel`: - `Screens` for multi-monitor awareness and DPI scaling. - `DragDrop` helpers (covered in Chapter 16) for integrating system drag-and-drop. - `TryGetFeature<T>` for platform-specific features (`ITrayIconImpl`, `IPlatformThemeVariant`).

6. Platform notes

- **Windows** – In-window `Menu` is standard. Tray icons appear in the notification area and expect `.ico` assets with multiple sizes. Native system dialogs are available; managed dialogs appear only if you opt in.
- **macOS** – Use `NativeMenu/NativeMenuBar` so menu items land in the global menu bar. Provide monochrome template tray icons via `MacOSProperties.SetIsTemplateIcon`.

- **Linux** – Desktop environments vary. Ship an in-window `Menu` even if you export a `NativeMenu`. Tray support may require AppIndicator or extensions.
- **Mobile (Android/iOS)** – Skip menu bars and tray icons. Replace them with toolbars, flyouts, and platform navigation. Storage providers surface document pickers that may not expose local file paths.
- **Browser** – No native menus or tray. Use in-app overlays and rely on the browser storage APIs (`BrowserStorageProvider`). Managed dialogs are not available.

7. Practice exercises

1. Build a shared `AppCommands` class that drives in-window menus, a `NativeMenu`, and a toolbar, verifying that `CanExecute` disables items everywhere.
2. Implement the dialog service above and log whether each operation used `IStorageProvider`, `SystemDialog`, or `ManagedFileChooser`. Run it on Windows, macOS, and Linux to compare behaviour.
3. Add a tray icon that toggles a "compact mode": closing the window hides it, the tray command re-opens it, and the tray menu reflects the current state.
4. Provide context menus for list items that reuse the same commands as the main menu. Confirm command parameters work for both entry points.
5. Surface toast notifications for long-running operations using `WindowNotificationManager`, and ensure they disappear automatically when the user navigates away.

Look under the hood (source bookmarks)

- Menus and native export: `Menu.cs`, `NativeMenu.Export.cs`
- Context menus & flyouts: `ContextMenu.cs`, `FlyoutBase.cs`
- Dialog infrastructure: `SystemDialog.cs`, `ManagedFileChooser.cs`
- Storage provider abstractions: `IStorageProvider.cs`
- Tray icons: `TrayIcon.cs`
- Notifications: `WindowNotificationManager.cs`

Check yourself

- How do `MenuItem` and `NativeMenuItem` share the same command instances, and why does that matter for `CanExecute`?
- When would you enable `UseManagedSystemDialogs`, and what UX differences should you anticipate compared to native dialogs?
- Which `TopLevel` services help you access storage, clipboard, and screens without referencing `Window` in view models?
- How can you detect tray icon availability before exposing tray-dependent features?
- What platform-specific adjustments do macOS and Linux require for menus and tray icons?

What's next - Next: Chapter 14

14. Lists, virtualization, and performance

Goal - Choose the right items control (`ItemsControl`, `ListBox`, `TreeView`, `DataGrid`, `ItemsRepeater`) for the data shape and user interactions you need. - Understand the `ItemsControl` pipeline (`ItemsSourceView`, item container generator, `ItemsPresenter`) and how virtualization keeps UIs responsive. - Apply virtualization techniques (`VirtualizingStackPanel`, `ItemsRepeater` layouts) alongside incremental loading and selection synchronization with `SelectionModel`. - Diagnose virtualization regressions using DevTools, logging, and layout instrumentation.

Why this matters - Lists power dashboards, log viewers, chat apps, and tables; poorly configured lists can freeze your UI. - Virtualization keeps memory and CPU usage manageable even with hundreds of thousands of rows. - Knowing the pipeline lets you extend list controls, add grouping, or inject placeholders without breaking performance.

Prerequisites - Binding and commands (Chapters 8–9), MVVM patterns (Chapter 11), styling and resources (Chapter 10).

Key namespaces - `ItemsControl.cs` - `ItemsSourceView.cs` - `ItemContainerGenerator.cs` - `VirtualizingStackPanel.cs` - `ItemsPresenter.cs` - `SelectionModel.cs` - `ItemsRepeater`

1. ItemsControl pipeline overview

Every items control follows the same data flow:

1. `Items`/`ItemsSource` is wrapped in an `ItemsSourceView` that projects the data as `IReadOnlyList<object?>`, tracks the current item, and provides grouping hooks.
2. `ItemContainerGenerator` materializes containers (`ListBoxItem`, `TreeViewItem`, etc.) for realized indices and recycles them when virtualization is enabled.
3. `ItemsPresenter` hosts the actual panel (by default `StackPanel` or `VirtualizingStackPanel`) and plugs into `ScrollViewer` to handle scrolling.
4. Templates render your view models inside each container.

Inspecting the view and generator helps when debugging:

```
var view = MyListBox.ItemsSourceView;
var current = view?.CurrentItem;

MyListBox.ItemContainerGenerator.Materialized += (_, e) =>
    Debug.WriteLine($"Realized range {e.StartIndex}..{e.StartIndex + e.Count - 1}");

MyListBox.ItemContainerGenerator.Dematerialized += (_, e) =>
    Debug.WriteLine($"Recycled {e.Count} containers");
```

Customize the items presenter when you need a different panel:

```
<ListBox Items="{Binding Orders}">
  <ListBox.ItemsPanel>
    <ItemsPanelTemplate>
      <VirtualizingStackPanel Orientation="Vertical"/>
    </ItemsPanelTemplate>
  </ListBox.ItemsPanel>
</ListBox>
```

`ItemsPresenter` can also be styled to add headers, footers, or empty-state placeholders while still respecting virtualization.

2. VirtualizingStackPanel in practice

`VirtualizingStackPanel` implements `ILogicalScrollable`, creating visuals only for the viewport (plus a configurable buffer). Keep virtualization intact by:

- Hosting the items panel directly inside a `ScrollViewer` (no extra wrappers between them).
- Avoiding nested `ScrollViewer`s inside item templates.
- Preferring fixed or predictable item sizes so layout calculations are cheap.

```
<ListBox Items="{Binding People}"
         SelectedItem="{Binding Selected}"
         Height="360"
         ScrollViewer.HorizontalScrollBarVisibility="Disabled">
  <ListBox.ItemsPanel>
    <ItemsPanelTemplate>
      <VirtualizingStackPanel Orientation="Vertical"
                              AreHorizontalSnapPointsRegular="True"
                              CacheLength="1"/>
    </ItemsPanelTemplate>
  </ListBox.ItemsPanel>
  <ListBox.ItemTemplate>
    <DataTemplate x:DataType="vm:PersonViewModel">
      <Grid ColumnDefinitions="Auto,*,Auto" Height="48" Margin="4">
        <TextBlock Grid.Column="0" Text="{CompiledBinding Id}" Width="56" HorizontalAlignment="Right"/>
        <StackPanel Grid.Column="1" Orientation="Vertical" Margin="12,0" Spacing="2">
          <TextBlock Text="{CompiledBinding FullName}" FontWeight="SemiBold"/>
          <TextBlock Text="{CompiledBinding Email}" FontSize="12" Foreground="#6B7280"/>
        </StackPanel>
        <Button Grid.Column="2"
                Content="Open"
                Command="{Binding DataContext.Open, RelativeSource={RelativeSource AncestorType=ListBox
                CommandParameter="{Binding}"/>
      </Grid>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

- `CacheLength` retains extra realized rows before and after the viewport (measured in viewport heights) for smoother scrolling.
- `ItemContainerGenerator.Materialized` events confirm virtualization: the count should remain small even with large data sets.
- Use `CompiledBinding` to avoid runtime reflection overhead when recycling containers.

3. Optimising item containers

Container recycling reuses realized `ListBoxItem` instances. Keep containers lightweight:

- Offload expensive visuals into shared `ControlTheme` resources.
- Style containers instead of adding extra elements for selection/hover state.

```
<Style Selector="ListBoxItem:selected TextBlock.title">
  <Setter Property="Foreground" Value="{DynamicResource AccentBrush}"/>
</Style>
```

When you need to interact with containers manually, use `ItemContainerGenerator.ContainerFromIndex/IndexFromContai` rather than walking the visual tree.

4. ItemsRepeater for custom layouts

`ItemsRepeater` separates data virtualization from layout so you can design custom grids or timelines.

```
<controls:ItemsRepeater Items="{Binding Photos}"
                        xmlns:controls="clr-namespace:Avalonia.Controls;assembly=Avalonia.Controls">
  <controls:ItemsRepeater.Layout>
    <controls:UniformGridLayout Orientation="Vertical" MinItemWidth="220" MinItemHeight="180"/>
  </controls:ItemsRepeater.Layout>
  <controls:ItemsRepeater.ItemTemplate>
    <DataTemplate x:DataType="vm:PhotoViewModel">
      <Border Margin="8" Padding="8" Background="#111827" CornerRadius="6">
        <StackPanel>
          <Image Source="{CompiledBinding Thumbnail}" Width="204" Height="128" Stretch="UniformToFill"/>
          <TextBlock Text="{CompiledBinding Title}" Margin="0,8,0,0"/>
        </StackPanel>
      </Border>
    </DataTemplate>
  </controls:ItemsRepeater.ItemTemplate>
</controls:ItemsRepeater>
```

- `ItemsRepeater.ItemsSourceView` exposes the same API as `ItemsControl`, so you can layer grouping or filtering on top.
- Implement a custom `VirtualizingLayout` when you need masonry or staggered layouts that still recycle elements.

5. Selection with `SelectionModel`

`SelectionModel<T>` tracks selection without relying on realized containers, making it virtualization-friendly.

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

Bind directly:

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

- `SelectionModel.SelectedItems` returns a snapshot of selected view models; use it for batch operations.
- Hook `SelectionModel.SelectionChanged` to synchronize selection with other views or persisted state.
- For custom surfaces (e.g., an `ItemsRepeater` dashboard), set `selectionModel.Source = repeater.ItemsSourceView` and drive selection manually.

6. Incremental loading patterns

Load data in pages to keep virtualization responsive. The view model owns the collection and exposes an async method that appends new items.

```
public sealed class LogViewModel : ObservableObject
{
    private readonly ILogService _service;
    private readonly ObservableCollection<LogEntryViewModel> _entries = new();
    private bool _isLoading;
    private int _pageIndex;
    private const int PageSize = 500;
```

```
    public LogViewModel(ILogService service)
    {
        _service = service;
        Entries = new ReadOnlyObservableCollection<LogEntryViewModel>(_entries);
        _ = LoadMoreAsync();
    }

    public ReadOnlyObservableCollection<LogEntryViewModel> Entries { get; }
    public bool HasMore { get; private set; } = true;

    public async Task LoadMoreAsync()
    {
        if (_isLoading || !HasMore)
            return;

        _isLoading = true;
        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;
        }
    }
}
```

Trigger loading when the user scrolls near the end:

```
private async void OnScrollChanged(object? sender, ScrollChangedEventArgs e)
{
    if (DataContext is LogViewModel vm &&
        vm.HasMore &&
        e.Source is ScrollViewer scroll &&
        scroll.Offset.Y + scroll.Viewport.Height >= scroll.Extent.Height - 200)
    {
        await vm.LoadMoreAsync();
    }
}
```

While loading, display lightweight placeholders (e.g., skeleton rows) bound to `IsLoading` flags; keep them inside the same template so virtualization still applies.

7. Diagnosing virtualization issues

When scrolling stutters or memory spikes:

- **DevTools Visual Tree**: select the list and open the **Diagnostics** tab to inspect realized item counts and virtualization mode.
- Enable layout/render logging:

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

- Monitor `ItemContainerGenerator.Materialized/Dematerialized` events; if counts climb with scroll distance, virtualization is broken.
- Verify the scroll host is the list's immediate parent; wrappers like `StackPanel` or `Grid` can disable virtualization.
- Profile templates with `dotnet-trace` or `dotnet-counters` to spot expensive bindings or allocations while scrolling.

8. Practice exercises

1. Inspect `ItemsControl.ItemsSourceView` for a dashboard list and log the current item index whenever selection changes. Explain how it differs from binding directly to `ItemsSource`.
2. Convert a slow `ItemsControl` to a virtualized `ListBox` with `VirtualizingStackPanel` and record container creation counts before/after.
3. Build an `ItemsRepeater` gallery with `UniformGridLayout` and compare realized item counts against a `WrapPanel` version.
4. Replace `SelectedItems` with `SelectionModel` in a multi-select list, then synchronize the selection with a detail pane while keeping virtualization intact.
5. Implement the incremental log viewer above, including skeleton placeholders during fetch, and capture frame-time metrics before and after the optimization.

Look under the hood (source bookmarks)

- Pipeline internals: `ItemsControl.cs`, `ItemContainerGenerator.cs`
- Data views: `ItemsSourceView.cs`, `CollectionView.cs`
- Virtualization core: `VirtualizingStackPanel.cs`, `VirtualizingLayout.cs`
- Selection infrastructure: `SelectionModel.cs`
- Diagnostics tooling: `LayoutDiagnosticBridge.cs`

Check yourself

- What distinguishes `ItemsSource` from `ItemsSourceView`, and when would you inspect the latter?
- How does `VirtualizingStackPanel` decide which containers to recycle, and what breaks that logic?
- Why does `SelectionModel` survive virtualization better than `SelectedItems`?
- Which DevTools views help you confirm virtualization is active?
- How can incremental loading keep long lists responsive without overwhelming the UI thread?

What's next - Next: Chapter 15

15. Accessibility and internationalization

Goal - Deliver interfaces that are usable with keyboard, screen readers, and high-contrast themes. - Implement automation metadata (`AutomationProperties`, custom `AutomationPeer`s) so assistive technologies understand your UI. - Localize content, formats, fonts, and layout direction for multiple cultures while supporting IME and text services. - Build a repeatable accessibility testing loop that spans platform tooling and automated checks.

Why this matters - Accessibility ensures compliance (WCAG/ADA) and a better experience for keyboard and assistive-technology users. - Internationalization widens your reach and avoids locale-specific bugs in formatting or layout direction. - Treating accessibility and localization as first-class requirements keeps your app portable across desktop, mobile, and browser targets.

Prerequisites - Keyboard input and commands (Chapter 9), resources (Chapter 10), MVVM patterns (Chapter 11), navigation and lifetimes (Chapter 12).

Key namespaces - `AutomationProperties.cs` - `AutomationPeer.cs` - `ControlAutomationPeer.cs` - `TextInputMethodClient.cs` - `TextInputOptions.cs` - `FontManagerOptions.cs` - `FlowDirection.cs`

1. Keyboard accessibility

1.1 Focus order and tab navigation

```
<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"` keeps focus within the container, ideal for dialogs.
- Use `AccessText` or `RecognizesAccessKey="True"` to expose mnemonic keys.
- Disable focus for decorative elements via `IsTabStop="False"` or `Focusable="False"`.

1.2 Keyboard navigation helpers

`KeyboardNavigation` (source: `KeyboardNavigation.cs`) provides: - `DirectionalNavigation="Cycle"` for arrow-key traversal in menus/panels. - `TabNavigation` modes (`Continue`, `Once`, `Local`, `Cycle`, `None`). - `Control.IsTabStop` per element when you need to skip items like labels or icons.

2. Screen reader semantics

Attach `AutomationProperties` to expose names, help text, and relationships:

```
<StackPanel Spacing="10">
  <TextBlock x:Name="EmailLabel" Text="Email"/>
```

```
  <TextBox Text="{Binding Email}"
           AutomationProperties.LabeledBy="{Binding #EmailLabel}"
           AutomationProperties.AutomationId="EmailInput"/>

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

- `AutomationProperties.Name` provides a fallback label when there is no visible text.
- `AutomationProperties.HelpText` supplies extra instructions for screen readers.
- `AutomationProperties.LiveSetting` (`Polite`, `Assertive`) controls how urgent announcements are.
- `AutomationProperties.ControlType` lets you override the role in edge cases (use sparingly).

`AutomationProperties` map to automation peers. The base `ControlAutomationPeer` inspects properties and pseudo-classes to expose state.

3. Custom automation peers

Create peers when you author custom controls so assistive technology can identify them correctly.

```
public class ProgressBadge : TemplatedControl
{
    public static readonly StyledProperty<string?> TextProperty =
        AvaloniaProperty.Register<ProgressBadge, string?>(nameof(Text));

    public string? Text
    {
        get => GetValue(TextProperty);
        set => SetValue(TextProperty, value);
    }

    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.Te
    protected override AutomationLiveSetting GetLiveSettingCore() => AutomationLiveSetting.Polite;
}
```

- Override `PatternInterfaces` (e.g., `IRangeValueProvider`, `IValueProvider`) when your control supports specific automation patterns.
- Use `AutomationProperties.AccessibilityView` to control whether a control appears in the content vs. control view.

4. High contrast and theme variants

Avalonia supports theme variants (`Light`, `Dark`, `HighContrast`). Bind colors to resources instead of hard-coding values.

```
<ResourceDictionary>
  <ResourceDictionary.ThemeDictionaries>
    <ResourceDictionary x:Key="Default">
      <SolidColorBrush x:Key="AccentBrush" Color="#2563EB"/>
    </ResourceDictionary>
    <ResourceDictionary x:Key="HighContrast">
      <SolidColorBrush x:Key="AccentBrush" Color="#00FF00"/>
    </ResourceDictionary>
  </ResourceDictionary.ThemeDictionaries>
</ResourceDictionary>
```

Switch variants for testing:

```
Application.Current!.RequestedThemeVariant = ThemeVariant.HighContrast;
```

Provide clear focus visuals using pseudo-classes (`:focus`, `:pointerover`) and ensure contrast ratios meet WCAG (4.5:1 for body text). For Windows, respect system accent colors by reading `RequestedThemeVariant` and `SystemBarColor` (Chapter 7).

5. Text input, IME, and text services

IME support matters for CJK languages and handwriting. `TextInputMethodClient` is the bridge between your control and platform IME surfaces. Text controls in Avalonia already implement it; custom text editors should derive from `TextInputMethodClient` (or reuse `TextPresenter`).

```
public sealed class CodeEditorTextInputClient : TextInputMethodClient
{
    private readonly CodeEditor _editor;

    public CodeEditorTextInputClient(CodeEditor editor) => _editor = editor;

    public override Visual TextViewVisual => _editor.TextLayer;
    public override bool SupportsPreedit => true;
    public override bool SupportsSurroundingText => true;
    public override string SurroundingText => _editor.Document.GetText();
    public override Rect CursorRectangle => _editor.GetCaretRect();
    public override TextSelection Selection
    {
        get => new(_editor.SelectionStart, _editor.SelectionEnd);
        set => _editor.SetSelection(value.Start, value.End);
    }

    public void UpdateCursor()
    {
        RaiseCursorRectangleChanged();
        RaiseSelectionChanged();
        RaiseSurroundingTextChanged();
    }
}
```

Configure text options with the attached `TextInputOptions` properties:

```
<TextBox Text="{Binding PhoneNumber}"
         InputMethod.TextInputOptions.ContentType="TelephoneNumber"
         InputMethod.TextInputOptions.ReturnKeyType="Done"
         InputMethod.TextInputOptions.IsCorrectionEnabled="False"/>
```

- On mobile, `ReturnKeyType` changes the soft keyboard button (e.g., “Go”, “Send”).

- `ContentType` hints at expected input, enabling numeric keyboards or email layouts.
- `IsContentPredictionEnabled`/`IsSpellCheckEnabled` toggle autocorrect.

When you detect IME-specific behaviour, test on Windows (IMM32), macOS, Linux (IBus/Fcitx), Android, and iOS — each backend surfaces slightly different capabilities.

6. Localization workflow

6.1 Resource management

Use RESX resources or a localization service that surfaces culture-specific strings.

```
public sealed class Loc : INotifyPropertyChanged
{
    private CultureInfo _culture = CultureInfo.CurrentUICulture;
    public string this[string key] => Resources.ResourceManager.GetString(key, _culture) ?? key;

    public void SetCulture(CultureInfo culture)
    {
        if (_culture.Equals(culture))
            return;

        _culture = culture;
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(null));
    }

    public event PropertyChangedEventHandler? PropertyChanged;
}
```

Register in `App.axaml` and bind:

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

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

Switch culture at runtime:

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

6.2 Formatting and layout direction

- Use binding `StringFormat` or `string.Format` with the current culture for dates, numbers, and currency.
- Set `FlowDirection="RightToLeft"` for RTL languages and override back to `LeftToRight` for controls that must remain LTR (e.g., numeric fields).
- Mirror icons and layout padding when mirrored (use `ScaleTransform` or `LayoutTransform`).

7. Fonts and fallbacks

Ensure glyph coverage with `FontManagerOptions`:

```
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" }
    }
})
.StartWithClassicDesktopLifetime(args);
```

- Ship branded fonts via `FontFamily="avares://MyApp/Assets/Fonts/Brand.ttf#Brand"`.
- Test scripts that require surrogate pairs (emoji, rare CJK ideographs) to ensure fallbacks load.
- On Windows, consider `TextRenderingMode` for clarity vs. smoothness.

8. Testing accessibility

Tips for a repeatable test loop:

- **Keyboard** – Tab through each screen, ensure focus indicators are visible, and verify shortcuts work.
- **Screen readers** – Use Narrator, NVDA, or JAWS on Windows; VoiceOver on macOS/iOS; TalkBack on Android; Orca on Linux. Confirm names, roles, and help text.
- **Automation tree** – Avalonia DevTools → **Automation** tab visualizes peers and properties.
- **Contrast** – Run `Accessibility Insights` (Windows), `Color Oracle`, or browser dev tools to verify contrast ratios.
- **Automated** – Combine `Avalonia.Headless` UI tests (Chapter 21) with assertions on `AutomationId` and localized content.

Document gaps (e.g., missing peers, insufficient contrast) and track them like any other defect.

9. Practice exercises

1. Annotate a settings page with `AutomationProperties.Name`, `HelpText`, and `AutomationId`; inspect the automation tree with DevTools and NVDA.
2. Derive a custom `AutomationPeer` for a progress pill control, exposing live updates and value patterns, then verify announcements in a screen reader.
3. Configure `TextInputOptions` for phone number input on Windows, Android, and iOS. Test with an IME (Japanese/Chinese) to ensure composition events render correctly.
4. Localize UI strings into two additional cultures (e.g., es-ES, ar-SA), toggle `FlowDirection`, and confirm mirrored layouts do not break focus order.
5. Set up `FontManagerOptions` with script-specific fallbacks and validate that Arabic, Cyrillic, and CJK text render without tofu glyphs.

Look under the hood (source bookmarks)

- Keyboard navigation: `KeyboardNavigation.cs`
- Automation metadata: `AutomationProperties.cs`, `ControlAutomationPeer.cs`
- Text input & IME: `TextInputMethodClient.cs`, `TextInputOptions.cs`
- Localization: `CultureInfoExtensions`, `RuntimePlatformServices`
- Font management: `FontManagerOptions.cs`
- Flow direction: `FlowDirection.cs`

Check yourself

- How do `AutomationProperties.LabeledBy` and `AutomationId` improve automated testing and screen reader output?

- When should you implement a custom `AutomationPeer`, and which patterns do you need to expose for value-based controls?
- Which `TextInputOptions` settings influence IME behaviour and soft keyboard layouts across platforms?
- How do you switch UI language at runtime and ensure both text and layout update correctly?
- Where do you configure font fallbacks to cover multiple scripts without shipping duplicate glyphs?

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`).

`IStorageProvider` exposes capability flags such as `CanOpen`, `CanSave`, and `CanPickFolder`. Check them before presenting commands so sandboxed targets (browser/mobile) can hide unsupported options. Dialog methods accept option records (`FilePickerOpenOptions`, `FolderPickerOpenOptions`, etc.) that describe filters, suggested locations, and tokens for continuing previous sessions.

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.

1.2 Launching files and URIs

`TopLevel.Launcher` gives access to `ILauncher`, which opens files, folders, or URIs using the platform shell (Finder, Explorer, default browser, etc.). Combine it with storage results to let users reveal files after saving.

```
var topLevel = TopLevel.GetTopLevel(control);
if (topLevel?.Launcher is { } launcher && file is not null)
{
    await launcher.LaunchFileAsync(file);
    await launcher.LaunchUriAsync(new Uri("https://docs.avaloniaui.net"));
}
```

Return values indicate whether the launch succeeded; fall back to in-app viewers when it returns false.

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. Bookmarks and persisted access

Some platforms revoke file permissions when your app suspends. If an `IStorageItem` reports `CanBookmark`, call `SaveBookmarkAsync()` and store the returned string (e.g., in preferences). Later, reopen it via `IStorageProvider.OpenFileBookmarkAsync`/`OpenFolderBookmarkAsync`.

```
var bookmarks = new Dictionary<string, string>();

if (file.CanBookmark)
{
    var bookmarkId = await file.SaveBookmarkAsync();
    if (!string.IsNullOrEmpty(bookmarkId))
        bookmarks[file.Path.ToString()] = bookmarkId;
}

var restored = await storage.OpenFileBookmarkAsync(bookmarkId);
```

Keep bookmarks updated when users revoke access. iOS and Android can throw when bookmarks expire—wrap calls in try/catch and ask users to reselect the folder. Desktop platforms typically return standard file paths, but bookmarks still help retain portal-granted access (e.g., Flatpak).

`IStorageItem.GetBasicPropertiesAsync()` exposes metadata (size, modified time) without opening streams—use it when building file browsers.

6. 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.

7. 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).

7.1 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.

7.2 Custom visuals and adorners

Wrap your layout in an `AdornerDecorator` and render drop cues while a drag is in progress. Toggle overlays in `DragEnter`/`DragLeave` handlers to show hit targets or counts.

```
private void OnDragEnter(object? sender, DragEventArgs e)
{
    _dropOverlay.IsVisible = true;
}

private void OnDragLeave(object? sender, RoutedEventArgs e)
{
    _dropOverlay.IsVisible = false;
}
```

You can also inspect `e.DragEffects` to switch icons (copy vs move) or reject unsupported formats with a custom message. For complex scenarios create a lightweight `Window` as a drag adorner so the pointer stays responsive on multi-monitor setups.

8. 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
}
```

8.1 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.

9. 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);
```

10. 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.

11. Practice exercises

1. Implement `IFileDialogService` and expose commands for Open, Save, and Pick Folder; update the UI with results.
2. Build a file manager pane that enumerates folders asynchronously, persists bookmarks for sandboxed platforms, and mirrors changes via drag/drop.
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) with a custom adorner overlay showing the item count.
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`
- Bookmarks & metadata: `IStorageItem`
- Picker options: `FilePickerOpenOptions`, `FilePickerSaveOptions`
- Drag/drop: `DragDrop.cs`, `DataObject.cs`
- Clipboard: `IClipboard`
- Launcher: `ILauncher`

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");
```

1.1 Dispatcher priorities

`DispatcherPriority` controls when queued work runs relative to layout, input, and rendering. Use `Dispatcher.UIThread.Post` with an explicit priority when you want work to wait until after animations or to run ahead of rendering.

```
Dispatcher.UIThread.Post(
    () => Notifications.Clear(),
    priority: DispatcherPriority.Background);

Dispatcher.UIThread.Post(
    () => Toasts.Enqueue(message),
    priority: DispatcherPriority.Input);
```

Avoid defaulting everything to `DispatcherPriority.Send` (synchronous) because it can starve input processing.

1.2 SynchronizationContext awareness

`DispatcherSynchronizationContext` is installed on the UI thread; async continuations captured there automatically hop back to Avalonia when you `await`. When running background tasks (e.g., unit tests or hosted services) ensure you resume on the UI thread by capturing the context:

```
var uiContext = SynchronizationContext.Current;

await Task.Run(async () =>
{
    var result = await LoadAsync(ct).ConfigureAwait(false);
    uiContext?.Post(_ => ViewModel.Result = result, null);
});
```

When you intentionally want to stay on a background thread, use `ConfigureAwait(false)` to avoid marshaling back.

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.

```
public interface INetworkStatusService
{
    IObservable<bool> ConnectivityChanges { get; }
}

public sealed class NetworkStatusService : INetworkStatusService
{
    public IObservable<bool> ConnectivityChanges { get; }

    public NetworkStatusService()
    {
        ConnectivityChanges = Observable
            .FromEventPattern<NetworkAvailabilityChangedEventHandler, NetworkAvailabilityEventArgs>(
```

```
                handler => NetworkChange.NetworkAvailabilityChanged += handler,
                handler => NetworkChange.NetworkAvailabilityChanged -= handler)
            .Select(args => args.EventArgs.IsAvailable)
            .StartWith(NetworkInterface.GetIsNetworkAvailable());
    }
}
```

Register different implementations per target in DI (`#if` or platform-specific partial classes). On mobile, back the observable with platform connectivity APIs; on WebAssembly, bridge to `navigator.onLine` via JS interop. View models can subscribe once and stay platform-agnostic.

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).

6.1 Orchestrating services across targets

For cross-platform apps, wrap periodic or startup work in services that plug into each lifetime. Example using `IHostedService` semantics:

```
public interface IBackgroundTask
{
    Task StartAsync(CancellationToken token);
    Task StopAsync(CancellationToken token);
}

public sealed class SyncBackgroundTask : IBackgroundTask
{
    private readonly IDataSync _sync;
    public SyncBackgroundTask(IDataSync sync) => _sync = sync;

    public Task StartAsync(CancellationToken token)
        => Task.Run(() => _sync.RunLoopAsync(token), token);

    public Task StopAsync(CancellationToken token)
        => _sync.StopAsync(token);
}

public static class BackgroundTaskExtensions
{
    public static void Attach(this IBackgroundTask task, IApplicationLifetime lifetime)
    {
        switch (lifetime)
        {
            case IClassicDesktopStyleApplicationLifetime desktop:
                desktop.Startup += async (_, _) => await task.StartAsync(CancellationToken.None);
                desktop.Exit += async (_, _) => await task.StopAsync(CancellationToken.None);
                break;
            case ISingleViewApplicationLifetime singleView when singleView.MainView is { } view:
                view.AttachedToVisualTree += async (_, _) => await task.StartAsync(CancellationToken.No
```

```
                view.DetachedFromVisualTree += async (_, _) => await task.StopAsync(CancellationToken.N
                break;
        }
    }
}
```

Desktop lifetimes expose `Startup`/`Exit`; single-view/mobile lifetimes expose `FrameworkInitializationCompleted`/`OnStoppe` Provide adapters per lifetime so the task implementation stays portable, and inject platform helpers (connectivity, storage) through interfaces.

7. Reactive event streams

`Observable.FromEventPattern` converts callbacks into composable streams. Combine it with `DispatcherScheduler.Current` (from System.Reactive) so observations switch back to the UI thread.

```
var pointerStream = Observable
    .FromEventPattern<PointerEventArgs>(handler => control.PointerMoved += handler,
                                        handler => control.PointerMoved -= handler)
    .Select(args => args.EventArgs.GetPosition(control))
    .Throttle(TimeSpan.FromMilliseconds(50))
    .ObserveOn(DispatcherScheduler.Current)
    .Subscribe(point => PointerPosition = point);

Disposables.Add(pointerStream);
```

This pattern keeps heavy processing (`Throttle`, network calls) off the UI thread while delivering results back in order. For view models, expose `IObservable<T>` properties and let the view subscribe using `ReactiveUI.WhenAnyValue` or manual subscriptions. `Disposables` here is a `CompositeDisposable` that you dispose when the view/control unloads.

8. 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.

9. 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.

10. 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. Transform pointer move events into an `Observable` pipeline with throttling and verify updates stay on the UI thread.
6. 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`
- Priorities & timers: `DispatcherPriority.cs`, `DispatcherTimer.cs`
- Lifetimes: `IClassicDesktopStyleApplicationLifetime`, `ISingleViewApplicationLifetime`
- 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. Desktop backends at a glance

Avalonia ships multiple desktop backends; `AppBuilder.UsePlatformDetect()` selects the correct platform at runtime. Understanding the differences helps when you tweak options or debug native interop.

Platform	Backend type	Namespace	Notes
Windows	`Win32Platform`	`Avalonia.Win32`	Win32 windowing with optional WinUI composition, ANGLE/OpenGL bridges, tray icon helpers.
Windows/macOS	`AvaloniaNativePlatform`	`Avalonia.Native`	Shared native host (AppKit on macOS). Used for windowless scenarios and for macOS desktop builds.
Linux (X11)	`X11Platform`	`Avalonia.X11`	Traditional X11 windowing; integrates with FreeDesktop protocols.
Linux portals	`FreeDesktopPlatform`	`Avalonia.FreeDesktop`	Supplements X11/Wayland with portal services (dialogs, notifications).

Startup options customize each backend:

```
AppBuilder.Configure<App>()
    .UsePlatformDetect()
    .With(new Win32PlatformOptions
    {
        RenderingMode = new[] { Win32RenderingMode.AngleEgl, Win32RenderingMode.Software },
        CompositionMode = new[] { Win32CompositionMode.WinUIComposition, Win32CompositionMode.Redirecti
        OverlayPopups = true
    })
    .With(new MacOSPlatformOptions
    {
        DisableDefaultApplicationMenuItems = false,
        ShowInDock = true
    })
```

```
    .With(new X11PlatformOptions
    {
        RenderingMode = new[] { X11RenderingMode.Glx, X11RenderingMode.Software },
        UseDBusMenu = true,
        WmClass = "MyAvaloniaApp"
    });
```

These options map to platform implementations in `Avalonia.Win32`, `Avalonia.Native`, and `Avalonia.X11`. Tune them when enabling extended client area, portals, or GPU interop.

2. 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
    });
}
```

3. 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).

4. 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.

5. 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.
};
```

6. Platform integration

6.1 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.
- `Win32PlatformOptions` exposes rendering toggles (`RenderingMode`, `CompositionMode`, `OverlayPopups`). Keep `Software` in the list as a fallback for Remote Desktop.
- Use `TryGetPlatformHandle()` to retrieve HWNDs when integrating with native libraries; avoid depending on internal `WindowImpl` types.

6.2 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.
- `MacOSPlatformOptions` lets you hide dock icons, disable the default menu items, or reuse an existing `NSApplication` delegate.

- Pair `AvaloniaNativeRenderingMode` with a `UseSkia` configuration so you always include `Software` fallback alongside `Metal`/`OpenGl` for older GPUs.

6.3 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).
- `X11PlatformOptions` controls GLX/EGL fallbacks, DBus menus, and IME support; pair it with Avalonia's FreeDesktop portal helpers when running inside Flatpak/Snap.
- Use `WmClass` (on `X11PlatformOptions`) to integrate with desktop launchers and icon themes.

7. Rendering & GPU selection

Avalonia renders through Skia; each backend exposes toggles for GPU acceleration and composition. Tune them to balance visuals versus compatibility.

Platform	Rendering options	When to change
Windows (`Win32PlatformOptions`)	`RenderingMode` (`AngleEgl`, `Wgl`, `Vulkan`, `Software`), `CompositionMode` (`WinUIComposition`, etc.), `GraphicsAdapterSelectionCallback`, `WinUICompositionBackdropCornerRadius`	Choose ANGLE + WinUI for blur effects, fall back to software for remote desktops, pick dedicated GPU in multi-adapter rigs.
macOS (`AvaloniaNativePlatformOptions`)	`RenderingMode` (Metal, OpenGL, Software)	Prefer Metal on modern macOS; include Software as fallback for virtual machines.
Linux (`X11PlatformOptions`)	`RenderingMode` (`Glx`, `Egl`, `Vulkan`, `Software`), `GlxRendererBlacklist`, `UseDBusMenu`, `UseDBusFilePicker`	Disable GLX on problematic drivers, force software when GPU drivers are unstable.

`UseSkia` accepts `SkiaOptions` for further tuning:

```
AppBuilder.Configure<App>()
    .UsePlatformDetect()
    .With(new SkiaOptions
    {
        MaxGpuResourceSizeBytes = 128 * 1024 * 1024, // cap VRAM usage
        UseOpacitySaveLayer = true
    })
    .UseSkia()
    .LogToTrace();
```

Inside a window you can inspect the actual implementation for diagnostics:

```
if (TryGetPlatformHandle() is { Handle: var hwnd, HandleDescriptor: "HWND" })
    Debug.WriteLine($"HWND: 0x{hwnd.ToInt64():X}");
```

Log area `Avalonia.Rendering.Platform` reports which backend was selected; capture it during startup when debugging GPU-related issues.

8. Packaging & deployment overview

- Windows: `dotnet publish -r win-x64 --self-contained` or MSIX via `dotnet publish /p:PublishTrimmed=false /p:WindowsPackageType=msix`. Bundle ANGLE DLLs (`libEGL.dll`, `libGLESv2.dll`) and `d3dcompiler_47.dll` when using GPU composition; ship `vc_redist` prerequisites for older OS versions.
- macOS: `.app` bundle; codesign and notarize for distribution (`dotnet publish -r osx-x64 --self-contained` followed by bundle packaging via Avalonia templates or scripts). Include `libAvaloniaNative.dylib`, ensure `Info.plist` declares `NSHighResolutionCapable`, and register custom URL schemes if you rely on `ILauncher`.
- Linux: produce .deb/.rpm, AppImage, or Flatpak; ensure dependencies (`libAvaloniaNative.so`, libSkia) are present. Flatpak portals rely on `xdg-desktop-portal`; declare it as a runtime dependency and verify DBus access so storage pickers keep working.

Reference docs: Avalonia publishing guide (docs/publish.md).

9. 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.

10. 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

11. 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. Log which backend (`Win32RenderingMode`, `X11RenderingMode`, etc.) starts under different option combinations and document the impact on transparency and input latency.
6. Create deployment artifacts: MSIX (Windows), .app (macOS), and AppImage/Flatpak (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`
- Backend options: `Win32PlatformOptions`, `AvaloniaNativePlatformExtensions`, `X11Platform.cs`
- Skia configuration: `SkiaOptions`

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?
- Which option sets control rendering fallbacks on Windows and Linux backends?

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 `AvaloniaActivity`, `AvaloniaApplication`, and `AvaloniaAppDelegate` lifetimes so your shared code boots correctly on each platform. - Integrate platform services (back button, clipboard, storage, notifications) while respecting safe areas, touch input, and trimming constraints.

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 hosting lifetimes, navigation, and platform service wiring.

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.

`dotnet new avalonia.app --multiplatform` creates the shared project plus heads (`MyApp.Android`, `MyApp.iOS`, optional `MyApp.Browser`). The Android head references `Avalonia.Android` (which contains `AvaloniaActivity` and `AvaloniaApplication`); the iOS head references `Avalonia.iOS` (which contains `AvaloniaAppDelegate`).

Keep trimming/linker settings in `Directory.Build.props` so shared code doesn't lose reflection-heavy ViewModels. Example additions:

```
<PropertyGroup>
  <TrimMode>partial</TrimMode>
  <IlcInvariantGlobalization>true</IlcInvariantGlobalization>
  <PublishTrimmed>true</PublishTrimmed>
</PropertyGroup>
```

Use `TrimmerRootAssembly` or `DynamicDependency` attributes if you depend on reflection-heavy frameworks (e.g., ReactiveUI). Test Release builds on devices early to catch linker issues.

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.

Hot reload: on Android, Rider/Visual Studio can use .NET Hot Reload against `MyApp.Android`. For XAML hot reload in Previewer, add `<ItemGroup><XamlIlAssemblyInfo>true</XamlIlAssemblyInfo></ItemGroup>` to the shared project and keep the head running via `dotnet build -t:Run`.

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` and mirror the same logic inside `AvaloniaAppDelegate` to react to swipe-back gestures on iOS.

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);
}
```

Touch input specifics: prefer gesture recognizers (`Tapped`, `DoubleTapped`, `PointerGestureRecognizer`) over mouse events, and test with real hardware—emulators may not surface haptics or multi-touch.

5. Platform head customization

5.1 Android head (MyApp.Android)

- `MainActivity.cs` inherits from `AvaloniaActivity`. Override `AppBuilder CustomizeAppBuilder(AppBuilder builder)` to inject logging or DI.
- `MyApplication.cs` can inherit from `AvaloniaApplication` to bootstrap services before the activity creates the view.
- `AndroidManifest.xml`: declare permissions (`INTERNET`, `READ_EXTERNAL_STORAGE`), orientation, minimum SDK.
- App icons/splash: `Resources/mipmap-*`, `Resources/xml/splashscreen.xml` for Android 12+ splash.
- Enable fast deployment/device hot reload by setting `<AndroidEnableProfiler>true</AndroidEnableProfiler>` in Debug configuration.
- Intercept hardware Back button by overriding `OnBackPressed` or using `AvaloniaLocator.Current.GetService<IMobil`

```
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. To embed Avalonia inside an existing native activity, host `AvaloniaView` inside a layout and call `AvaloniaView.Initialize(this, AppBuilder.Configure<App>()...)`.

5.2 iOS head (MyApp.iOS)

- `AppDelegate.cs` inherits from `AvaloniaAppDelegate`. Override `CustomizeAppBuilder` to inject services or register platform-specific singletons.
- `Program.cs` wraps `UIApplication.Main(args, null, typeof(AppDelegate))` so the delegate boots Avalonia.
- `Info.plist`: permissions (e.g., camera), orientation, status bar style.
- Launch screen via `LaunchScreen.storyboard` or SwiftUI resources in Xcode.
- Use `AvaloniaViewController` to embed Avalonia content inside UIKit navigation stacks or tab controllers.

Handle universal links or background tasks by bridging to shared services in `AppDelegate`. For swipe-back gestures, implement `TryGoBack` inside `AvaloniaNavigationController` or intercept `UINavigationControllerDelegate` callbacks.

5.3 Sharing services across heads

Inject platform implementations for `IClipboard`, `IStorageProvider`, notifications, and share targets via dependency injection. Register them in `AvaloniaLocator.CurrentMutable` inside `CustomizeAppBuilder` to keep shared code unaware of head-specific services.

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.
- Use `EssentialsPermissions` helper libraries carefully—Release builds with trimming must preserve permission classes. Validate by running `dotnet publish -c Release` on device/emulator.
- Push notifications and background fetch require platform services: expose custom interfaces (e.g., `IPushNotificationService`) that platform heads implement and inject into shared locator.

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.
- Keep layout responsive: use `TopLevel.Screen` to detect orientation/size classes and expose them via your view models.

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.
- Enable GPU frame stats with `adb shell dumpsys gfxinfo` or Xcode's Metal throughput counters to detect rendering bottlenecks.

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.
- Linker errors often show up only in Release; enable `--warnaserror` on linker warnings to catch missing assemblies early.

Optional: Tizen

Avalonia's Tizen backend (`Avalonia.Tizen`) targets smart TVs/wearables. The structure mirrors Android/iOS: implement a Tizen `Program.cs` that calls `AppBuilder.Configure<App>().UseTizen<TizenApplication>()` and handles platform storage/permissions via Tizen APIs.

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), validate icons/splash screens, and confirm Release trimming did not strip services.
6. (Stretch) Embed Avalonia inside a native screen (`AvaloniaView` on Android, `AvaloniaViewController` on iOS) and pass data between native and Avalonia layers.

Look under the hood (source bookmarks)

- Android hosting: `AvaloniaActivity`, `AvaloniaApplication`
- iOS hosting: `AvaloniaAppDelegate`, `AvaloniaViewController`
- Single-view lifetime: `SingleViewApplicationLifetime.cs`
- Insets and input pane: `IInsetsManager`, `IInputPane`
- Platform services: `AvaloniaLocator`, `IClipboard`
- Tizen backend: `Avalonia.Tizen`
- 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, DOM interop). - Debug, profile, 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, and understanding how Avalonia binds to the JS runtime keeps those differences manageable.

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. `MyApp.Browser` references `Avalonia.Browser`, which wraps the WebAssembly host (`BrowserAppBuilder`, `BrowserSingleViewLifetime`, `BrowserNativeControlHost`).

When adding the head manually, target `net8.0-browserwasm`, configure `<WasmMainJSPath>wwwroot/main.js</WasmMainJSP` and keep trimming hints (e.g., `<InvariantGlobalization>true</InvariantGlobalization>`). Browser heads use the NativeAOT toolchain; Release builds can set `<PublishAot>true</PublishAot>` for faster startup and smaller payloads.

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 `BrowserAppBuilder` directly:

```
await BrowserAppBuilder.Configure<App>()
    .SetupBrowserAppAsync(options =>
    {
        options.MainAssembly = typeof(App).Assembly;
        options.AppBuilder = AppBuilder.Configure<App>().LogToTrace();
```

```
        options.Selector = "#out";
    });
```

`SetupBrowserAppAsync` lets you delay instantiation (wait for configuration, auth, etc.) or mount multiple roots in different DOM nodes.

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).
- `PreferFileDialogPolyfill`: toggle between File System Access API and download/upload fallback for unsupported browsers.

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.
- `HttpClient` respects browser caching rules. Adjust `Cache-Control` headers or add cache-busting query parameters during development to avoid stale responses.

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.

To host native DOM content inside Avalonia, use `BrowserNativeControlHost` with a `JSObjectControlHandle`:

```
var handle = await JSRuntime.CreateControlHandleAsync("div", new { @class = "web-frame" });
var host = new BrowserNativeControlHost { Handle = handle };
```

This enables hybrid UI scenarios (rich HTML editors, video elements) while keeping sizing/layout under Avalonia control.

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. Hosting strategies

- Static hosting: publish bundle to `AppBundle` and serve from any static host (GitHub Pages, S3 + CloudFront, Azure Static Web Apps). Ensure service worker scope matches site root.
- ASP.NET Core: use `MapFallbackToFile("index.html")` or `UseBlazorFrameworkFiles()` to serve the bundle from a Minimal API or MVC backend.
- Reverse proxies: configure caching (Brotli, gzip) and set `Cross-Origin-Embedder-Policy/Cross-Origin-Opener-Poli` headers when enabling multithreaded WASM.

During development, `dotnet run` on the browser head launches a Kestrel server with live reload and proxies console logs back to the terminal.

11. Debugging and diagnostics

- 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.
- Pass `--logger:WebAssembly` to `dotnet run` for runtime messages (assembly loading, exception details).
- Use `wasm-tools wasm-strip` or `wasm-tools wasm-opt` (installed via `dotnet wasm build-tools --install`) to analyze and reduce bundle sizes.

12. Performance tips

- Measure download size: inspect `AppBundle`, track `.wasm`, `.dat`, and compressed assets.
- Prefer compiled bindings and avoid reflection-heavy converters to keep the IL linker effective.
- Enable multithreading (COOP/COEP headers) when animations or background tasks stutter; Avalonia will schedule the render loop on a dedicated worker thread.
- Integrate `BrowserSystemNavigationManager` with your navigation service so browser back/forward controls work as expected.

13. 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.

14. 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.

15. 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 and COOP/COEP headers.
6. Use `wasm-tools wasm-strip` (or `wasm-opt`) to inspect bundle size before/after trimming and record the change.

Look under the hood (source bookmarks)

- Browser app builder: `BrowserAppBuilder.cs`
- DOM interop: `JSObjectControlHandle.cs`
- Browser lifetime: `BrowserSingleViewLifetime.cs`
- Native control host: `BrowserNativeControlHost.cs`
- Storage provider: `BrowserStorageProvider.cs`
- System navigation manager: `BrowserSystemNavigationManager.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 and COOP/COEP headers?

What's next - Next: Chapter 21

21. Headless and testing

Goal - Test Avalonia UI components without a display server using `Avalonia.Headless` (`AvaloniaHeadlessPlatformExtensi - 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, automation, 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.

If you need Avalonia to drive the render loop before reading pixels, call `CaptureRenderedFrame()` instead of `GetLastRenderedFrame()`—it schedules a composition pass and forces a render tick. This mirrors what desktop renderers do when they flush the `CompositionTarget`, keeping the snapshot pipeline close to production.

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. Custom fixtures and automation hooks

- Build reusable fixtures around `HeadlessUnitTestSession.StartNew(typeof(App))` when you need deterministic startup logic outside the provided xUnit/NUnit attributes. Wrap it in `IAsyncLifetime` so tests share a dispatcher loop safely.
- Register platform services for tests inside the session by entering an `AvaloniaLocator` scope and injecting fakes (e.g., mock `IClipboard`, stub `IStorageProvider`).
- Expose convenience methods (e.g., `ShowControlAsync<TControl>()`) that create a `Window`, attach the control, call `ForceRenderTimerTick`, and return the control for assertions.
- For automation cues, use Avalonia's UI automation peers: call `AutomationPeer.CreatePeerForElement(control)` and assert patterns (`InvokePattern`, `ValuePattern`) without relying on visual tree traversal.
- Study the headless unit tests in `external/Avalonia/tests/Avalonia.Headless.UnitTests` for patterns that wrap `AppBuilder` and expose helpers for reuse across cases.

7. Advanced headless scenarios

7.1 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.

7.2 Simulating time & timers

Use `AvaloniaHeadlessPlatform.ForceRenderTimerTick()` to advance timers. For `DispatcherTimer` or animations, call it repeatedly.

7.3 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.

8. 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);
    }
}
```

9. 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).

10. 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.
6. Write an automation-focused test that inspects `AutomationPeer` patterns (Invoke/Value) to validate accessibility contracts alongside visual assertions.

Look under the hood (source bookmarks)

- Headless platform setup: `AvaloniaHeadlessPlatform.cs`
- Session control: `HeadlessUnitTestSession.cs`
- Input helpers: `HeadlessWindowExtensions`
- Test adapters: `Avalonia.Headless.XUnit`, `Avalonia.Headless.NUnit`
- Samples: `tests/Avalonia.Headless.UnitTests`, `tests/Avalonia.Headless.XUnit.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?
- When would you reach for AutomationPeers in headless tests instead of asserting on visuals alone?

What's next - Next: Chapter 22

22. Rendering pipeline in plain words

Goal - Understand how Avalonia turns your visual tree into frames across every backend. - Know the responsibilities of the UI thread, render loop, compositor, renderer, and GPU interface. - Learn how to tune rendering with `SkiaOptions`, `RenderOptions`, timers, 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. **Scene graph** represents visuals and draw operations in a batched form (`SceneGraph.cs`).
3. **Compositor** commits scene graph updates to the render thread and keeps track of dirty rectangles.
4. **Render loop** (driven by an `IRenderTimer`) asks the renderer to draw frames while work is pending.
5. **Renderer** walks the scene graph, issues drawing commands, and marshals them to Skia or another backend.
6. **Skia/render interface** rasterizes shapes/text/images into GPU textures (or CPU bitmaps) before the platform swapchain presents the frame.

Avalonia uses two main threads: UI thread and render thread. Keep the UI thread free of long-running work so animations, input dispatch, and composition stay 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 ships both `CompositingRenderer` (default) and `DeferredRenderer`. The renderer uses dirty rectangles to redraw minimal regions and produces scene graph nodes consumed by Skia.

CompositionTarget

`CompositionTarget` abstracts the surface being rendered. It holds references to swapchains, frame buffers, and frame timing metrics. You usually observe it through `IRenderer.Diagnostics` (frame times, dirty rect counts) or via DevTools/remote diagnostics rather than accessing the object directly.

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 the 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 the UI thread is momentarily busy.

Render timers

- `IRenderTimer` (see `IRenderTimer.cs`) abstracts ticking. Implementations include `DefaultRenderTimer`, `DispatcherRenderTimer`, and headless timers used in tests.
- Customize via `AppBuilder.UseRenderLoop(new RenderLoop(new DispatcherRenderTimer()))` to integrate external timing sources (e.g., game loops).
- Timers raise `Tick` on the render thread. Avoid heavy work in handlers: queue work through the UI thread if necessary.

Scene graph commits

Each `RenderLoop` tick calls `Compositor.CommitScenes`. The compositor transforms dirty visuals into render passes, prunes unchanged branches, and tracks retained GPU resources for reuse across frames.

5. Backend selection and GPU interfaces

Avalonia targets multiple render interfaces via `IRenderInterface`. Skia is the default implementation and chooses GPU versus CPU paths per platform.

Backend selection logic

- Desktop defaults to GPU (OpenGL/ANGLE on Windows, OpenGL/Vulkan on Linux, Metal on macOS).
- Mobile uses OpenGL ES (Android) or Metal (iOS/macOS Catalyst).
- Browser compiles Skia to WebAssembly and falls back to WebGL2/WebGL1/software.
- Server/headless falls back to CPU rendering.

Force a backend with `UseSkia(new SkiaOptions { RenderMode = RenderMode.Software })` or by setting `AVALONIA_RENDERER` environment variable (e.g., `software`, `open_gl`). Always pair overrides with tests on target hardware.

GPU resource management

- `SkiaOptions` exposes GPU cache limits and toggles like `UseOpacitySaveLayer`.
- `IRenderSurface` implementations (swapchains, framebuffers) own platform handles; leaks appear as rising `RendererDiagnostics.SceneGraphDirtyRectCount`.

Skia configuration

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. Frame timing instrumentation

Renderer diagnostics

- Enable `RendererDiagnostics` (see `RendererDiagnostics.cs`) via `RenderRoot.Renderer.Diagnostics`. Metrics include dirty rectangle counts, render phase durations, and draw call tallies.
- Pair diagnostics with `SceneInvalidated`/`RenderLoop` timestamps to push frame data into tracing systems such as `EventSource` or Prometheus exporters.

DevTools

- Press `F12` to open DevTools.
- `Diagnostics` panel toggles overlays and displays frame timing graphs.
- `Rendering` view (when available) shows render loop cadence, render thread load, and GPU backend in use.

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).
- `Avalonia.Diagnostics.RenderingDebugOverlays` integrates with `Avalonia.Remote.Protocol`. Use `avalonia-devtools://` clients to stream metrics from remote devices (Chapter 24).

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. Replace the render timer with a custom `IRenderTimer` implementation and graph frame cadence using timestamps collected from `SceneInvalidated`.
2. Override `SkiaOptions.RenderMode` to force software rendering, then switch back to GPU; profile render time using overlays in both modes.
3. Capture frame diagnostics (`RendererDebugOverlays.LayoutTimeGraph | RenderTimeGraph`) during an animation and export metrics for analysis.
4. Instrument `RenderRoot.Renderer.Diagnostics` to log dirty rectangle counts when toggling `InvalidateVisual`; correlate with DevTools overlays.
5. Use DevTools remote transport to attach from another process (Chapter 24) and verify frame timing matches local instrumentation.

Look under the hood (source bookmarks)

- Renderer interface: `IRenderer.cs`
- Compositor: `Compositor.cs`
- Scene graph: `RenderDataDrawingContext.cs`
- Immediate renderer: `ImmediateRenderer.cs`

- Render loop: `RenderLoop.cs`
- Render timer abstraction: `IRenderTimer.cs`
- Render options: `RenderOptions.cs`
- Skia options and platform interface: `SkiaOptions.cs`, `PlatformRenderInterface.cs`
- Renderer diagnostics: `RendererDiagnostics.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. Template lifecycle, presenters, and template results

- `TemplatedControl` raises `TemplateApplied` when the `ControlTemplate` is inflated. Override `OnApplyTemplate(TemplateAppliedEventArgs e)` to wire named parts via `e.NameScope`.

- Templates compiled from XAML return a `TemplateResult<Control>` behind the scenes (`ControlTemplate.Build`). It carries a `NameScope` so you can fetch presenters (`e.NameScope.Find<ContentPresenter>("PART_Content")`).
- Common presenters include `ContentPresenter`, `ItemsPresenter`, `ScrollContentPresenter`, and `ToggleSwitchPresenter`. They bridge templated surfaces with logical children (content, items, scrollable regions).
- Use `TemplateApplied` to subscribe to events on named parts, but always detach previous handlers before attaching new ones to prevent leaks.

Example:

```
protected override void OnApplyTemplate(TemplateAppliedEventArgs e)
{
    base.OnApplyTemplate(e);
    _toggleRoot?.PointerPressed -= OnToggle;
    _toggleRoot = e.NameScope.Find<Border>("PART_ToggleRoot");
    _toggleRoot?.PointerPressed += OnToggle;
}
```

For library-ready controls publish a `ControlTheme` default template so consumers can restyle without copying large XAML fragments.

5. 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).

6. Templated control example: Badge

Create `Badge : TemplatedControl` with properties (`Content`, `Background`, `Foreground`, `CornerRadius`, `MaxWidth`, etc.). Default style in `Styles.axaml`:

```
<ControlTheme TargetType="local:Badge">
  <Setter Property="Template">
    <ControlTemplate TargetType="local:Badge">
      <Border x:Name="PART_Border"
              Background="{TemplateBinding Background}"
              CornerRadius="{TemplateBinding CornerRadius}"
              Padding="6,0"
```

```
                MinHeight="16" MinWidth="20"
                HorizontalAlignment="Left"
                VerticalAlignment="Top">
          <ContentPresenter x:Name="PART_Content"
                            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"/>
</ControlTheme>
```

In code, capture named parts once the template applies:

```
public sealed class Badge : TemplatedControl
{
    public static readonly StyledProperty<object?> ContentProperty =
        AvaloniaProperty.Register<Badge, object?>(nameof(Content));

    Border? _border;

    public object? Content
    {
        get => GetValue(ContentProperty);
        set => SetValue(ContentProperty, value);
    }

    protected override void OnApplyTemplate(TemplateAppliedEventArgs e)
    {
        base.OnApplyTemplate(e);
        _border = e.NameScope.Find<Border>("PART_Border");
    }
}
```

Expose additional state through `StyledProperty`s so themes and animations can target them.

7. Visual states and control themes

- Use `PseudoClasses` (e.g., `PseudoClasses.Set(":badge-has-content", true)`) to signal template states that styles can observe.
- Combine `PseudoClasses` with `Transitions` or `Animations` to create hover/pressed effects without rewriting templates.
- Ship alternate appearances via additional `ControlTheme` resources referencing the same `TemplatedControl` type.
- For re-usable primitive parts, create internal `Visual` subclasses (e.g., `BadgeGlyph`) and expose them as named template parts.

8. Accessibility & input

- Set `Focusable` as appropriate; override `OnPointerPressed`/`OnKeyDown` for interaction or to update pseudo classes.
- Expose automation metadata via `AutomationProperties.Name`, `HelpText`, or custom `AutomationPeer` for drawn controls.
- Override `OnCreateAutomationPeer` when your control represents a unique semantic (`BadgeAutomationPeer` describing count, severity).

9. 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.

10. 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.

11. 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.

12. 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.

13. Practice exercises

1. Build a templated notification badge that swaps between “pill” and “dot” visuals by toggling `PseudoClasses` within `OnApplyTemplate`.
2. Embed a custom drawn sparkline into that badge (composed via `RenderTargetBitmap` or direct drawing) and expose it as a named part in the template.

3. Implement `OnCreateAutomationPeer` so assistive tech can report badge count and severity; verify with the accessibility tree in DevTools.
4. Use DevTools `Logical Tree` to confirm your presenter hierarchy (content vs drawn part) matches expectations and retains bindings when themes change.

Look under the hood (source bookmarks)

- Visual/render infrastructure: `Visual.cs`
- DrawingContext API: `DrawingContext.cs`
- StreamGeometry: `StreamGeometryContextImpl`
- Template loading: `ControlTemplate.cs`
- Template applied hook: `TemplateAppliedEventArgs.cs`
- Name scopes: `NameScope.cs`
- Templated control base: `TemplatedControl.cs`
- Control theme infrastructure: `ControlTheme.cs`
- Pseudo classes: `StyledElement.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. Renderer diagnostics API

- Every `TopLevel` exposes `IRenderer Diagnostics`. Subscribe to `PropertyChanged` to stream overlay toggles or other diagnostics to logs, counters, or dashboards.
- Toggle overlays without opening DevTools: set `renderer.Diagnostics.DebugOverlays` from code or configuration.
- Hook `SceneInvalidated` when you need per-frame insight into which rectangles triggered redraws. Pair this with your own timers to understand long layout/render passes.
- Enable `LogArea.Composition` in logging when you need to correlate compositor operations (scene graph updates, render thread work) with on-screen symptoms.

```
using System.Diagnostics;

if (TopLevel is { Renderer: { } renderer })
{
    renderer.SceneInvalidated += (_, e) =>
    {
        Debug.WriteLine($"Invalidated {e.Rect}");
    };

    renderer.Diagnostics.PropertyChanged += (_, e) =>
    {
        if (e.PropertyName == nameof(RendererDiagnostics.DebugOverlays))
        {
            Debug.WriteLine($"Overlays now: {renderer.Diagnostics.DebugOverlays}");
        }
    };
}
```

5. 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).

6. Remote diagnostics (`Avalonia.Remote.Protocol`)

- Remote DevTools streams frames and inspection data over the transports defined in `Avalonia.Remote.Protocol` (BSON/TCP by default).
- Use `Avalonia.Controls.Remote.RemoteServer` with a `BsonTcpTransport` to expose an interactive surface when debugging devices without a local inspector (mobile, kiosk). Connect using an Avalonia DevTools client (dotnet `avalonia` tool or IDE integration) pointing at `tcp-bson://host:port`.

- Messages such as `TransportMessages.cs` describe the payloads (frame buffers, input, diagnostics). Extend them if you build custom tooling.
- Remote sessions respect overlay and logging flags, so enabling `RendererDebugOverlays` locally will surface in the remote stream as well.
- For secure deployments, wrap `TcpTransportBase` in an authenticated tunnel (SSH port forward, reverse proxy) and disable remote servers in production builds.

7. 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).

8. 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.

9. 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).

10. 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.

11. Practice exercises

1. Attach DevTools, toggle `RendererDebugOverlays.Fps | LayoutTimeGraph`, and record metrics before/after enabling virtualization in a long list.

2. Capture binding noise by raising `LogArea.Binding` to `Debug`, then fix the source and verify the log stream quiets down.
3. Spin up a `RemoteServer` with `BsonTcpTransport`, connect using an Avalonia DevTools client (dotnet `avalonia` tool or IDE integration), and confirm overlays/logging data mirror the local session.
4. Profile the same interaction with `dotnet-trace` and align CPU spikes with render diagnostics to validate the chosen fix.

Look under the hood (source bookmarks)

- DevTools attach helpers: `DevToolsExtensions.cs`
- DevTools view models (toggling overlays): `MainViewModel.cs`
- Renderer diagnostics: `RendererDiagnostics.cs`
- Renderer overlays: `RendererDebugOverlays.cs`
- Logging infrastructure: `LogArea`
- RenderOptions (quality settings): `RenderOptions.cs`
- Layout diagnostics: `LayoutHelper`
- Remote transport messages: `TransportMessages.cs`
- Remote server host: `RemoteServer.cs`

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. Previewer pipeline and transport

IDE hosts spawn a preview process that loads your view or resource dictionary over the remote protocol. `DesignWindowLoader` spins up `RemoteDesignerEntryPoint`, which compiles your project with the design configuration, loads the control, then streams rendered frames back to the IDE through `Avalonia.Remote.Protocol.DesignMessages`.

Key components: - `Design.cs` toggles design mode (`Design.IsDesignMode`) and surfaces attached properties consumed only by the previewer. - `DesignWindowLoader` boots the preview process, configures the runtime XAML loader, and registers services. - `PreviewerWindowImpl` hosts the live surface, translating remote transport messages into frames. - `RemoteDesignerEntryPoint` sets up `RuntimeXamlLoader` and dependency injection so types resolve the same way they will at runtime.

Because the previewer compiles your project, build errors surface exactly as in `dotnet build`. Keep `AvaloniaResource` items and generated code in sync or the previewer will refuse to load.

2. Mock data with `Design.DataContext`

Provide lightweight POCOs or design view models for preview without touching production services.

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. When you need multiple sample contexts, expose them as static properties on a design-time provider class and bind with `{x:Static}`.

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. Inspect previewer logs and compilation errors

- Visual Studio and Rider show previewer logs in the dedicated "Avalonia Previewer" tool window; VS Code prints to the Output panel (`Avalonia Previewer` channel).
- Logs come from `DesignMessages`; look for `JsonRpcError` entries when bindings fail—those line numbers map to generated XAML.
- If compilation fails, open the temporary build directory path printed in the log. Running `dotnet build /p:Configuration=Design` replicates the preview build.
- Enable `Diagnostics -> Capture frames` to export a `.png` snapshot of the preview surface when you troubleshoot rendering glitches.

6. Extend design-time services

`RemoteDesignerEntryPoint` registers services in a tiny IoC container separate from your production DI. Override or extend them by wiring a helper that only executes when `Design.IsDesignMode` is true:

```
using Avalonia;
using Avalonia.Controls;

public static class DesignTimeServices
{
    public static void Register()
    {
        if (!Design.IsDesignMode)
            return;

        AvaloniaLocator.CurrentMutable
            .Bind<INavigationService>()
            .ToConstant(new FakeNavigationService());
    }
}
```

Call `DesignTimeServices.Register();` inside `BuildAvaloniaApp().AfterSetup(...)` so the previewer receives the fake services without altering production setup. Use this pattern to swap HTTP clients, repositories, or configuration with in-memory fakes while keeping runtime untouched.

7. 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.
- Enable verbose logs via `Previewer -> Options -> Enable Diagnostics` to capture transport traces when the preview window stays blank.

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.
- Rider caches preview assemblies under `%LOCALAPPDATA%/Avalonia`. Use "Invalidate caches" if you ship new resource dictionaries and the previewer shows stale data.

VS Code

- Avalonia `.vsix` extension hosts the previewer through the dotnet CLI; keep the extension and SDK workloads in sync.

- Run `dotnet workload install wasm-tools` (previewer uses WASM-hosted renderer). Use the `Avalonia Previewer: Show Log` command if the embedded browser surface fails.

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.

8. 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	`Design.*` stripped at runtime; if they leak, inspect generated `.g.cs` to confirm transformer ran
Resource dictionary preview fails	Add `Design.PreviewWith`; ensure resources compiled (check `AvaloniaResource` includes)
Sample data not showing	Confirm namespace mapping correct, sample object constructs without exceptions, and preview log shows `DataContext` attachment
Slow preview	Remove animations/effects temporarily; large data sets or virtualization can slow preview host
Transport errors (`SocketException`)	Restart previewer. Firewalls can block the loopback port used by `Avalonia.Remote.Protocol`

9. 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.
- Add an automated smoke test that loads critical views with `Design.IsDesignModeProperty` set to true via `RuntimeXamlLoader` to detect regressions before IDE users do.

10. 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. Open the previewer diagnostics window, reproduce a binding failure, and note how `DesignMessages` trace the failing binding path.
5. Guard service initialization with `if (Design.IsDesignMode)` and confirm preview load improves.
6. Bonus: implement a design-only service override and register it from `BuildAvaloniaApp().AfterSetup(...)`.

Look under the hood (source bookmarks)

- Design property helpers: `Design.cs`
- Preview transport wiring: `DesignWindowLoader.cs`
- Previewer bootstrapping: `RemoteDesignerEntryPoint.cs`
- Design-time property transformer: `AvaloniaXamlIlDesignPropertiesTransformer.cs`
- Previewer window implementation: `PreviewerWindowImpl.cs`
- Protocol messages: `Avalonia.Remote.Protocol/DesignMessages.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. Avalonia build tooling & project file essentials

Avalonia ships MSBuild targets that compile XAML and pack resources alongside your assemblies. Understanding them keeps design-time and publish-time behavior in sync.

- `CompileAvaloniaXamlTask` runs during `BeforeCompile` to turn `.axaml` into generated `.g.cs`. If a publish build reports missing generated files, confirm the Avalonia NuGet packages are referenced and the project imports `Avalonia.props/targets`.
- `AvaloniaResource` items embed static content. Include them explicitly so publish outputs contain everything:

```
<ItemGroup>
  <AvaloniaResource Include="Assets/**" />
  <AvaloniaResource Include="Themes/**/*.axaml" />
</ItemGroup>
```

- Shared targets such as `build/BuildTargets.targets` tweak platform packaging. Review them before overriding publish stages.
- Use property flags like `<AvaloniaUseCompiledBindings>true</AvaloniaUseCompiledBindings>` consistently across Debug/Release so the previewer and publish builds agree.
- For custom steps (version stamping, signing) extend `Target Name="AfterPublish"` or a Directory.Build.targets file; Avalonia emits its files before your target runs, so you can safely zip or notarize afterward.

3. 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).

4. 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

5. Output directories and manifest validation

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. Use `dotnet publish /bl:publish.binlog` and inspect the binlog with MSBuild Structured Log to confirm each resource path is copied.

- Check the generated `appsettings.json`, `MyApp.runtimeconfig.json`, and `.deps.json` to ensure trimming or single-file options didn't remove dependencies.
- For RID-specific bundles, review the platform manifests (e.g., `MyApp.app/Contents/Info.plist`, `MyApp.msix`) before shipping.

6. Asset packing and resources

- Group related resources into folders and wildcard them via `AvaloniaResource Include="Assets/Icons/**/*"`. The build task preserves folder structure when copying to publish output.

- Embedded assets larger than a few MB (videos, large fonts) can remain external files by setting `<CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>` alongside `AvaloniaResource`. That avoids bloating assemblies and keeps startup fast.
- When you refactor project layout, set explicit logical paths: `<AvaloniaResource Update="Assets/logo.svg" LogicalPath="resm:MyApp.Assets.logo.svg">`. Logical paths become the keys your app uses with `AssetLoader`.
- Hook a custom `Target Name="VerifyAvaloniaResources" AfterTargets="ResolveAvaloniaResource"` to ensure required files exist; failing early prevents subtle runtime crashes after publish.

7. 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.

8. Automation (CI/CD)

- Use GitHub Actions/Azure Pipelines/GitLab CI to run `dotnet publish` per target.
- Example GitHub Actions matrix aligned with Avalonia's build tasks:

```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'
      - name: Restore workloads
        run: dotnet workload restore
      - name: Publish
        run: |
          dotnet publish src/MyApp/MyApp.csproj \
            -c Release \
            -r ${{ matrix.rid }} \
            --self-contained true \
            /p:PublishSingleFile=true \
            /p:InformationalVersion=${{ github.sha }}
      - name: Collect binlog on failure
        if: failure()
        run: dotnet publish src/MyApp/MyApp.csproj -c Release -r ${{ matrix.rid }} /bl:publish-${{ matri
      - uses: actions/upload-artifact@v4
        with:
          name: myapp-${{ matrix.rid }}
          path: src/MyApp/bin/Release/net8.0/${{ matrix.rid }}/publish
      - name: Upload binlog
        if: failure()
        uses: actions/upload-artifact@v4
        with:
          name: publish-logs-${{ matrix.rid }}
          path: publish-${{ matrix.rid }}.binlog
```

- Add packaging steps (MSIX, DMG) via platform-specific actions/tools.
- Sign artifacts in CI where possible (store certificates securely).
- Azure Pipelines alternative: copy patterns from `azure-pipelines.yml` to reuse matrix publishing, signing, and artifact staging.
- For custom MSBuild integration, define `Target Name="SignArtifacts" AfterTargets="Publish"` or `AfterTargets="BundleApp"` in `Directory.Build.targets` so both local builds and CI run the same packaging hooks.

9. 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.
- Capture a SHA/hash of the publish folder (or installer) and include it in release notes so users can verify downloads.

10. 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.
`CompileAvaloniaXamlTask` failure	Clean `obj/`, fix XAML build errors, and examine the `/bl` binlog to inspect generated task arguments.
Native dependency failure (Skia/WASM)	Use `ldd`/`otool`/`wasm-ld` reports to list missing libraries; bundle them or switch to self-contained publishes.

11. 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; protect reflective types with `DynamicDependency` or `ILLinkTrim` descriptors and verify at runtime.
4. Set up a GitHub Actions workflow to publish RID-specific artifacts, collect binlogs on failure, and attach a checksum file.
5. Optional: create MSIX (Windows) or DMG (macOS) packages, sign them, and run locally to test installation/updates.
6. Bonus: add a custom MSBuild target that zips the publish folder and uploads a checksum to your CI artifacts.

Look under the hood (source & docs)

- Build docs: `docs/build.md`
- XAML compiler task: `CompileAvaloniaXamlTask.cs`
- Resource pipeline: `AvaloniaResource.cs`
- Shared targets: `build/BuildTargets.targets`
- Samples for packaging patterns: `samples/ControlCatalog`
- CI reference: `azure-pipelines.yml`
- .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 MSBuild tasks make sure `.axaml` files and resources reach your publish output?
- 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 highlights: - Core source: `src/` contains assemblies such as `Avalonia.Base`, `Avalonia.Controls`, `Avalonia.Markup.Xaml`, platform heads, and backend integrations (`Skia`, `WinUI`, browser). - Tests: `tests/` mixes unit tests, headless UI tests, integration tests, and rendering verification harnesses. Tests often reveal intended behavior and edge cases. - Samples: `samples/` hosts ControlCatalog, BindingDemo, and scenario-driven apps. They double as regression repros. - Tooling: `build/` and `build/NukeBuild` power CI, packaging, and developer workflows. - Docs: `docs/` complements the separate avalonia-docs site. - Contributor policy: `CONTRIBUTING.md`, coding conventions, and `.editorconfig` enforce consistent style (spacing, naming) across submissions.

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. - The repo also ships `build/NukeBuild`. Run `dotnet run --project build/NukeBuild` or `.\build.ps1 --target Test` to execute curated pipelines (`Compile`, `Test`, `Package`, etc.) identical to CI.

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. Testing strategy overview

Avalonia's quality gates rely on multiple test layers: - Unit tests (`tests/Avalonia.Base.UnitTests`, etc.) validate core property system, styling, and helper utilities. - Headless interaction tests (`tests/Avalonia.Headless.UnitTests`) simulate input/rendering without a visible window. - Integration/UI tests leverage the `Avalonia.IntegrationTests.Appium` harness for cross-platform smoke tests. - Performance benchmarks (look under `tests/Avalonia.Benchmarks`) measure layout, rendering, and binding regressions.

When contributing, prefer adding or updating the test nearest to the code you touch. For visual bugs, a headless interaction test plus a ControlCatalog sample usually gives maintainers confidence.

4. 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.

5. Debugging into Avalonia

- Enable symbol loading for Avalonia assemblies. NuGet packages ship SourceLink metadata—turn on "Load symbols from Microsoft symbol servers" (VS) or configure Rider's symbol caches so `.pdb` files download automatically.
- Add a fallback source path pointing at your local clone (`external/Avalonia/src`) to guarantee line numbers match when you build from source.
- 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.
- When debugging ControlCatalog or your own sample against a local build, reference the project outputs directly (`dotnet pack` + `nuget add source`) so you test the same bits you'll propose in a PR.

6. 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.

7. 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` (or `.\build.ps1 --target Test` / `nuke 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.

Doc-only or sample-only PRs

- Target `avalonia-docs` or `docs/` when API behavior changes. Reference the code PR in your documentation PR so reviewers can coordinate releases.
- For book/doc updates that do not touch runtime code, label the PR `Documentation` and mention "no runtime changes" in the description; CI can skip heavy legs when reviewers apply the label.
- Keep screenshots or GIFs small and check them into `docs/images/` or the appropriate sample folder. Update markdown links accordingly.

8. Docs & sample contributions

- Docs source: avalonia-docs repository. Preview the site locally with `npm install` + `npm start` to validate links before submitting.
- In-repo docs under `docs/` explain build and architecture topics; align book/new content with these guides.
- Samples: add new sample to `samples/` illustrating advanced patterns or new controls. Update `samples/README.md` when you add something new.
- Keep docs in sync with code changes for features/bug fixes and cross-link PRs so reviewers merge them together.

9. 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.

10. 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.

11. Practice exercises

1. Clone the Avalonia repo and run `.\build.ps1 --target Compile` (or `dotnet run --project build/NukeBuild Compile`). Verify the build succeeds and inspect the generated artifacts.
2. Launch ControlCatalog from the sample folder, then step into the code for one control you use frequently.
3. Configure symbol/source mapping in your IDE and step into `TextBlock` rendering while running ControlCatalog.
4. File a sample issue in a sandbox repo (practice minimal repro). Outline expected vs actual behavior clearly.
5. Write a headless unit test for a simple control (e.g., verifying a custom control draws expected output) and run it locally.
6. Draft a doc-only PR in `avalonia-docs` describing a workflow you improved (link back to the code sample or issue).

Look under the hood (source bookmarks)

- Repo root: github.com/AvaloniaUI/Avalonia
- Build scripts: `build.ps1`, `build.sh`
- NUKE entry point: `build/NukeBuild`
- Tests index: `tests/`
- Sample gallery: `samples/`
- 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?

- When would you reach for the NUKE build scripts instead of calling `dotnet build` directly?
- Which community channels help you stay informed about releases and roadmap?

What's next - Next: Chapter28

28. Advanced input system and interactivity

Goal - Coordinate pointer, keyboard, gamepad/remote, and text input so complex UI stays responsive. - Build custom gestures and capture strategies that feel natural across mouse, touch, and pen. - Keep advanced interactions accessible by mirroring behaviour across input modalities and IME scenarios.

Why this matters - Modern apps must work with touch, pen, mouse, keyboard, remotes, and assistive tech simultaneously. - Avalonia's input stack is highly extensible; understanding the pipeline prevents subtle bugs (ghost captures, lost focus, broken gestures). - When you marry gestures with automation, you avoid excluding keyboard- or screen-reader-only users.

Prerequisites - Chapter 9 (commands, events, and user input) for routed-event basics. - Chapter 15 (accessibility) to validate keyboard/automation parity. - Chapter 23 (custom controls) if you plan to surface bespoke surfaces that drive input directly.

1. How Avalonia routes input

Avalonia turns OS-specific events into a three-stage pipeline (`InputManager.ProcessInput`).

1. **Raw input** arrives as `RawInputEventArgs` (mouse, touch, pen, keyboard, gamepad). Each `IRenderRoot` has devices that call `Device.ProcessRawEvent`.
2. **Pre-process observers** (`InputManager.Instance?.PreProcess`) can inspect or cancel before routing. Use this sparingly for diagnostics, not business logic.
3. **Device routing** converts raw data into routed events (`PointerPressedEvent`, `KeyDownEvent`, `TextInputMethodClientRequestedEvent`).
4. **Process/PostProcess observers** see events after routing—handy for analytics or global shortcuts.

Because the input manager lives in `AvaloniaLocator`, you can temporarily subscribe:

```
using IDisposable? sub = InputManager.Instance?
    .PreProcess.Subscribe(raw => _log.Debug("Raw input {Device} {Type}", raw.Device, raw.RoutedEvent));
```

Remember to dispose subscriptions; the pipeline never terminates while the app runs.

2. Pointer fundamentals and event order

`InputElement` exposes pointer events (bubble strategy by default).

Event	Trigger	Key data
`PointerEntered` / `PointerExited`	Pointer crosses hit-test boundary	`Pointer.Type`, `KeyModifiers`, `Pointer.IsPrimary`
`PointerPressed`	Button/contact press	`PointerUpdateKind`, `PointerPointProperties`, `ClickCount` in `PointerPressedEventArgs`
`PointerMoved`	Pointer moves while inside or captured	`GetPosition`, `GetIntermediatePoints`
`PointerWheelChanged`	Mouse wheel / precision scroll	`Vector delta`, `PointerPoint.Properties`
`PointerReleased`	Button/contact release	`Pointer.IsPrimary`, `Pointer.Captured`
`PointerCaptureLost`	Capture re-routed, element removed, or pointer disposed	`PointerCaptureLostEventArgs.Pointer`

Event routing is tunable:

```
protected override void OnInitialized()
{
    base.OnInitialized();
    AddHandler(PointerPressedEvent, OnPreviewPressed, handledEventsToo: true);
    AddHandler(PointerPressedEvent, OnPressed, routingStrategies: RoutingStrategies.Tunnel | RoutingStra
}
```

Use tunnel handlers (`RoutingStrategies.Tunnel`) for global shortcuts (e.g., closing flyouts). Keep bubbling logic per control.

Working with pointer positions

- `e.GetPosition(this)` projects coordinates into any visual's space; pass `null` for top-level coordinates.
- `e.GetIntermediatePoints(this)` yields historical samples—crucial for smoothing freehand ink.
- `PointerPoint.Properties` exposes pressure, tilt, contact rectangles, and button states. Always verify availability (`Pointer.Type == PointerType.Pen` before reading pressure).

3. Pointer capture and lifetime handling

Capturing sends subsequent input to an element regardless of pointer location—vital for drags.

```
protected override void OnPointerPressed(PointerPressedEventArgs e)
{
    if (e.Pointer.Type == PointerType.Touch)
    {
        e.Pointer.Capture(this);
        _dragStart = e.GetPosition(this);
        e.Handled = true;
    }
}

protected override void OnPointerReleased(PointerReleasedEventArgs e)
{
    if (ReferenceEquals(e.Pointer.Captured, this))
    {
        e.Pointer.Capture(null);
        CompleteDrag(e.GetPosition(this));
        e.Handled = true;
    }
}
```

Key rules: - Always release capture (`Capture(null)`) on completion or cancellation. - Watch `PointerCaptureLost`—it fires if the element leaves the tree or another control steals capture. - Don't forget to handle the gesture recognizer case: if a recognizer captures the pointer, your control stops receiving `PointerMoved` events until capture returns. - When chaining capture up the tree (`Control` → `Window`), consider `e.Pointer.Capture(this)` in the top-level to avoid anomalies when children are removed mid-gesture.

4. Multi-touch, pen, and high-precision data

Avalonia assigns unique IDs per contact (`Pointer.Id`) and marks a primary contact (`Pointer.IsPrimary`). Keep per-pointer state in a dictionary:

```
private readonly Dictionary<int, PointerTracker> _active = new();

protected override void OnPointerPressed(PointerPressedEventArgs e)
{
```

```
    _active[e.Pointer.Id] = new PointerTracker(e.Pointer.Type, e.GetPosition(this));
    UpdateManipulation();
}

protected override void OnPointerReleased(PointerReleasedEventArgs e)
{
    _active.Remove(e.Pointer.Id);
    UpdateManipulation();
}
```

Pen-specific data lives in `PointerPoint.Properties`:

```
var sample = e.GetCurrentPoint(this);
float pressure = sample.Properties.Pressure; // 0-1
bool isEraser = sample.Properties.IsEraser;
```

Touch sends a contact rectangle (`ContactRect`) you can use for palm rejection or handle-size aware UI.

5. Gesture recognizers in depth

Two gesture models coexist:

1. **Predefined routed events** in `Avalonia.Input.Gestures` (`Tapped`, `DoubleTapped`, `RightTapped`). Attach with `Gestures.AddDoubleTappedHandler` or `AddHandler`.
2. **Composable recognizers** (`InputElement.GestureRecognizers`) for continuous gestures (pinch, pull-to-refresh, scroll).

To attach built-in recognizers:

```
GestureRecognizers.Add(new PinchGestureRecognizer
{
    // Your subclasses can expose properties via styled setters
});
```

Creating your own recognizer lets you coordinate multiple pointers and maintain internal state:

```
public class PressAndHoldRecognizer : GestureRecognizer
{
    public static readonly RoutedEvent<RoutedEventArgs> PressAndHoldEvent =
        RoutedEvent.Register<InputElement, RoutedEventArgs>(
            nameof(PressAndHoldEvent), RoutingStrategies.Bubble);

    public TimeSpan Threshold { get; set; } = TimeSpan.FromMilliseconds(600);

    private CancellationTokenSource? _hold;
    private Point _pressOrigin;

    protected override async void PointerPressed(PointerPressedEventArgs e)
    {
        if (Target is not Visual visual)
            return;

        _pressOrigin = e.GetPosition(visual);
        Capture(e.Pointer);

        _hold = new CancellationTokenSource();
        try
        {
```

```
            await Task.Delay(Threshold, _hold.Token);
            Target?.RaiseEvent(new RoutedEventArgs(PressAndHoldEvent));
        }
        catch (TaskCanceledException)
        {
            // Swallow cancellation when pointer moves or releases early.
        }
    }

    protected override void PointerMoved(PointerEventArgs e)
    {
        if (Target is not Visual visual || _hold is null || _hold.IsCancellationRequested)
            return;

        var current = e.GetPosition(visual);
        if ((current - _pressOrigin).Length > 8)
            _hold.Cancel();
    }

    protected override void PointerReleased(PointerReleasedEventArgs e) => _hold?.Cancel();
    protected override void PointerCaptureLost(IPointer pointer) => _hold?.Cancel();
}
```

Register the routed event (`PressAndHoldEvent`) on your control and listen just like other events. Note the call to `Capture(e.Pointer)` which also calls `PreventGestureRecognition()` to stop competing recognizers.

Manipulation gestures and inertia

Avalonia exposes higher-level manipulation data through gesture recognizers so you do not have to rebuild velocity tracking yourself.

- `ScrollGestureRecognizer` raises `ScrollGestureEventArgs` with linear deltas and velocities—ideal for kinetic scrolling or canvas panning.
- `PinchGestureRecognizer` produces `PinchEventArgs` that report scale, rotation, and centroid changes for zoom surfaces.
- `PullGestureRecognizer` keeps track of displacement against a threshold (`PullGestureRecognizer.TriggerDistance`) so you can drive pull-to-refresh visuals without reimplementing spring physics.
- Internally, each recognizer uses `VelocityTracker` to compute momentum; you can hook `GestureRecognizer.Completed` to project inertia with your own easing.

Attach event handlers directly on the recognizer when you need raw data:

```
var scroll = new ScrollGestureRecognizer();
scroll.Scroll += (_, e) => _viewport += e.DeltaTranslation;
scroll.Inertia += (_, e) => StartInertiaAnimation(e.Velocity);
GestureRecognizers.Add(scroll);
```

Manipulation events coexist with pointer events. Mark the gesture event as handled when you consume it so the default scroll viewer does not fight your logic. For custom behaviors (elastic edges, snap points), tune `ScrollGestureRecognizer.IsContinuous`, `ScrollGestureRecognizer.CanHorizontallyScroll`, and `ScrollGestureRecognizer.CanVerticallyScroll` to match your layout.

6. Designing complex pointer experiences

Strategies for common scenarios:

- **Drag handles on templated controls:** capture the pointer in the handle `Thumb`, raise a routed

`DragDelta` event, and update layout in response. Release capture in `PointerReleased` and `PointerCaptureLost`.
- **Drawing canvases:** store sampled points per pointer ID, use `GetIntermediatePoints` for smooth curves, and throttle invalidation with `DispatcherTimer` to keep the UI responsive.
- **Canvas panning + zooming:** differentiate gestures by pointer count—single pointer pans, two pointers feed `PinchGestureRecognizer` for zoom. Combine with `MatrixTransform` on the content.
- **Edge swipe or pull-to-refresh:** use `PullGestureRecognizer` with `PullDirection` to recognise deflection and expose progress to the view model.
- **Hover tooltips:** `PointerEntered` kicks off a timer, `PointerExited` cancels it; inspect `e.GetCurrentPoint(this).Properties.PointerUpdateKind` to ignore quick flicks.

Platform differences worth noting: - **Windows/macOS/Linux** share pointer semantics, but only touch-capable hardware raises `PointerType.Touch`. Guard pen-specific paths behind `Pointer.Type == PointerType.Pen` because Linux/X11 backends can omit advanced pen properties. - **Mobile backends** (Android/iOS) dispatch multi-touch contacts without a mouse concept; ensure commands have keyboard fallbacks if you reuse the view for desktop. - **Browser (WASM)** lacks raw access to OS cursors and some pen metrics; `PointerPoint.Properties.Pressure` may always be `1.0`. - **Tizen** requires declaring the `http://tizen.org/privilege/haptic` privilege before you can trigger haptics from pull or press gestures.

7. Keyboard navigation, focus, and shortcuts

Avalonia's focus engine is pluggable.

- Each `TopLevel` exposes a `FocusManager` (via `(this.GetVisualRoot() as IInputRoot)?.FocusManager`) that drives tab order (`TabIndex`, `IsTabStop`).
- `IKeyboardNavigationHandler` orchestrates directional nav; register your own implementation before building the app, e.g. `AvaloniaLocator.CurrentMutable.Bind<IKeyboardNavigationHandler>().ToSingleton<Cus`
- `XYFocus` attached properties override directional targets for gamepad/remote scenarios:

```
<StackPanel
    input:XYFocus.Up="{Binding ElementName=SearchBox}"
    input:XYFocus.NavigationModes="Keyboard,Gamepad" />
```

Key bindings complement commands without requiring specific controls:

```
KeyBindings.Add(new KeyBinding
{
    Gesture = new KeyGesture(Key.N, KeyModifiers.Control | KeyModifiers.Shift),
    Command = ViewModel.NewNoteCommand
});
```

`HotKeyManager` subscribes globally:

```
HotKeyManager.SetHotKey(this, KeyGesture.Parse("F2"));
```

Ensure the target control implements `ICommandSource` or `IClickableControl`; Avalonia wires the gesture into the containing `TopLevel` and executes the command or raises `Click`.

Ensure focus cues remain visible: call `NavigationMethod.Tab` when moving focus programmatically so keyboard users see an adorner.

8. Gamepad, remote, and spatial focus

When Avalonia detects non-keyboard key devices, it sets `KeyDeviceType` on key events. Use `FocusManager.GetFocusManager(this)?.Focus(elem, NavigationMethod.Directional, modifiers)` to respect D-Pad navigation.

Configure XY focus per visual:

Property	Purpose
`XYFocus.Up/Down/Left/Right`	Explicit neighbours when layout is irregular
`XYFocus.NavigationModes`	Enable keyboard, gamepad, remote individually
`XYFocus.LeftNavigationStrategy`	Choose default algorithm (closest edge, projection, navigation axis)

For dense grids (e.g., TV apps), set `XYFocus.NavigationModes="Gamepad,Remote"` and assign explicit neighbours to avoid diagonal jumps. Pair with `KeyBindings` for shortcuts like `Back` or `Menu` buttons on controllers (map gamepad keys via key modifiers on the key event).

Where hardware exposes haptic feedback (mobile, TV remotes), query the platform implementation with `TopLevel.PlatformImpl?.TryGetFeature<TFeature>()`. Some backends surface rumble/vibration helpers; when none are available, fall back gracefully so keyboard-only users are not blocked.

9. Text input services and IME integration

Text input flows through `InputMethod`, `TextInputMethodClient`, and `TextInputOptions`.

- `TextInputOptions` attached properties describe desired keyboard UI.
- `TextInputMethodClient` adapts a text view to IMEs (caret rectangle, surrounding text, reconversion).
- `InputMethod.GetIsInputMethodEnabled` lets you disable the IME for password fields.

Set options in XAML:

```
<TextBox
    Text=""
    input:TextInputOptions.ContentType="Email"
    input:TextInputOptions.ReturnKeyType="Send"
    input:TextInputOptions.ShowSuggestions="True"
    input:TextInputOptions.IsSensitive="False" />
```

When you implement custom text surfaces (code editors, chat bubbles):

1. Implement `TextInputMethodClient` to expose text range, caret rect, and surrounding text.
2. Handle `TextInputMethodClientRequested` in your control to supply the client.
3. Call `InputMethod.SetIsInputMethodEnabled(this, true)` and update the client's `TextViewVisual` so IME windows track the caret.
4. On geometry changes, raise `TextInputMethodClient.CursorRectangleChanged` so the backend updates composition windows.

Remember to honor `TextInputOptions.IsSensitive`—set it when editing secrets so onboard keyboards hide predictions.

10. Accessibility and multi-modal parity

Advanced interactions must fall back to keyboard and automation:

- Offer parallel commands (`KeyBindings`, buttons) for pointer-only gestures.
- When adding custom gestures, raise semantic routed events (e.g., `CopyRequested`) so automation peers can invoke them.
- Keep automation peers updated (`AutomationProperties.ControlType`, `AutomationProperties.IsControlElement`) when capture changes visual state.
- Respect `FocusManager` decisions—never suppress focus adorners merely because a pointer started the interaction.
- Use `InputMethod.SetIsInputMethodEnabled` and `TextInputOptions` to support assistive text input (switch control, dictation).

11. Multi-modal input lab (practice)

Create a playground that exercises every surface:

1. **Project setup**: scaffold `dotnet new avalonia.mvvm -n InputLab`. Add a `CanvasView` control hosting drawing, a side panel for logs, and a bottom toolbar.
2. **Pointer canvas**: capture touch/pen input, buffer points per pointer ID, and render trails using `DrawingContext.DrawGeometry`. Display pressure as stroke thickness.
3. **Custom gesture**: add the `PressAndHoldRecognizer` (above) to show context commands after 600 ms. Hook the resulting routed event to toggle a radial menu.
4. **Pinch & scroll**: attach `PinchGestureRecognizer` and `ScrollGestureRecognizer` to pan/zoom the canvas. Update a `MatrixTransform` as gesture delta arrives.
5. **Keyboard navigation**: define `KeyBindings` for `Ctrl+Z`, `Ctrl+Shift+Z`, and arrow-key panning. Update `XYFocus` properties so D-Pad moves between toolbar buttons.
6. **Gamepad test**: using a controller or emulator, verify focus flows across the UI. Log `KeyDeviceType` in `KeyDown` to confirm Avalonia recognises it as Gamepad.
7. **IME sandbox**: place a chat-style `TextBox` with `TextInputOptions.ReturnKeyType="Send"`, plus a custom `MentionTextBox` implementing `TextInputMethodClient` to surface inline completions.
8. **Accessibility pass**: ensure every action has a keyboard alternative, set automation names on dynamically created controls, and test the capture cycle with screen reader cursor.
9. **Diagnostics**: subscribe to `InputManager.Instance?.Process` and log pointer ID, update kind, and capture target into a side list for debugging.

Document findings in README (which gestures compete, how capture behaves on focus loss) so the team can adjust default UX.

12. Troubleshooting & best practices

- **Missing pointer events**: ensure `IsHitTestVisible` is true and that no transparent sibling intercepts input. For overlays, set `IsHitTestVisible="False"`.
- **Stuck capture**: always release capture during `PointerCaptureLost` and when the control unloads. Wrap capture in `try/finally` on operations that may throw.
- **Gesture conflicts**: call `e.PreventGestureRecognition()` when manual pointer logic should trump recognizers—or avoid attaching recognizers to nested elements.
- **High-DPI offsets**: convert to screen coordinates using `Visual.PointToScreen` when working across popups; pointer positions are per-visual, not global.
- **Keyboard focus lost after drag**: store `(this.GetVisualRoot() as IInputRoot)?.FocusManager?.GetFocusedEle` before capture and restore it when the operation completes to preserve keyboard flow.
- **IME composition rectangles misplaced**: update `TextInputMethodClient.TextViewVisual` whenever layout changes; failing to do so leaves composition windows floating in the old position.

Look under the hood (source bookmarks)

- Pointer lifecycle: `Pointer.cs`
- Pointer events & properties: `PointerEventArgs.cs`, `PointerPoint.cs`
- Gesture infrastructure: `GestureRecognizer.cs`, `Gestures.cs`
- Continuous gestures: `ScrollGestureRecognizer.cs`, `PinchGestureRecognizer.cs`, `PullGestureRecognizer.cs`
- Keyboard & XY navigation: `IKeyboardNavigationHandler.cs`, `XYFocus.Properties.cs`
- Device data: `KeyEventArgs.cs`, `KeyDeviceType.cs`, `TouchDevice.cs`, `PenDevice.cs`
- Text input pipeline: `TextInputOptions.cs`, `TextInputMethodManager.cs`
- Input manager stages: `InputManager.cs`

Check yourself

- How do tunnelling handlers differ from bubbling handlers when mixing pointer capture and gestures?

- Which `PointerPointProperties` matter for pen input and how do you guard against unsupported platforms?
- What steps are required to surface a custom `TextInputMethodClient` in your control?
- How can you ensure a drag interaction remains keyboard-accessible?
- When would you replace the default `IKeyboardNavigationHandler`?

What's next - Next: Chapter29

29. Animations, transitions, and composition

Goal - Shape motion with Avalonia's keyframe animations, property transitions, and composition effects. - Decide when to stay in the styling layer versus dropping to the compositor for GPU-driven effects. - Orchestrate smooth navigation and reactive UI feedback without sacrificing performance.

Why this matters - Motion guides attention, expresses hierarchy, and communicates state changes; Avalonia gives you several layers to accomplish that. - Choosing the right animation surface (XAML, transitions, or composition) avoids wasted CPU, jank, and hard-to-maintain code. - Composition unlocks scenarios—material blurs, connected animations, fluid navigation—that are hard to express with traditional rendering.

Prerequisites - Chapter 22 (Rendering pipeline) for the frame loop and renderer semantics. - Chapter 23 (Custom drawing) for custom visuals that you might animate. - Chapter 8 (Data binding) for reactive triggers, and Chapter 24 (Diagnostics) for measuring performance.

1. Keyframe animation building blocks

Avalonia's declarative animation stack lives in `Avalonia.Animation.Animation` and friends. Every control derives from `Animatable`, so you can plug animations into styles or run them directly in code.

Concept	Type	Highlights
Timeline	`Animation` (`Animation.cs`)	`Duration`, `Delay`, `IterationCount`, `PlaybackDirection`, `FillMode`, `SpeedRatio`
Track	`KeyFrame` (`KeyFrames.cs`)	Specifies a cue (`0%..100%`) with one or more `Setter`s
Interpolation	`Animator<T>` (`Animators/DoubleAnimator.cs`, etc.)	Avalonia ships animators for primitives, transforms, brushes, shadows
Easing	`Easing` (`Animation/Easings/*`)	Over 30 easing curves, plus `SplineEasing` for custom cubic Bezier
Clock	`IClock` / `Clock` (`Clock.cs`)	Drives animations, default is the global clock

A minimal style animation:

```
<Window xmlns="https://github.com/avaloniaui">
  <Window.Styles>
    <Style Selector="Rectangle.alert">
      <Setter Property="Fill" Value="Red"/>
      <Style.Animations>
        <Animation Duration="0:0:0.6"
                   IterationCount="INFINITE"
                   PlaybackDirection="Alternate">
          <KeyFrame Cue="0%">
            <Setter Property="Opacity" Value="0.4"/>
            <Setter Property="RenderTransform.ScaleX" Value="1"/>
            <Setter Property="RenderTransform.ScaleY" Value="1"/>
          </KeyFrame>
          <KeyFrame Cue="100%">
            <Setter Property="Opacity" Value="1"/>
            <Setter Property="RenderTransform.ScaleX" Value="1.05"/>
```

```
            <Setter Property="RenderTransform.ScaleY" Value="1.05"/>
          </KeyFrame>
        </Animation>
      </Style.Animations>
    </Style>
  </Window.Styles>
</Window>
```

Key points: - `Animation.IterationCount="INFINITE"` loops forever; avoid pairing with `Animation.RunAsync` (throws by design). - `FillMode` controls which keyframe value sticks before/after the timeline. Use `FillMode="Both"` for a resting value. - You can scope animations to a resource dictionary and reference them by `{StaticResource}` from templates or code.

2. Controlling playback from code

`Animation.RunAsync` and `Animation.Apply` let you start, await, or conditionally run animations from code-behind or view models (`Animation.cs`, `RunAsync`).

```
public class ToastController
{
    private readonly Animation _slideIn;
    private readonly Animation _slideOut;
    private readonly Border _host;

    public ToastController(Border host, Animation slideIn, Animation slideOut)
    {
        _host = host;
        _slideIn = slideIn;
        _slideOut = slideOut;
    }

    public async Task ShowAsync(CancellationToken token)
    {
        await _slideIn.RunAsync(_host, token); // awaits completion
        await Task.Delay(TimeSpan.FromSeconds(3), token);
        await _slideOut.RunAsync(_host, token); // reuse the same host, different cues
    }
}
```

Behind the scenes `RunAsync` applies the animation with an `IClock` (defaults to `Clock.GlobalClock`) and completes when the last animator reports completion. Create the `_slideOut` animation by cloning `_slideIn`, switching its cues, or temporarily setting `PlaybackDirection = PlaybackDirection.Reverse` before calling `RunAsync`.

Reactive triggers map easily to animations by using `Apply(control, clock, IObservable<bool> match, Action onComplete)`:

```
var animation = (Animation)Resources["HighlightAnimation"];
var match = viewModel.WhenAnyValue(vm => vm.IsDirty);
var subscription = animation.Apply(border, null, match, null);
_disposables.Add(subscription);
```

- The observable controls when the animation should run (`true` pulses start it, `false` cancels).
- Supply your own `Clock` to coordinate multiple animations (e.g., `new Clock(globalClock)` with `PlayState.Pause` to scrub).
- Use the cancellation overload to stop animating when the control unloads or the view model changes.

3. Implicit transitions and styling triggers

For property tweaks (hover states, theme switches) `Animatable.Transitions` (`Animatable.cs`) is lighter weight than keyframes. A `Transition<T>` blends from the old value to a new one automatically.

```
<Button Classes="primary">
  <Button.Transitions>
    <Transitions>
      <DoubleTransition Property="Opacity" Duration="0:0:0.150"/>
      <TransformOperationsTransition Property="RenderTransform" Duration="0:0:0.200"/>
    </Transitions>
  </Button.Transitions>
</Button>
```

Rules of thumb: - Transitions cannot target direct properties (validation happens in `Transitions.cs`). Use styled properties or wrappers. - Attach them at the control level (`Button.Transitions`) or in a style (`<Setter Property="Transitions">`). - Combine with selectors to drive implicit animation from pseudo-classes:

```
<Style Selector="Button:pointerover">
  <Setter Property="Opacity" Value="1"/>
  <Setter Property="RenderTransform">
    <Setter.Value>
      <ScaleTransform ScaleX="1.02" ScaleY="1.02"/>
    </Setter.Value>
  </Setter>
</Style>
```

When the property switches, the matching `Transition<T>` eases between the two values. Avalonia ships transitions for numeric types, brushes, thickness, transforms, box shadows, and more (`Animation/Transitions/*.cs`).

Animator-driven transitions

`AnimatorDrivenTransition` lets you reuse keyframe logic as an implicit transition. Add an `Animation` to `Transition` by setting `Property` and plugging a custom `Animator<T>` if you need non-linear interpolation or multi-stop blends.

4. Page transitions and content choreography

Navigation surfaces (`TransitioningContentControl`, `Frame`, `NavigationView`) rely on `IPageTransition` (`PageSlide.cs`, `CrossFade.cs`).

```
<TransitioningContentControl Content="{Binding CurrentPage}">
  <TransitioningContentControl.PageTransition>
    <CompositePageTransition>
      <CompositePageTransition.PageTransitions>
        <PageSlide Duration="0:0:0.25" Orientation="Horizontal" Offset="32"/>
        <CrossFade Duration="0:0:0.20"/>
      </CompositePageTransition.PageTransitions>
    </CompositePageTransition>
  </TransitioningContentControl.PageTransition>
</TransitioningContentControl>
```

- `PageSlide` shifts content in/out; set `Orientation` and `Offset` to control direction.
- `CrossFade` fades the outgoing and incoming visuals.
- Compose transitions with `CompositePageTransition` to layer multiple effects.

- Listen to `TransitioningContentControl.TransitionCompleted` to dispose view models or preload the next page.

For navigation stacks, pair page transitions with parameterized view-model lifetimes so you can cancel transitions on route changes (`TransitioningContentControl.cs`).

5. Reactive animation flows

Because each animation pipes through `IObservable<bool>` internally, you can stitch motion into reactive pipelines:

- `match` observables allow gating by business rules (focus state, validation errors, elapsed time).
- Use `Animation.Apply(control, clock, observable, onComplete)` to bind to `WhenAnyValue`, `Observable.Interval`, or custom subjects.
- Compose animations: the returned `IDisposable` unsubscribes transitions when your view deactivates (critical for `Animatable.DisableTransitions`).

Example: flash a text box when validation fails, but only once every second.

```
var throttle = validationFailures
    .Select(_ => true)
    .Throttle(TimeSpan.FromSeconds(1))
    .StartWith(false);
animation.Apply(textBox, null, throttle, null);
```

6. Composition vs classic rendering

Avalonia's compositor (`Compositor.cs`) mirrors the Windows Composition model: a scene graph of `CompositionVisual` objects runs on a dedicated thread and talks directly to GPU backends. Advantages:

- Animations stay smooth even when the UI thread is busy.
- Effects (blur, shadows, opacity masks) render in hardware.
- You can build visuals that never appear in the standard logical tree (overlays, particles, diagnostics).

Getting the compositor:

```
var elementVisual = ElementComposition.GetElementVisual(myControl);
var compositor = elementVisual?.Compositor;
```

You can inject custom visuals under an existing control:

```
var compositor = ElementComposition.GetElementVisual(host)!.Compositor;
var root = ElementComposition.GetElementVisual(host) as CompositionContainerVisual;

var sprite = compositor.CreateSolidColorVisual();
sprite.Color = Colors.DeepSkyBlue;
sprite.Size = new Vector2((float)host.Bounds.Width, 4);
sprite.Offset = new Vector3(0, (float)host.Bounds.Height - 4, 0);
root!.Children.Add(sprite);
```

When mixing visuals, ensure they come from the same `Compositor` instance (`ElementCompositionPreview.cs`).

Composition target and hit testing

`CompositionTarget` (`CompositionTarget.cs`) owns the visual tree that the compositor renders. It handles hit testing, coordinate transforms, and redraw scheduling. Most apps use the compositor implicitly via the built-in renderer, but custom hosts (e.g., embedding Avalonia) can create their own target (`Compositor.CreateCompositionTarget`).

Composition brushes, effects, and materials

The compositor supports more than simple solids:

- `CompositionColorBrush` and `CompositionGradientBrush` mirror familiar WPF/UWP concepts and can be animated directly on the render thread.
- `CompositionEffectBrush` applies blend modes and image effects defined in `Composition.Effects`. Use it to build blur/glow pipelines without blocking the UI thread.
- `CompositionExperimentalAcrylicVisual` ships a ready-made fluent-style acrylic material. Combine it with backdrop animations for frosted surfaces.
- `CompositionDrawListVisual` lets you record drawing commands once and replay them efficiently; great for particle systems or dashboards.

Use `Compositor.TryCreateBlurEffect()` (platform-provided helpers) to probe support before enabling expensive effects. Not every backend exposes every effect type; guard features behind capability checks.

Backend considerations

Composition runs on different engines per platform:

- **Windows** defaults to Direct3D via Angle; transparency and acrylic require desktop composition (check `DwmIsCompositionEnabled`).
- **macOS/iOS** lean on Metal; some blend modes fall back to software when Metal is unavailable.
- **Linux/X11** routes through OpenGL or Vulkan depending on the build; verify `TransparencyLevel` and composition availability via `X11Globals.IsCompositionEnabled`.
- **Browser** currently renders via WebGL and omits composition-only APIs. Always branch your motion layer so WebAssembly users still see essential transitions.

When features are missing, prefer classic transitions so the experience remains functional.

7. Composition animations and implicit animations

Composition animations live in `Avalonia.Rendering.Composition.Animations`:

- `ExpressionAnimation` lets you drive properties with formulas (e.g., parallax, inverse transforms).
- `KeyFrameAnimation` offers high-frequency GPU keyframes.
- `ImplicitAnimationCollection` attaches animations to property names and fires when the property changes (`CompositionObject.ImplicitAnimations`).

Example: create a parallax highlight that lags slightly behind its host.

```
var compositor = ElementComposition.GetElementVisual(header)!.Compositor;
var hostVisual = ElementComposition.GetElementVisual(header)!;

var glow = compositor.CreateSolidColorVisual();
glow.Color = Colors.Gold;
glow.Size = new Vector2((float)header.Bounds.Width, 4);
ElementComposition.SetElementChildVisual(header, glow);

var parallax = compositor.CreateExpressionAnimation("Vector3(host.Offset.X * 0.05, host.Offset.Y * 0.05
parallax.SetReferenceParameter("host", hostVisual);
parallax.Target = nameof(CompositionVisual.Offset);
glow.StartAnimation(nameof(CompositionVisual.Offset), parallax);
```

For property-driven motion, use implicit animations: create an `ImplicitAnimationCollection`, add an animation keyed by the composition property name (for example `nameof(CompositionVisual.Opacity)`), then assign the collection to `visual.ImplicitAnimations`. Each time that property changes, the compositor automatically plays the animation using `this.FinalValue` inside the expression to reference the target value (`ImplicitAnimationCollection.cs`).

`StartAnimation` pushes the animation to the render thread. Use `CompositionAnimationGroup` to start multiple animations atomically, and `Compositor.RequestCommitAsync()` to flush batched changes before measuring results.

8. Performance and diagnostics

- Prefer animating transforms (`RenderTransform`, `Opacity`) over layout-affecting properties (`Width`, `Height`). Layout invalidation happens on the UI thread and can stutter.
- Reuse animation instances; parsing keyframes or easings each time allocates. Store them as static resources.
- Disable transitions when loading data-heavy lists to avoid dozens of simultaneous animations (`Animatable.DisableTransitions`). Re-enable after the initial bind.
- For composition, batch changes and let `Compositor.RequestCommitAsync()` coalesce writes instead of spamming per-frame updates.
- Use `RendererDiagnostics` overlays (Chapter 24) to spot dropped frames and long render passes. Composition visuals show up as separate layers, so you can verify they batch correctly.
- Brush transitions fall back to discrete jumps for incompatible brush types (`BrushTransition.cs`). Verify gradients or image brushes blend the way you expect.

9. Practice lab: motion system

1. **Explicit keyframes** – Build a reusable animation resource that pulses a `NotificationBanner`, then start it from a view model with `RunAsync`. Add cancellation so repeated notifications restart smoothly.
2. **Implicit hover transitions** – Define a `Transitions` block for cards in a dashboard: fade elevation shadows, scale the card slightly, and update `TranslateTransform.Y`. Drive the transitions purely from pseudo-classes.
3. **Navigation choreography** – Wrap your page host in a `TransitioningContentControl`. Combine `PageSlide` with `CrossFade`, listen for `TransitionCompleted`, and cancel transitions when the navigation stack pops quickly.
4. **Composition parallax** – Build a composition child visual that lags behind its host using an expression animation, then snap it back with an implicit animation when pointer capture is lost.
5. **Diagnostics** – Toggle renderer diagnostics overlays, capture a short trace, and confirm that the animations remain smooth when background tasks run.

Document timing curves, easing choices, and any performance issues so the team can iterate on the experience.

10. Troubleshooting & best practices

- Animation not firing? Ensure the target property is styled (not direct) and the selector matches the control. For composition, check the animation `Target` matches the composition property name (case-sensitive).
- Looped animations via `RunAsync` throw—drive infinite loops with `Apply` or manual scheduler instead.
- Transitions chaining oddly? They trigger per property; animating both `RenderTransform` and its sub-properties simultaneously causes conflicts. Use a single `TransformOperationsTransition` to animate complex transforms.
- Composition visuals disappear after resizing? Update `Size` and `Offset` whenever the host control's bounds change, then call `Compositor.RequestCommitAsync()` to flush.
- Hot reload spawns multiple composition visuals? Remove the old child visual (`Children.Remove`) before adding a new one, or cache the sprite in the control instance.

Look under the hood (source bookmarks)

- Animation timeline & playback: `external/Avalonia/src/Avalonia.Base/Animation/Animation.cs`
- Property transitions: `external/Avalonia/src/Avalonia.Base/Animation/Transitions.cs`

- Page transitions: `external/Avalonia/src/Avalonia.Base/Animation/PageSlide.cs`, `external/Avalonia/src/Ava`
- Composition gateway: `external/Avalonia/src/Avalonia.Base/Rendering/Composition/Compositor.cs`, `external/Avalonia/src/Avalonia.Base/Rendering/Composition/CompositionTarget.cs`
- Composition effects & materials: `external/Avalonia/src/Avalonia.Base/Rendering/Composition/CompositionDr` `external/Avalonia/src/Avalonia.Base/Rendering/Composition/CompositionExperimentalAcrylicVisual.cs`, `external/Avalonia/src/Avalonia.Base/Rendering/Composition/Expressions/ExpressionAnimation.cs`
- Implicit composition animations: `external/Avalonia/src/Avalonia.Base/Rendering/Composition/CompositionOb`

Check yourself

- When would you pick a `DoubleTransition` over a keyframe animation, and why does that matter for layout cost?
- How do `IterationCount`, `FillMode`, and `PlaybackDirection` interact to determine an animation's resting value?
- What are the risks of animating direct properties, and how does Avalonia guard against them?
- How do you attach a composition child visual so it uses the same compositor as the host control?
- What steps ensure a navigation animation cancels cleanly when the route changes mid-flight?

What's next - Next: Chapter30

30. Markup, XAML compiler, and extensibility

Goal - Understand how Avalonia turns `.axaml` files into IL, resources, and runtime objects. - Choose between compiled and runtime XAML loading, and configure each for trimming, design-time, and diagnostics. - Extend the markup language with custom namespaces, markup extensions, and services without breaking tooling.

Why this matters - XAML is your declarative UI language; mastering its toolchain keeps builds fast and error messages actionable. - Compiled XAML (XamlIl) affects startup time, binary size, trimming, and hot reload behaviour. - Custom markup extensions, namespace maps, and runtime loaders enable reusable component libraries and advanced scenarios (dynamic schemas, plug-ins).

Prerequisites - Chapter 02 (project setup) for templates and build targets. - Chapter 07 (styles and selectors) and Chapter 10 (resources) for consuming XAML assets. - Chapter 08 (bindings) for compiled binding references.

1. The XAML asset pipeline

When you add `.axaml` files, the SDK-driven build uses two MSBuild tasks from `Avalonia.Build.Tasks`:

1. `GenerateAvaloniaResources` (`external/Avalonia/src/Avalonia.Build.Tasks/GenerateAvaloniaResourcesTask`
   - Runs before compilation. Packs every `AvaloniaResource` item into the `*.axaml` resource bundle (`avares://`).
   - Parses each XAML file with `XamlFileInfo.Parse`, records `x:Class` entries, and writes `/!AvaloniaResourceXamlInfo` metadata so runtime lookups can map CLR types to resource URIs.
   - Emits MSBuild diagnostics (`BuildEngine.LogError`) if it sees invalid XML or duplicate `x:Class` declarations.
2. `CompileAvaloniaXaml` (`external/Avalonia/src/Avalonia.Build.Tasks/CompileAvaloniaXamlTask.cs`)
   - Executes after C# compilation. Loads the produced assembly and references via Mono.Cecil.
   - Invokes `XamlCompilerTaskExecutor.Compile`, which runs the XamlIl compiler over each XAML resource, generates partial classes, compiled bindings, and lookup stubs under the `CompiledAvaloniaXaml` namespace, then rewrites the IL in-place.
   - Writes the updated assembly (and optional reference assembly) to `$(IntermediateOutputPath)`.

Key metadata: - `AvaloniaResource` item group entries exist by default in SDK templates; make sure custom build steps preserve the `AvaloniaCompileOutput` metadata so incremental builds work. - Set `<VerifyXamlIl>true</VerifyXamlIl>` to enable IL verification after compilation; this slows builds slightly but catches invalid IL earlier. - `<AvaloniaUseCompiledBindingsByDefault>true</AvaloniaUseCompiledBindingsByDefau` opts every binding into compiled bindings unless opted out per markup (see Chapter 08).

2. Inside the XamlIl compiler

XamlIl is Avalonia's LLVM-style pipeline built on XamlX:

1. **Parsing** (`XamlX.Parsers`) transforms XAML into an AST (`XamlDocument`).
2. **Transform passes** (`Avalonia.Markup.Xaml.XamlIl.CompilerExtensions`) rewrite the tree, resolve namespaces (`XmlnsDefinitionAttribute`), expand markup extensions, and inline templates.
3. **IL emission** (`XamlCompilerTaskExecutor`) creates classes such as `CompiledAvaloniaXaml.!XamlLoader`, `CompiledAvaloniaXaml.!AvaloniaResources`, and compiled binding factories.
4. **Runtime helpers** (`external/Avalonia/src/Markup/Avalonia.Markup.Xaml/XamlIl/Runtime/XamlIlRuntimeHelp` provide services for deferred templates, parent stacks, and resource resolution at runtime.

Every `.axaml` file with `x:Class="Namespace.View"` yields: - A partial class initializer calling `AvaloniaXamlIlRuntimeXamlLoader`. This ensures your code-behind `InitializeComponent()` wires the compiled tree. - Registration in the resource map so `AvaloniaXamlLoader.Load(new Uri("avares://..."))` can find the compiled loader.

If you set `<SkipXamlCompilation>true</SkipXamlCompilation>`, the compiler bypasses IL generation; `AvaloniaXamlLoader` then falls back to runtime parsing for each load (slower and reflection-heavy, but useful during prototyping).

3. Runtime loading and hot reload

`AvaloniaXamlLoader` (`external/Avalonia/src/Markup/Avalonia.Markup.Xaml/AvaloniaXamlLoader.cs`) chooses between: - **Compiled XAML** – looks for `CompiledAvaloniaXaml.!XamlLoader.TryLoad(string)` in the owning assembly and instantiates the pre-generated tree. - **Runtime loader** – if no compiled loader exists or when you invoke `AvaloniaLocator.CurrentMutable.Register<IRuntimeXamlLoader>(...)`. This constructs a `RuntimeXamlLoaderDocument` with your stream or string, applies `RuntimeXamlLoaderConfiguration`, and parses with PortableXaml + XamlIl runtime.

Runtime configuration knobs: - `UseCompiledBindingsByDefault` toggles compiled binding behaviour when parsing at runtime. - `DiagnosticHandler` lets you downgrade/upgrade runtime warnings or feed them into telemetry. - `DesignMode` ensures design-time services (`Design.IsDesignMode`, previews) do not execute app logic.

Use cases for runtime loading: - Live preview / hot reload (IDE hosts register their own `IRuntimeXamlLoader`). - Pluggable modules that ship XAML as data (load from database, theme packages). - Unit tests where compiling all XAML would slow loops; the headless test adapters provide a runtime loader.

4. Namespaces, schemas, and lookup

Avalonia uses `XmlnsDefinitionAttribute` (`external/Avalonia/src/Avalonia.Base/Metadata/XmlnsDefinitionAttribu` to map XML namespaces to CLR namespaces. Assemblies such as `Avalonia.Markup.Xaml` declare:

```
[assembly: XmlnsDefinition("https://github.com/avaloniaui", "Avalonia.Markup.Xaml.MarkupExtensions")]
```

Guidelines: - Add your own `[assembly: XmlnsDefinition]` for component libraries so users can `xmlns:controls="clr-namespace:MyApp.Controls"` or reuse the default Avalonia URI. - Use `[assembly: XmlnsPrefix]` (also in `Avalonia.Metadata`) to suggest a prefix for tooling. - Custom types must be public and reside in an assembly referenced by the consuming project; otherwise XamlIl will emit a type resolution error.

`IXamlTypeResolver` is available through the service provider (`Extensions.ResolveType`). When you write custom markup extensions, you can resolve types that respect `XmlnsDefinition` mappings.

5. Markup extensions and service providers

All markup extensions inherit from `Avalonia.Markup.Xaml.MarkupExtension` (`MarkupExtension.cs`) and implement `ProvideValue(IServiceProvider serviceProvider)`.

Avalonia supplies extensions such as `StaticResourceExtension`, `DynamicResourceExtension`, `CompiledBindingExtension`, and `OnPlatformExtension` (`external/Avalonia/src/Markup/Avalonia.Markup.Xaml/MarkupExtensions/*`). The service provider gives access to: - `INameScope` for named elements. - `IAvaloniaXamlIlParentStackProvider` for parent stacks (`Extensions.GetParents<T>()`). - `IRootObjectProvider`, `IUriContext`, and design-time services.

Custom markup extension example:

```
public class UppercaseExtension : MarkupExtension
{
    public string? Text { get; set; }

    public override object ProvideValue(IServiceProvider serviceProvider)
    {
        var source = Text ?? serviceProvider.GetDefaultAnchor() as TextBlock;
```

```
        return source switch
        {
            string s => s.ToUpperInvariant(),
            TextBlock block => block.Text?.ToUpperInvariant() ?? string.Empty,
            _ => string.Empty
        };
    }
}
```

Usage in XAML:

```
<TextBlock Text="{local:Uppercase Text=hello}"/>
```

Tips: - Always guard against null `Text`; the extension may be instantiated at parse time without parameters. - Use services (e.g., `serviceProvider.GetService<IServiceProvider>`) sparingly; they run on every instantiation. - For asynchronous or deferred value creation, return a delegate implementing `IProvideValueTarget` or use `XamlIlRuntimeHelpers.DeferredTransformationFactoryV2`.

6. Custom templates, resources, and compiled bindings

XamlIl optimises templates and bindings when you: - Declare controls with `x:Class` so partial classes can inject compiled fields (`InitializeComponent`). - Use `x:DataType` on `DataTemplates` to enable compiled bindings with compile-time type checking. - Add `x:CompileBindings="False"` on a scope if you need fallback to classic binding for dynamic paths.

The compiler hoists resource dictionaries and template bodies into factory methods, reducing runtime allocations. When you inspect generated IL (use `ilspy`), you'll see `new Func<IServiceProvider, object>(...)` wrappers for control templates referencing `XamlIlRuntimeHelpers.DeferredTransformationFactoryV2`.

7. Debugging and diagnostics

- Build errors referencing `AvaloniaXamlDiagnosticCodes` include the original file path; MSBuild surfaces them in IDEs with line/column.
- Runtime `XamlLoadException` (`XamlLoadException.cs`) indicates missing compiled loaders or invalid markup; the message suggests ensuring `x:Class` and `AvaloniaResource` build actions.
- Enable verbose compiler exceptions with `<AvaloniaXamlIlVerboseOutput>true</AvaloniaXamlIlVerboseOutput>` to print stack traces from the XamlIl pipeline.
- Use `avalonia-preview` (design-time host) to spot issues with namespace resolution; the previewer logs originate from the runtime loader and respect `RuntimeXamlLoaderConfiguration.DiagnosticHandler`.

8. Authoring workflow checklist

1. **Project file** – confirm `<UseCompiledBindingsByDefault>` and `<VerifyXamlIl>` match your requirements.
2. **Namespaces** – add `[assembly: XmlnsDefinition]` for every exported namespace; document the suggested prefix.
3. **Resources** – place `.axaml` under the project root or set `Link` metadata so `GenerateAvaloniaResources` records the intended resource URI.
4. **InitializeComponent** – always call it in partial classes; otherwise the compiled loader is never invoked.
5. **Testing** – run unit tests with `AvaloniaHeadless` (Chapter 21) to exercise runtime loader paths without the full compositor.

9. Practice lab: extending the markup toolchain

1. **Inspect build output** – build your project with `dotnet build /bl`. Open the MSBuild log and confirm `GenerateAvaloniaResources` and `CompileAvaloniaXaml` run with the expected inputs.

2. **Add XML namespace mappings** – create a component library, decorate it with `[assembly: XmlnsDefinition("https://schemas.myapp.com/ui", "MyApp.Controls")]`, and consume it from a separate app.
3. **Create a markup extension** – implement `{local:Uppercase}` as above, inject `IServiceProvider` utilities, and write tests that call `ProvideValue` with a fake service provider.
4. **Toggle compiled bindings** – set `<AvaloniaUseCompiledBindingsByDefault>false>`, then selectively enable compiled bindings in XAML with `{x:CompileBindings}` and observe the generated IL (via dotnet-monitor or ILSpy).
5. **Runtime loader experiment** – register a custom `IRuntimeXamlLoader` in a test harness to load XAML from strings, flip `UseCompiledBindingsByDefault`, and log diagnostics through `RuntimeXamlLoaderConfiguration.DiagnosticHandler`.

10. Troubleshooting & best practices

- Build succeeds but UI is blank? Check that your `.axaml` file still has `x:Class` and `InitializeComponent` is called. Without it, the compiled loader never runs.
- Duplicate `x:Class` errors: two XAML files declare the same CLR type; rename one or adjust the namespace. The compiler stops on duplicates to avoid ambiguous partial classes.
- `XamlTypeResolutionException`: ensure the target assembly references the library exposing the type and that you provided an `XmlnsDefinition` mapping.
- Missing resources at runtime (`avares://` fails): verify `AvaloniaResource` items exist and the resource path matches the URI (case-sensitive on Linux/macOS).
- Large diff after build: compiled XAML rewrites the primary assembly; add `obj/*.dll` to `.gitignore` and avoid checking in intermediate outputs.
- Hot reload issues: if you disable compiled XAML for faster iteration, remember to re-enable it before shipping to restore startup performance.

Look under the hood (source bookmarks)

- Resource packer: `external/Avalonia/src/Avalonia.Build.Tasks/GenerateAvaloniaResourcesTask.cs`
- XamlIl compiler driver: `external/Avalonia/src/Avalonia.Build.Tasks/CompileAvaloniaXamlTask.cs`, `external/Avalonia/src/Avalonia.Build.Tasks/XamlCompilerTaskExecutor.cs`
- Runtime loader: `external/Avalonia/src/Markup/Avalonia.Markup.Xaml/AvaloniaXamlLoader.cs`, `RuntimeXamlLoaderDocument.cs`
- Runtime helpers: `external/Avalonia/src/Markup/Avalonia.Markup.Xaml/XamlIl/Runtime/XamlIlRuntimeHelper`
- Extensions & services: `external/Avalonia/src/Markup/Avalonia.Markup.Xaml/Extensions.cs`

Check yourself

- What MSBuild tasks touch `.axaml` files, and what metadata do they emit?
- How does XamlIl decide between compiled and runtime loading for a given URI?
- Where would you place `[XmlnsDefinition]` attributes when publishing a control library?
- How do you access the root object or parent stack from inside a markup extension?
- What steps enable you to load XAML from a raw string while still using compiled bindings?

What's next - Next: Chapter31

31. Extended control modules and component gallery

Goal - Master specialized Avalonia controls that sit outside the "common controls" set: color pickers, pull-to-refresh, notifications, date/time inputs, split buttons, and more. - Understand how these modules are organized, what platform behaviours they rely on, and how to style or automate them. - Build a reusable component gallery to showcase advanced controls with theming and accessibility baked in.

Why this matters - These controls unlock polished, production-ready experiences (dashboards, media apps, mobile refresh gestures) without reinventing UI plumbing. - Many live in separate namespaces such as `Avalonia.Controls.ColorPicker` or `Avalonia.Controls.Notifications`; knowing what ships in the box saves time. - Styling, automation, and platform quirks differ from core controls—you need dedicated recipes to avoid regressions.

Prerequisites - Chapter 06 (controls tour) and Chapter 07 (styling) for basic control usage. - Chapter 09 (input) and Chapter 15 (accessibility) to reason about interactions. - Chapter 29 (animations) for transitional polish.

1. Survey of extended control namespaces

Avalonia groups advanced controls into focused namespaces:

Module	Namespace	Highlights
Color editing	`Avalonia.Controls.ColorPicker`	`ColorPicker`, `ColorView`, palette data, HSV/RGB components
Refresh gestures	`Avalonia.Controls.PullToRefresh`	`RefreshContainer`, `RefreshVisualizer`, `RefreshInfoProvider`
Notifications	`Avalonia.Controls.Notifications`	`WindowNotificationManager`, `NotificationCard`, `INotification`
Date & time	`Avalonia.Controls.DateTimePickers`	`DatePicker`, `TimePicker`, presenters, culture support
Interactive navigation	`Avalonia.Controls.SplitView`, `Avalonia.Controls.SplitButton`	Collapsible panes, hybrid buttons
Document text	`Avalonia.Controls.Documents`	Inline elements (`Run`, `Bold`, `InlineUIContainer`)
Misc UX	`Avalonia.Controls.Transitioning` `Avalonia.Controls.Notifications` `Avalonia.Controls.Primitives` helpers	`ContentControl`, `ReversibleStackPanel`,

Each module ships styles in Fluent/Simple theme dictionaries. Include the relevant `.axaml` resource dictionaries when building custom themes.

2. ColorPicker and color workflows

`ColorPicker` extends `ColorView` by providing a preview area and flyout editing UI (`ColorPicker.cs`). Key elements: - Preview content via `Content`/`ContentTemplate` (defaults to swatch + ARGB string). - Editing flyout hosts `ColorSpectrum`, sliders, and palette pickers. - Palettes live in `ColorPalettes/*`—you can supply custom palettes or localize names.

Usage snippet:

```
<ColorPicker SelectedColor="{Binding AccentColor, Mode=TwoWay}">
  <ColorPicker.ContentTemplate>
```

```
    <DataTemplate>
      <StackPanel Orientation="Horizontal" Spacing="8">
        <Border Width="24" Height="24" Background="{Binding}" CornerRadius="4"/>
        <TextBlock Text="{Binding Converter={StaticResource RgbFormatter}}"/>
      </StackPanel>
    </DataTemplate>
  </ColorPicker.ContentTemplate>
</ColorPicker>
```

Tips: - Set `ColorPicker.FlyoutPlacement` (via template) to adapt for touch vs desktop usage. - Hook `ColorView.ColorChanged` to react immediately to slider changes (e.g., update live preview alt text). - Add automation peers (`ColorPickerAutomationPeer`) if you expose color selection to screen readers.

3. Pull-to-refresh infrastructure

`RefreshContainer` wraps scrollable content and coordinates `RefreshVisualizer` animations (`RefreshContainer.cs`). Concepts: - `PullDirection` (top/bottom/left/right) chooses gesture direction. - `RefreshRequested` event fires when the user crosses the threshold. Use `RefreshCompletionDeferral` to await async work. - `RefreshInfoProviderAdapter` adapts `ScrollViewer` offsets to the visualizer; you can replace it for custom panels.

Example:

```
<ptr:RefreshContainer RefreshRequested="OnRefresh">
  <ptr:RefreshContainer.Visualizer>
    <ptr:RefreshVisualizer Orientation="TopToBottom"
                           Content="{DynamicResource PullHintTemplate}"/>
  </ptr:RefreshContainer.Visualizer>
  <ScrollViewer>
    <ItemsControl ItemsSource="{Binding FeedItems}"/>
  </ScrollViewer>
</ptr:RefreshContainer>
```

```
private async void OnRefresh(object? sender, RefreshRequestedEventArgs e)
{
    using var deferral = e.GetDeferral();
    await ViewModel.LoadLatestAsync();
}
```

Notes: - On desktop, pull gestures require touchpad/touch screen; keep a manual refresh fallback (button) for mouse-only setups. - Provide localized feedback via `RefreshVisualizer.StateChanged` (show “Release to refresh” vs “Refreshing…”). - For virtualization, ensure the underlying `ItemsControl` defers updates until after refresh completes so the visualizer can retract smoothly.

4. Notifications and toast UIs

`WindowNotificationManager` hosts toast-like notifications overlaying a `TopLevel` (`WindowNotificationManager.cs`). - Set `Position` (TopRight, BottomCenter, etc.) and `MaxItems`. - Call `Show(INotification)` or `Show(object)`; the manager wraps content in a `NotificationCard` with pseudo-classes per `NotificationType`. - Attach `WindowNotificationManager` to your main window (`new WindowNotificationManager(this)` or via XAML `NotificationLayer`).

Custom template example:

```
<Style Selector="NotificationCard">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="NotificationCard">
```

```
        <Border Classes="toast" CornerRadius="6" Background="{ThemeResource SurfaceBrush}">
          <StackPanel Orientation="Vertical" Margin="12">
            <TextBlock Text="{Binding Content.Title}" FontWeight="SemiBold"/>
            <TextBlock Text="{Binding Content.Message}" TextWrapping="Wrap"/>
            <Button Content="Dismiss" Classes="subtle"
                    notifications:NotificationCard.CloseOnClick="True"/>
          </StackPanel>
        </Border>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

Considerations: - Provide keyboard dismissal: map `Esc` to close the newest notification. - For MVVM, store `INotificationManager` in DI so view models can raise toasts without referencing the view. - On future platforms (mobile), swap to platform notification managers when available.

5. DatePicker/TimePicker for forms

`DatePicker` and `TimePicker` share presenters and respect culture-specific formats (`DatePicker.cs`, `TimePicker.cs`). - Properties: `SelectedDate`, `MinYear`, `MaxYear`, `DayVisible`, `MonthFormat`, `YearFormat`. - Template parts expose text blocks and a popup presenter; override the template to customize layout. - Two-way binding uses `DateTimeOffset?` (stay mindful of time zones).

Validation strategies: - Use `Binding` with data annotations or manual rules to block invalid ranges. - For forms, show hint text using pseudo-class `:hasnodate` when `SelectedDate` is null. - Provide automation names for the button and popup to assist screen readers.

Calendar control for planners

`Calendar` gives you a full month or decade view without the flyout wrapper. - `DisplayMode` toggles Month, Year, or Decade views—useful for date pickers embedded in dashboards. - `SelectedDates` supports multi-selection when `SelectionMode` is `MultipleRange`; bind it to a collection for booking scenarios. - Handle `DisplayDateChanged` to lazy-load data (appointments, deadlines) as the user browses months. - Customize the template to expose additional adorners (badges, tooltips). Keep `PART_DaysPanel` and related names intact so the control keeps functioning.

When you need both `Calendar` and `DatePicker`, reuse the same `CalendarDatePicker` styles so typography and spacing stay consistent.

6. SplitView and navigation panes

`SplitView` builds side drawers with flexible display modes (`SplitView.cs`). - `DisplayMode`: Overlay, Inline, CompactOverlay, CompactInline. - `IsPaneOpen` toggles state; handle `PaneOpening/PaneClosing` to intercept. - `UseLightDismissOverlayMode` enables auto-dismiss when the user clicks outside.

Usage example:

```
<SplitView IsPaneOpen="{Binding IsMenuOpen, Mode=TwoWay}"
           DisplayMode="CompactOverlay"
           PanePlacement="Left"
           CompactPaneLength="56"
           OpenPaneLength="240">
  <SplitView.Pane>
    <StackPanel>
      <Button Content="Dashboard" Command="{Binding GoHome}"/>
      <Button Content="Reports" Command="{Binding GoReports}"/>
```

```
    </StackPanel>
  </SplitView.Pane>
  <Frame Content="{Binding CurrentPage}"/>
</SplitView>
```

Tips: - On desktop, use keyboard shortcuts to toggle the pane (e.g., assign `HotKey` to `SplitButton` or global command). - Manage focus: when the pane opens via keyboard, move focus to the first focusable element; when closing, restore focus to the toggle. - Combine with `TransitioningContentControl` (Chapter 29) for smooth page transitions.

TransitioningContentControl for dynamic views

`TransitioningContentControl` wraps a content presenter with `IPageTransition` support. - Assign `PageTransition` in XAML (slide, cross-fade, custom transitions) to animate view-model swaps. - Hook `TransitionCompleted` to dispose old view models or trigger analytics when navigation ends. - Pair with `SplitView` or navigation shells to animate content panes independently of the chrome.

For component galleries, use it to showcase before/after states or responsive layouts without writing manual animation plumbing.

7. SplitButton and ToggleSplitButton

`SplitButton` provides a main action plus a secondary flyout (`SplitButton.cs`). - Primary click raises `Click`/`Command`; the secondary button shows `Flyout`. - Pseudo-classes `:flyout-open`, `:pressed`, `:checked` (for `ToggleSplitButton`). - Works nicely with `MenuFlyout` for command lists or settings.

Example:

```
<SplitButton Content="Export"
             Command="{Binding ExportAll}">
  <SplitButton.Flyout>
    <MenuFlyout>
      <MenuItem Header="Export CSV" Command="{Binding ExportCsv}"/>
      <MenuItem Header="Export JSON" Command="{Binding ExportJson}"/>
    </MenuFlyout>
  </SplitButton.Flyout>
</SplitButton>
```

Ensure `Command.CanExecute` updates by binding to view model state; `SplitButton` listens for `CanExecuteChanged` and toggles `IsEnabled` accordingly.

8. Notifications, documents, and media surfaces

- `Inline`, `Run`, `Span`, and `InlineUIContainer` in `Avalonia.Controls.Documents` let you build rich text with embedded controls (useful for notifications or chat bubbles).
- Use `InlineUIContainer` sparingly; it affects layout performance.
- Combine `NotificationCard` with document inlines to highlight formatted content (bold text, links).

`MediaPlayerElement` (available when you reference the media package) embeds audio/video playback with transport controls. - Bind `Source` to URIs or streams; the element manages decoding via platform backends (`Windows` uses Angle/DX, `Linux` goes through FFmpeg when available). - Toggle `AreTransportControlsEnabled` to show built-in play/pause UI; for custom chrome, bind to `MediaPlayer` and drive commands yourself. - Handle `MediaOpened`/`MediaEnded` to chain playlists or update state. - On platforms without native codecs, surface fallbacks (download prompts, external players) so the UI stays predictable.

9. Building a component gallery

Create a `ComponentGalleryWindow` that showcases each control with explanations and theme toggles:

```
<TabControl>
  <TabItem Header="Color">
    <StackPanel Spacing="16">
      <TextBlock Text="ColorPicker" FontWeight="SemiBold"/>
      <ColorPicker SelectedColor="{Binding ThemeColor}"/>
    </StackPanel>
  </TabItem>
  <TabItem Header="Refresh">
    <ptr:RefreshContainer RefreshRequested="OnRefreshRequested">
      <ListBox ItemsSource="{Binding Items}"/>
    </ptr:RefreshContainer>
  </TabItem>
  <TabItem Header="Notifications">
    <StackPanel>
      <Button Content="Show success" Click="OnShowSuccess"/>
      <TextBlock Text="Notifications appear top-right"/>
    </StackPanel>
  </TabItem>
</TabControl>
```

Best practices: - Offer theme toggle (Fluent light/dark) to reveal styling differences. - Surface accessibility guidance (keyboard shortcuts, screen reader notes) alongside each sample. - Provide code snippets via `TextBlock` or copy buttons so teammates can reuse patterns.

10. Practice lab: responsibility matrix

1. **Color workflows** – Customize `ColorPicker` palettes, bind to view model state, and expose automation peers for UI tests.
2. **Mobile refresh** – Implement `RefreshContainer` in a list, test on touch-enabled hardware, and add fallback commands for desktop.
3. **Toast scenarios** – Build a notification service that queues messages and exposes dismissal commands, then craft styles for different severities.
4. **Dashboard shell** – Combine `SplitView`, `SplitButton`, and `TransitioningContentControl` to create a responsive navigation shell with keyboard and pointer parity.
5. **Component gallery** – Document each control with design notes, theming tweaks, and automation IDs; integrate into project documentation.

Troubleshooting & best practices

- Many controls rely on template parts (`PART_*`). When restyling, preserve these names or update code-behind references.
- Notification overlays run on the UI thread; throttle or batch updates to avoid flooding `WindowNotificationManager` with dozens of toasts.
- `RefreshContainer` needs a `ScrollViewer` or adapter implementing `IRefreshInfoProvider`; custom panels must adapt to supply offset data.
- Date/time pickers use `DateTimeOffset`. When binding to `DateTime`, convert carefully to retain time zones.
- SplitView on compact widths: watch out for layout loops if your pane content uses `HorizontalAlignment.Stretch`; consider fixed width.

Look under the hood (source bookmarks)

- Color picker foundation: `external/Avalonia/src/Avalonia.Controls.ColorPicker/ColorPicker/ColorPicker.cs`
- Pull-to-refresh: `external/Avalonia/src/Avalonia.Controls/PullToRefresh/RefreshContainer.cs`
- Notifications: `external/Avalonia/src/Avalonia.Controls/Notifications/WindowNotificationManager.cs`, `NotificationCard.cs`
- Calendar & date/time: `external/Avalonia/src/Avalonia.Controls/Calendar/Calendar.cs`, `external/Avalonia/src/Avalonia.Controls/DateTimePickers/DatePicker.cs`, `TimePicker.cs`
- Split view/button: `external/Avalonia/src/Avalonia.Controls/SplitView/SplitView.cs`, `external/Avalonia/src/Avalonia.Controls/SplitButton/SplitButton.cs`
- Documents: `external/Avalonia/src/Avalonia.Controls/Documents/*`
- Transitions host: `external/Avalonia/src/Avalonia.Controls/TransitioningContentControl.cs`

Check yourself

- Which namespace hosts `RefreshContainer`, and why does it need a `RefreshVisualizer`?
- How does `WindowNotificationManager` limit concurrent notifications and close them programmatically?
- What steps keep `DatePicker` in sync with `DateTime` view-model properties?
- How do you style `SplitView` for light-dismiss overlay vs inline mode?
- What belongs in a component gallery to help teammates reuse advanced controls?

What's next - Next: Chapter32

32. Platform services, embedding, and native interop

Goal - Integrate Avalonia with native hosts: Windows, macOS, X11, browsers, mobile shells, and custom embedding scenarios. - Leverage `NativeControlHost`, `EmbeddableControlRoot`, and platform services (`IWindowingPlatform`, tray icons, system dialogs) to build hybrid applications. - Understand remote protocols and thin-client options to drive Avalonia content from external processes.

Why this matters - Many teams embed Avalonia inside existing apps (Win32, WPF, WinForms), or host native controls inside Avalonia shells. - Platform services expose tray icons, system navigation managers, storage providers, and more. Using them correctly keeps UX idiomatic per OS. - Remote rendering and embedding power tooling (previewers, diagnostics, multi-process architectures).

Prerequisites - Chapter 12 (windows & lifetimes) for top-level concepts. - Chapter 18–20 (platform targets) for backend overviews. - Chapter 32 builds on Chapter 29 (animations/composition) when synchronizing native surfaces.

1. Platform abstractions overview

Avalonia abstracts windowing via interfaces in `Avalonia.Controls.Platform` and `Avalonia.Platform`:

Interface	Location	Purpose
`IWindowingPlatform`	`external/Avalonia/src/Avalonia.Controls/Platform/IWindowingPlatform.cs`	Creates windows/handles, event loop levels, tray icons
`INativeControlHostImpl`	platform backends (Win32, macOS, iOS, Browser)	Hosts native HWND/NSView/UIViews inside Avalonia (`NativeControlHost`)
`ITrayIconImpl`	backend-specific	Implements tray icons (`PlatformManager.CreateTrayIcon`)
`IPlatformStorageProvider`, `ILauncher`	`Avalonia.Platform.Storage`	File pickers, launchers across platforms
`IApplicationPlatformEvents`	`Avalonia.Controls.Platform`	System-level events (activation, protocol handlers)

`PlatformManager` coordinates these services and surfaces high-level helpers (tray icons, dialogs). Check `TopLevel.PlatformImpl` to access backend-specific features.

2. Hosting native controls inside Avalonia

`NativeControlHost` (`external/Avalonia/src/Avalonia.Controls/NativeControlHost.cs`) lets you wrap native views:

- Override `CreateNativeControlCore(IPlatformHandle parent)` to instantiate native widgets (Win32 HWND, NSView, Android View).
- Avalonia attaches/detaches the native control when the host enters/leaves the visual tree, using `INativeControlHostImpl` from the current `TopLevel`.
- `TryUpdateNativeControlPosition` translates Avalonia bounds into platform coordinates and resizes the native child.

Example (Win32 HWND):

```
public class Win32WebViewHost : NativeControlHost
{
    protected override IPlatformHandle CreateNativeControlCore(IPlatformHandle parent)
    {
        var hwnd = Win32Interop.CreateWebView(parent.Handle);
```

```
        return new PlatformHandle(hwnd, "HWND");
    }

    protected override void DestroyNativeControlCore(IPlatformHandle control)
    {
        Win32Interop.DestroyWindow(control.Handle);
    }
}
```

Guidelines: - Ensure thread affinity: most native controls expect creation/destruction on the UI thread. - Handle DPI changes by listening to size changes (`BoundsProperty`) and calling the platform API to adjust scaling. - Use `NativeControlHandleChanged` for interop with additional APIs (e.g., hooking message loops). - For accessibility, expose appropriate semantics; Avalonia's `NativeControlHostAutomationPeer` helps but you may need custom peers.

3. Embedding Avalonia inside native hosts

`EmbeddableControlRoot` (`external/Avalonia/src/Avalonia.Controls/Embedding/EmbeddableControlRoot.cs`) wraps a `TopLevel` that can live in non-Avalonia environments:

- Construct with an `ITopLevelImpl` supplied by platform-specific hosts (`WinFormsAvaloniaControlHost`, `X11 XEmbed`, `Android AvaloniaView`, `iOS AvaloniaView`).
- Call `Prepare()` to initialize the logical tree and run the initial layout pass.
- Use `StartRendering`/`StopRendering` to control drawing when the host window shows/hides.
- `EnforceClientSize` ensures Avalonia matches the host surface size; disable for custom measure logic.

Examples: - **WinForms**: `WinFormsAvaloniaControlHost` hosts `EmbeddableControlRoot` inside Windows Forms. Remember to call `InitAvalonia()` before creating controls. - **X11 embedding**: `XEmbedPlug` uses `EmbeddableControlRoot` to embed into foreign X11 windows (tooling, remote previews). - **Mobile views**: `Avalonia.Android.AvaloniaView` and `Avalonia.iOS.AvaloniaView` wrap `EmbeddableControlRoot` to integrate with native UI stacks.

Interop tips: - Manage lifecycle carefully: dispose the root when the host closes to release GPU/threads. - Expose the `Content` property to your native layer for dynamic view injection. - Bridge focus and input: e.g., WinForms host sets `TabStop` and forwards focus events to the Avalonia root.

MicroCom bridges for Windows interop

Avalonia relies on MicroCom to generate COM-compatible wrappers. When embedding on Windows (drag/drop, menus, Win32 interop): - Use `Avalonia.MicroCom.CallbackBase` as the base for custom COM callbacks; it handles reference counting and error reporting. - `OleDropTarget` and native menu exporters in `Avalonia.Win32` demonstrate wrapping Win32 interfaces without hand-written COM glue. - When exposing Avalonia controls to native hosts, keep MicroCom proxies alive for the lifetime of the host window to avoid releasing underlying HWND/IDispatch too early.

You rarely need to touch MicroCom directly, but understanding it helps when diagnosing drag/drop or accessibility issues on Windows.

4. Remote rendering and previews

Avalonia's remote protocol (`external/Avalonia/src/Avalonia.Remote.Protocol`) powers the XAML previewer and custom remoting scenarios.

- `RemoteServer` (`external/Avalonia/src/Avalonia.Controls/Remote/RemoteServer.cs`) wraps an `EmbeddableControlRoot` backed by `RemoteServerTopLevelImpl`. It responds to transport messages (layout updates, pointer events) from a remote client.
- Transports: BSON over TCP (`BsonTcpTransport`), streams (`BsonStreamTransport`), or custom `IAvaloniaRemoteTransportConnection` implementations.

- Use `Avalonia.DesignerSupport` components to spin up preview hosts; they bind to `IWindowingPlatform` stubs suitable for design-time.
- On the client side, `RemoteWidget` hosts the mirrored visual tree. It pairs with `RemoteServer` to marshal input/output.
- Implement a custom `ITransport` when you need alternate channels (named pipes, WebSockets). The protocol is message-based, so you can plug in encryption or compression as needed.

Potential use cases: - Live XAML preview in IDEs (already shipped). - Remote control panels (render UI in a service, interact via TCP). - UI testing farms capturing frames via remote composition.

Security note: remote transports expose the UI tree—protect endpoints if you ship this beyond trusted tooling.

5. Tray icons, dialogs, and platform services

`IWindowingPlatform.CreateTrayIcon()` supplies backend-specific tray icon implementations. Use `PlatformManager.CreateTrayIcon()` to instantiate one:

```
var trayIcon = PlatformManager.CreateTrayIcon();
trayIcon.Icon = new WindowIcon("avares://Assets/tray.ico");
trayIcon.ToolTipText = "My App";
trayIcon.Menu = new NativeMenu
{
    Items =
    {
        new NativeMenuItem("Show", (sender, args) => mainWindow.Show()),
        new NativeMenuItem("Exit", (sender, args) => app.Shutdown())
    }
};
trayIcon.IsVisible = true;
```

Other services: - **File pickers/storage**: `StorageProvider` (Chapter 16) uses platform storage APIs; embed scenarios must supply providers in DI. - **System dialogs**: `SystemDialog` classes fall-back to managed dialogs when native APIs are unavailable. - **Application platform events**: `IApplicationPlatformEvents` handles activation (protocol URLs, file associations). Register via `AppBuilder` extensions. - **System navigation**: On mobile, `SystemNavigationManager` handles back-button events; ensure `UsePlatformDetect` registers the appropriate lifetime. - **Window chrome**: `Window` exposes `SystemDecorations`, `ExtendClientAreaToDecorationsHint`, `WindowTransparencyLevel`, and the `Chrome.WindowChrome` helpers so you can blend custom title bars with OS hit testing. Always provide resize grips and fall back to system chrome when composition is disabled.

6. Browser, Android, iOS views

- **Browser**: `Avalonia.Browser.AvaloniaView` hosts `EmbeddableControlRoot` atop WebAssembly; `NativeControlHost` implementations for the browser route to JS interop.
- **Android/iOS**: `AvaloniaView` provides native controls (Android View, iOS UIView) embedding Avalonia UI. Use `SingleViewLifetime` to tie app lifetimes to host platforms.
- Expose Avalonia content to native navigation stacks, but run Avalonia's message loop (`AppBuilder.AndroidLifecycleE` / `AppBuilder.iOS`).

7. Offscreen rendering and interoperability

`OffscreenTopLevel` (`external/Avalonia/src/Avalonia.Controls/Embedding/Offscreen/OffscreenTopLevel.cs`) allows rendering to a framebuffer without showing a window—useful for: - Server-side rendering (generate bitmaps for PDFs, emails). - Unit tests verifying layout/visual output. - Thumbnail generation for design tools.

Pair with `RenderTargetBitmap` to save results.

8. Practice lab: hybrid UI playbook

1. **Embed native control** – Host a Win32 WebView or platform-specific map view inside Avalonia using `NativeControlHost`. Ensure resize and DPI updates work.
2. **Avalonia-in-native** – Create a WinForms or WPF shell embedding `EmbeddableControlRoot`. Swap Avalonia content dynamically and synchronize focus/keyboard.
3. **Tray integration** – Add a tray icon that controls window visibility and displays context menus. Test on Windows and Linux (AppIndicator fallback).
4. **Remote preview** – Spin up `RemoteServer` with a TCP transport and connect using the Avalonia preview client to render a view remotely.
5. **Offscreen rendering** – Render a control to bitmap using `OffscreenTopLevel` + `RenderTargetBitmap` and compare results in a unit test.

Document interop boundaries (threading, disposal, event forwarding) for your team.

Troubleshooting & best practices

- Always dispose hosts (`EmbeddableControlRoot`, tray icons, remote transports) to release native resources.
- Ensure Avalonia is initialized (`BuildAvaloniaApp().SetupWithoutStarting()`) before embedding in native shells.
- Watch for DPI mismatches: use `TopLevel.PlatformImpl?.TryGetFeature<IDpiProvider>()` or subscribe to scaling changes.
- For `NativeControlHost`, guard against parent changes; detach native handles during visual tree transitions to avoid orphaned HWNDs.
- Remote transports may drop messages under heavy load—implement reconnection logic and validation.
- On macOS, tray icons require the app to stay alive (use `NSApplication.ActivateIgnoringOtherApps` when needed).

Look under the hood (source bookmarks)

- Native hosting: `external/Avalonia/src/Avalonia.Controls/NativeControlHost.cs`
- Embedding root: `external/Avalonia/src/Avalonia.Controls/Embedding/EmbeddableControlRoot.cs`
- Platform manager & services: `external/Avalonia/src/Avalonia.Controls/Platform/PlatformManager.cs`
- Remote protocol: `external/Avalonia/src/Avalonia.Controls/Remote/RemoteServer.cs`, `external/Avalonia/src/Avalonia.Controls/Remote/RemoteWidget.cs`, `external/Avalonia/src/Avalonia.Remot`
- Win32 platform: `external/Avalonia/src/Windows/Avalonia.Win32/Win32Platform.cs`
- Browser/Android/iOS hosts: `external/Avalonia/src/Browser/Avalonia.Browser/AvaloniaView.cs`, `external/Avalonia/src/Android/Avalonia.Android/AvaloniaView.cs`, `external/Avalonia/src/iOS/Avalonia.i`
- MicroCom interop: `external/Avalonia/src/Avalonia.MicroCom/CallbackBase.cs`, `external/Avalonia/src/Windo`
- Window chrome helpers: `external/Avalonia/src/Avalonia.Controls/Chrome/WindowChrome.cs`, `external/Avalonia/src/Avalonia.Controls/Window.cs`

Check yourself

- How does `NativeControlHost` coordinate `INativeControlHostImpl` and what events trigger repositioning?
- What steps are required to embed Avalonia inside an existing WinForms/WPF app?
- Which services does `IWindowingPlatform` expose, and how do you use them to create tray icons or embeddable top levels?
- How would you stream Avalonia UI to a remote client for live previews?
- When rendering offscreen, which classes help you create an isolated top level and capture the framebuffer?

What's next - Next: Chapter33

33. Code-only startup and architecture blueprint

Goal - Bootstrap Avalonia apps entirely from C# so you can skip XAML without losing features. - Structure resources, styles, and themes in code-first projects that still feel modular. - Integrate dependency injection, services, and lifetimes using the same primitives Avalonia's XAML templates rely on internally.

Why this matters - Many teams prefer a single-language stack (pure C#) for greater refactorability, dynamic UI, or source generator workflows. - Understanding the startup pipeline (`AppBuilder`, lifetimes, `Application.RegisterServices`) lets you shape architecture to match modular backends or plug-ins. - Code-first projects must explicitly wire themes, resources, and styles—knowing the underlying APIs prevents surprises when copying snippets from XAML-centric samples.

Prerequisites - Chapter 4 (startup and lifetimes) for the `AppBuilder` pipeline. - Chapter 7 (styling) to recognize how selectors, themes, and resources work. - Chapter 11 (MVVM) for structuring view-models and locator patterns that code-first projects often lean on.

1. Start from `Program.cs`: configuring the builder yourself

Avalonia templates scaffold XAML, but the real work happens in `Program.BuildAvaloniaApp()` (see `external/Avalonia/src/Avalonia.Templates/`). Code-first apps use the same `AppBuilder<TApp>` API.

```
using Avalonia;
using Avalonia.Controls.ApplicationLifetimes;
using Avalonia.ReactiveUI; // optional: add once for ReactiveUI-centric apps

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

    private static AppBuilder BuildAvaloniaApp()
        => AppBuilder.Configure<App>()
            .UsePlatformDetect()
            .LogToTrace()
            .With(new Win32PlatformOptions
            {
                CompositionMode = new[] { Win32CompositionMode.WinUIComposition } // example tweak
            })
            .With(new X11PlatformOptions { EnableIme = true })
            .With(new AvaloniaNativePlatformOptions { UseDeferredRendering = true })
            .UseSkia();
}
```

Key points from `AppBuilder.cs`: - `Configure<App>()` wires Avalonia's service locator (`AvaloniaLocator`) with the type parameter you pass. - `UsePlatformDetect()` resolves the proper backend at runtime. Replace it with `UseWin32()`, `UseAvaloniaNative()`, etc., to force a backend for tests. - `.UseReactiveUI()` (from `Avalonia.ReactiveUI/AppBuilderExtensions.cs`) registers ReactiveUI's scheduler, command binding, and view locator glue—call it in code-first projects that rely on `ReactiveCommand`. - `.With<TOptions>()` registers backend-specific option objects. Because you're not using `App.axaml`, code is the only place to set them.

Remember you can split configuration across methods for clarity:

```
private static AppBuilder ConfigurePlatforms(AppBuilder builder)
    => builder.UsePlatformDetect()
              .With(new Win32PlatformOptions { UseWgl = false })
              .With(new AvaloniaNativePlatformOptions { UseGpu = true });
```

Chaining explicit helper methods keeps `BuildAvaloniaApp` readable while preserving fluent semantics.

2. Crafting an `Application` subclass without XAML

`Application` lives in `external/Avalonia/src/Avalonia.Controls/Application.cs`. The default XAML template overrides `OnFrameworkInitializationCompleted()` after loading XAML. In code-first scenarios you:

1. Override `Initialize()` to register styles/resources explicitly.
2. (Optionally) override `RegisterServices()` to set up dependency injection.
3. Override `OnFrameworkInitializationCompleted()` to set the root visual for the selected lifetime.

```
using Avalonia;
using Avalonia.Controls.ApplicationLifetimes;
using Avalonia.Markup.Xaml.Styling;
using Avalonia.Themes.Fluent;

public sealed class App : Application
{
    public override void Initialize()
    {
        Styles.Clear();

        Styles.Add(new FluentTheme
        {
            Mode = FluentThemeMode.Dark
        });

        Styles.Add(new StyleInclude(new Uri("avares://App/Styles"))
        {
            Source = new Uri("avares://App/Styles/Controls.axaml") // optional: you can still load XAML
        });

        Styles.Add(CreateButtonStyle());

        Resources.MergedDictionaries.Add(CreateAppResources());
    }

    protected override void RegisterServices()
    {
        // called before Initialize(). Great spot for DI container wiring.
        AvaloniaLocator.CurrentMutable.Bind<IMyService>().ToSingleton<MyService>();
    }

    public override void OnFrameworkInitializationCompleted()
    {
        if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime desktop)
        {
            desktop.MainWindow = new MainWindow
            {
                DataContext = new MainWindowViewModel()
```

```
            };
        }
        else if (ApplicationLifetime is ISingleViewApplicationLifetime singleView)
        {
            singleView.MainView = new HomeView
            {
                DataContext = new HomeViewModel()
            };
        }

        base.OnFrameworkInitializationCompleted();
    }

    private static Style CreateButtonStyle()
        => new(x => x.OfType<Button>())
        {
            Setters =
            {
                new Setter(Button.CornerRadiusProperty, new CornerRadius(6)),
                new Setter(Button.PaddingProperty, new Thickness(16, 8)),
                new Setter(Button.ClassesProperty, Classes.Parse("accent"))
            }
        };

    private static ResourceDictionary CreateAppResources()
    {
        return new ResourceDictionary
        {
            ["AccentBrush"] = new SolidColorBrush(Color.Parse("#FF4F8EF7")),
            ["AccentForegroundBrush"] = Brushes.White,
            ["BorderRadiusSmall"] = new CornerRadius(4)
        };
    }
}
```

Notes from source: - `Styles` is an `IList<IStyle>` exposed by `Application`. Clearing it ensures you start from a blank slate (no default theme). Add `FluentTheme` or your own style tree. - `StyleInclude` can still ingest axaml fragments—code-first doesn't forbid XAML, it just avoids `Application.LoadComponent`. - `RegisterServices()` is invoked early in `AppBuilderBase<TApp>.Setup()` before the app is instantiated. It's designed for code-first registration patterns. - Always call `base.OnFrameworkInitializationCompleted()` to ensure any registered `OnFrameworkInitializationCompleted` handlers fire.

3. Building windows and views directly in C

When you skip XAML, every control tree is instantiated manually. You can: - Derive from `Window`, `UserControl`, or `ContentControl` and compose UI in the constructor. - Use factory methods to build complex layouts. - Compose view-model bindings using `Binding` objects or extension helpers.

```
public sealed class MainWindow : Window
{
    public MainWindow()
    {
        Title = "Code-first Avalonia";
        Width = 800;
        Height = 600;
```

```
        Content = BuildLayout();
    }

    private static Control BuildLayout()
    {
        return new DockPanel
        {
            LastChildFill = true,
            Children =
            {
                CreateHeader(),
                CreateBody()
            }
        };
    }

    private static Control CreateHeader()
        => new Border
        {
            Background = (IBrush)Application.Current!.Resources["AccentBrush"],
            Padding = new Thickness(24, 16),
            Child = new TextBlock
            {
                Text = "Dashboard",
                FontSize = 22,
                Foreground = Brushes.White,
                FontWeight = FontWeight.SemiBold
            }
        }.DockTop();

    private static Control CreateBody()
        => new StackPanel
        {
            Margin = new Thickness(24),
            Spacing = 16,
            Children =
            {
                new TextBlock { Text = "Welcome!", FontSize = 18 },
                new Button
                {
                    Content = "Refresh",
                    Command = ReactiveCommand.Create(() => Debug.WriteLine("Refresh requested"))
                }
            }
        };
}
```

Helper extension methods keep layout code tidy. You can author them in a static class:

```
public static class DockPanelExtensions
{
    public static T DockTop<T>(this T control) where T : Control
    {
        DockPanel.SetDock(control, Dock.Top);
```

```
        return control;
    }
}
```

Because you're constructing controls in code, you can register them with the `NameScope` for later lookup:

```
var scope = new NameScope();
NameScope.SetNameScope(this, scope);

var statusText = new TextBlock { Text = "Idle" };
scope.Register("StatusText", statusText);
```

This matches `NameScope` behaviour from XAML (see `external/Avalonia/src/Avalonia.Base/LogicalTree/NameScope.cs`).

4. Binding, commands, and services without markup extensions

Code-first projects rely on the same binding engine, but you create bindings manually or use compiled binding helpers.

Creating bindings programmatically

```
var textBox = new TextBox();
textBox.Bind(TextBox.TextProperty, new Binding("Query")
{
    Mode = BindingMode.TwoWay,
    UpdateSourceTrigger = UpdateSourceTrigger.PropertyChanged,
    ValidatesOnDataErrors = true
});

var searchButton = new Button
{
    Content = "Search"
};
searchButton.Bind(Button.CommandProperty, new Binding("SearchCommand"));
```

`Binding` lives in `external/Avalonia/src/Avalonia.Base/Data/Binding.cs`. Anything you can express via `{Binding}` markup is available as properties on this class. For compiled bindings, use `CompiledBindingFactory` from `Avalonia.Data.Core` directly:

```
var factory = new CompiledBindingFactory();
var compiled = factory.Create<object, string>(
    vmGetter: static vm => ((SearchViewModel)vm).Query,
    vmSetter: static (vm, value) => ((SearchViewModel)vm).Query = value,
    name: nameof(SearchViewModel.Query),
    mode: BindingMode.TwoWay);

textBox.Bind(TextBox.TextProperty, compiled);
```

Services and dependency injection

Use `AvaloniaLocator.CurrentMutable` (defined in `Application.RegisterServices`) to register services. For richer DI, integrate libraries like `Microsoft.Extensions.DependencyInjection`.

```
protected override void RegisterServices()
{
    var services = new ServiceCollection();
    services.AddSingleton<IMyService, MyService>();
    services.AddSingleton<HomeViewModel>();
```

```
    var provider = services.BuildServiceProvider();

    AvaloniaLocator.CurrentMutable.Bind<IMyService>().ToSingleton(() => provider.GetRequiredService<IMyS
    AvaloniaLocator.CurrentMutable.Bind<HomeViewModel>().ToTransient(() => provider.GetRequiredService<
}
```

Later, resolve services via `AvaloniaLocator.Current.GetService<HomeViewModel>()` or inject them into controls. Because `RegisterServices` runs before `Initialize`, you can use registered services while building resources.

5. Theming, resources, and modular structure

Code-first theming revolves around `ResourceDictionary`, `Styles`, and `StyleInclude`.

Centralize app resources

```
private static ResourceDictionary CreateAppResources()
{
    return new ResourceDictionary
    {
        MergedDictionaries =
        {
            new ResourceDictionary
            {
                ["Spacing.Small"] = 4.0,
                ["Spacing.Medium"] = 12.0,
                ["Spacing.Large"] = 24.0
            }
        },
        ["AccentBrush"] = Brushes.CornflowerBlue,
        ["AccentForegroundBrush"] = Brushes.White
    };
}
```

Use namespaced keys (`Spacing.Medium`) to avoid collisions. If you rely on resizable themes, store them in a dedicated class:

```
public static class AppTheme
{
    public static Styles Light { get; } = new Styles
    {
        new FluentTheme { Mode = FluentThemeMode.Light },
        CreateSharedStyles()
    };

    public static Styles Dark { get; } = new Styles
    {
        new FluentTheme { Mode = FluentThemeMode.Dark },
        CreateSharedStyles()
    };

    private static Styles CreateSharedStyles()
        => new Styles
        {
            new Style(x => x.OfType<Window>())
```

```
            {
                Setters =
                {
                    new Setter(Window.BackgroundProperty, Brushes.Transparent)
                }
            }
        };
}
```

Switch themes at runtime:

```
public void UseDarkTheme()
{
    Application.Current!.Styles.Clear();
    foreach (var style in AppTheme.Dark)
    {
        Application.Current.Styles.Add(style);
    }
}
```

Iterate the collection when swapping themes—`Styles` implements `IEnumerable<IStyle>` so a simple `foreach` keeps dependencies minimal. Remember to freeze brushes (`Brushes.Transparent` is already frozen) when reusing them to avoid unnecessary allocations.

Organize modules by feature

A common pattern is to place each feature in its own namespace with: - A factory method returning a `Control` (for pure code) or a partial class if you mix `.axaml` for templates. - A `ViewModel` class registered via DI. - Optional `IStyle`/`ResourceDictionary` definitions encapsulated in static classes.

Example folder layout:

```
src/
  Infrastructure/
    Services/
    Styles/
  Features/
    Dashboard/
      DashboardView.cs
      DashboardViewModel.cs
      DashboardStyles.cs
    Settings/
      SettingsView.cs
      SettingsViewModel.cs
```

`DashboardStyles` might expose a `Styles` property you merge into `Application.Styles`. Keep style/helper definitions close to the controls they customize to maintain cohesion.

6. Migrating from XAML to code-first

To convert an existing XAML-based app:

1. **Copy property settings**: For each control, move attribute values into constructors or object initializers. Attached properties map to static setters (`Grid.SetColumn(button, 1)`).
2. **Convert bindings**: Replace `{Binding}` with `control.Bind(Property, new Binding("Path"))`. For `ElementName` references, call `NameScope.Register` and `FindControl`.
3. **Transform styles**: Use `new Style(x => x.OfType<Button>().Class("accent"))` for selectors. Set `Setters` to match `<Setter>` elements.

4. **Load templates**: Where XAML used `<ControlTemplate>`, build `FuncControlTemplate`. The constructor signature matches the control type and returns the template content.
5. **Merge resources**: Replace `<ResourceDictionary.MergedDictionaries>` with `ResourceDictionary.MergedDictio`
6. **Replace markup extensions**: Many map to APIs (`DynamicResource` → `DynamicResourceBindingExtensions`, `StaticResource` → dictionary lookup). For `OnPlatform` or `OnFormFactor`, implement custom helper methods that return values based on `RuntimeInformation`.

Testing after each step keeps parity. Avalonia DevTools still works with code-first UI, so inspect logical/visual trees to confirm bindings and styles resolved correctly.

7. Practice lab

1. **From template to C#** – Scaffold a standard Avalonia MVVM template, then delete `App.axaml` and `MainWindow.axaml`. Recreate them as classes mirroring their original layout using C# object initializers. Verify styles, resources, and data bindings behave identically using DevTools.
2. **Theme switcher** – Implement light/dark `Styles` groups in code. Add a toggle button that swaps `Application.Current.Styles` and persists the choice using your service layer.
3. **DI-first startup** – Register services in `RegisterServices()` using your preferred container. Resolve view-models in `OnFrameworkInitializationCompleted` rather than `new`, ensuring the container owns lifetimes.
4. **Factory-based navigation** – Build a code-first navigation shell where pages are created via factories (`Func<Control>`). Inject factories through DI and demonstrate a plugin module adding new pages without touching XAML.
5. **Headless smoke test** – Pair with Chapter 38 by writing a headless unit test that spins up your code-first app, navigates to a view, and asserts control properties to guarantee the code-only tree is intact.

By mastering these patterns you gain confidence that Avalonia's internals don't require XAML. The framework's property system, theming engine, and lifetimes remain fully accessible from C#, letting teams tailor architecture to their tooling and review preferences.

What's next - Next: Chapter34

34. Layouts and controls authored in pure C

Goal - Compose Avalonia visual trees entirely in code using layout containers, attached properties, and fluent helpers. - Understand how `AvaloniaObject` APIs (`SetValue`, `SetCurrentValue`, observers) replace attribute syntax when you skip XAML. - Build reusable factory methods and extension helpers that keep code-generated UI readable and testable.

Why this matters - Code-first teams still need the full power of Avalonia's layout system: panels, attached properties, and templated controls all live in namespaces you can reach from C#. - Explicit property APIs make dynamic UI safer—no magic strings or runtime parsing, just compile-time members and analyzers. - Once you see how to structure factories and name scopes, you can generate UI from data, plug-ins, or source generators without sacrificing maintainability.

Prerequisites - Chapter 7 (styles) for context on how styles interact with control trees. - Chapter 9 (input) if you plan to attach event handlers in code-behind. - Chapter 33 (code-first startup) for application scaffolding and DI patterns.

1. Layout primitives in code: `StackPanel`, `Grid`, `DockPanel`

Avalonia's panels live in `external/Avalonia/src/Avalonia.Controls/`. Construct them exactly as you would in XAML, but populate `Children` and set properties directly.

```
var layout = new StackPanel
{
    Orientation = Orientation.Vertical,
    Spacing = 12,
    Margin = new Thickness(24),
    Children =
    {
        new TextBlock { Text = "Customer" },
        new TextBox { Watermark = "Name" },
        new TextBox { Watermark = "Email" }
    }
};
```

`StackPanel`'s measure logic (see `StackPanel.cs`) respects `Spacing` and `Orientation`. Because you're in code, you can wrap control creation in helper methods to keep constructors clean:

```
private static TextBox CreateLabeledInput(string label, out TextBlock caption)
{
    caption = new TextBlock { Text = label, FontWeight = FontWeight.SemiBold };
    return new TextBox { Margin = new Thickness(0, 4, 0, 16) };
}
```

Grids without XAML strings

`Grid` exposes `RowDefinitions`/`ColumnDefinitions` collections of `RowDefinition`/`ColumnDefinition`. You add definitions and set attached properties programmatically.

```
var grid = new Grid
{
    ColumnDefinitions =
    {
        new ColumnDefinition(GridLength.Auto),
        new ColumnDefinition(GridLength.Star)
    },
    RowDefinitions =
    {
```

```
        new RowDefinition(GridLength.Auto),
        new RowDefinition(GridLength.Auto),
        new RowDefinition(GridLength.Star)
    }
};

var title = new TextBlock { Text = "Orders", FontSize = 22 };
Grid.SetColumnSpan(title, 2);
grid.Children.Add(title);

var filterLabel = new TextBlock { Text = "Status" };
Grid.SetRow(filterLabel, 1);
Grid.SetColumn(filterLabel, 0);
grid.Children.Add(filterLabel);

var filterBox = new ComboBox { Items = Enum.GetValues<OrderStatus>() };
Grid.SetRow(filterBox, 1);
Grid.SetColumn(filterBox, 1);
grid.Children.Add(filterBox);
```

Attached property methods (`Grid.SetRow`, `Grid.SetColumnSpan`) are static for clarity. Because they ultimately call `AvaloniaObject.SetValue`, you can wrap them in fluent helpers if you prefer chaining (example later in section 3).

Dock layouts and last-child filling

`DockPanel` (source: `DockPanel.cs`) uses the `Dock` attached property. From code you set it with `DockPanel.SetDock(control, Dock.Left)`.

```
var dock = new DockPanel
{
    LastChildFill = true,
    Children =
    {
        CreateSidebar().DockLeft(),
        CreateFooter().DockBottom(),
        CreateMainRegion()
    }
};
```

Implement `DockLeft()` as an extension to keep code terse:

```
public static class DockExtensions
{
    public static T DockLeft<T>(this T control) where T : Control
    {
        DockPanel.SetDock(control, Dock.Left);
        return control;
    }

    public static T DockBottom<T>(this T control) where T : Control
    {
        DockPanel.SetDock(control, Dock.Bottom);
        return control;
    }
}
```

You own these helpers, so you can tailor them for your team's conventions (dock with margins, apply classes, etc.).

2. Working with the property system: `SetValue`, `SetCurrentValue`, observers

Without XAML attribute syntax you interact with `AvaloniaProperty` APIs directly. Every control inherits from `AvaloniaObject` (`AvaloniaObject.cs`), which exposes:

- `SetValue(AvaloniaProperty property, object? value)` – sets the property locally, raising change notifications and affecting bindings.
- `SetCurrentValue(AvaloniaProperty property, object? value)` – updates the effective value but preserves existing bindings/animations (great for programmatic defaults).
- `GetObservable<T>(AvaloniaProperty<T>)` – returns an `IObservable<T?>` when you need to react to changes.

Example: highlight focused text boxes by toggling a pseudo-class while keeping bindings intact.

```
var box = new TextBox();
box.GotFocus += (_, _) => box.PseudoClasses.Set(":focused", true);
box.LostFocus += (_, _) => box.PseudoClasses.Set(":focused", false);

// Provide a default width but leave bindings alone
box.SetCurrentValue(TextBox.WidthProperty, 240);
```

To wire property observers, use `GetObservable` or `GetPropertyChangedObservable` (for any property change):

```
box.GetObservable(TextBox.TextProperty)
   .Subscribe(text => _logger.Information("Text changed to {Text}", text));
```

`GetObservable` is defined in `AvaloniaObject`. Remember to dispose subscriptions when controls leave the tree—store `IDisposable` tokens and call `Dispose` in your control's `DetachedFromVisualTree` handler.

Creating reusable property helpers

When repeating property patterns, encapsulate them:

```
public static class ControlHelpers
{
    public static T WithMargin<T>(this T control, Thickness margin) where T : Control
    {
        control.Margin = margin;
        return control;
    }

    public static T Bind<T, TValue>(this T control, AvaloniaProperty<TValue> property, IBinding binding
        where T : AvaloniaObject
    {
        control.Bind(property, binding);
        return control;
    }
}
```

These mirror markup extensions in code, making complex layouts more declarative.

3. Factories, builders, and fluent composition

Large code-first views benefit from factory methods that return configured controls. Compose factories from smaller functions to keep logic readable.

```
public static class DashboardViewFactory
{
    public static Control Create(IDashboardViewModel vm)
    {
        return new Grid
        {
            ColumnDefinitions =
            {
                new ColumnDefinition(GridLength.Star),
                new ColumnDefinition(GridLength.Star)
            },
            Children =
            {
                CreateSummary(vm).WithGridPosition(0, 0),
                CreateChart(vm).WithGridPosition(0, 1)
            }
        };
    }

    private static Control CreateSummary(IDashboardViewModel vm)
        => new Border
        {
            Padding = new Thickness(24),
            Child = new TextBlock().Bind(TextBlock.TextProperty, new Binding(nameof(vm.TotalSales)))
        };
}
```

`WithGridPosition` is a fluent helper you define:

```
public static class GridExtensions
{
    public static T WithGridPosition<T>(this T element, int row, int column) where T : Control
    {
        Grid.SetRow(element, row);
        Grid.SetColumn(element, column);
        return element;
    }
}
```

This approach keeps UI declarations near data bindings, reducing mental overhead for reviewers.

Repeating structures via LINQ or loops

Because you're in C#, generate children dynamically:

```
var cards = vm.Notifications.Select((item, index) =>
    CreateNotificationCard(item).WithGridPosition(index / 3, index % 3));

var grid = new Grid
{
    ColumnDefinitions = { new ColumnDefinition(GridLength.Star), new ColumnDefinition(GridLength.Star),
};

foreach (var card in cards)
{
    grid.Children.Add(card);
```

```
}
```

`Grid` measure logic handles dynamic counts; just ensure `RowDefinitions` fits the generated children (add rows as needed or rely on `GridLength.Auto`).

Sharing styles between factories

Factories can return both controls and supporting `Styles`:

```
public static Styles DashboardStyles { get; } = new Styles
{
    new Style(x => x.OfType<TextBlock>().Class("section-title"))
    {
        Setters = { new Setter(TextBlock.FontSizeProperty, 18), new Setter(TextBlock.FontWeightProperty
    }
};
```

Merge these into `Application.Current.Styles` in `App.Initialize()` or on demand when the feature loads.

4. Managing `NameScope`, logical/visual trees, and lookup

XAML automatically registers names in a `NameScope`. In code-first views you create and assign it manually when you need element lookup or `ElementName`-like references.

```
var scope = new NameScope();
var container = new Grid();
NameScope.SetNameScope(container, scope);

var detailPanel = new StackPanel { Orientation = Orientation.Vertical };
scope.Register("DetailPanel", detailPanel);

container.Children.Add(detailPanel);
```

Later you can resolve controls with `FindControl<T>`:

```
var detail = container.FindControl<StackPanel>("DetailPanel");
```

`NameScope` implementation lives in `external/Avalonia/src/Avalonia.Base/LogicalTree/NameScope.cs`. Remember that nested scopes behave like XAML: children inherit the nearest scope unless you assign a new one.

Logical tree utilities

Avalonia's logical tree helpers (`LogicalTreeExtensions.cs`) are just as useful without XAML. Use them to inspect or traverse the tree:

```
Control? parent = myControl.GetLogicalParent();
IEnumerable<IControl> children = myControl.GetLogicalChildren().OfType<IControl>();
```

This is handy when you dynamically add/remove controls and need to ensure data contexts or resources flow correctly. To validate at runtime, enable DevTools (`Avalonia.Diagnostics`) even in code-only views—the visual tree is identical.

5. Advanced controls entirely from C

`TabControl` and dynamic pages

`TabControl` expects `TabItem` children. Compose them programmatically and bind headers/content.

```
var tabControl = new TabControl
{
    Items = new[]
    {
        new TabItem
        {
            Header = "Overview",
            Content = new OverviewView { DataContext = vm.Overview }
        },
        new TabItem
        {
            Header = "Details",
            Content = CreateDetailsGrid(vm.Details)
        }
    }
};
```

If you prefer data-driven tabs, set `Items` to a collection of view-models and provide `ItemTemplate` using `FuncDataTemplate` (see Chapter 36 for full coverage). Even then, you create the template in code:

```
tabControl.ItemTemplate = new FuncDataTemplate<IDetailViewModel>((context, _) =>
    new DetailView { DataContext = context },
    supportsRecycling: true);
```

Lists with factories

`ItemsControl` and `ListBox` take `Items` plus optional panel templates. Build the items panel in code to control layout.

```
var list = new ListBox
{
    ItemsPanel = new FuncTemplate<Panel?>(() => new WrapPanel { ItemWidth = 160, ItemHeight = 200 }),
    Items = vm.Products.Select(p => CreateProductCard(p))
};
```

Here `FuncTemplate` comes from `Avalonia.Controls.Templates` (source: `FuncTemplate.cs`). It mirrors `<ItemsPanelTemplate>`.

Popups and overlays

Controls like `FlyoutBase` or `Popup` are fully accessible in code. Example: attach a contextual menu.

```
var button = new Button { Content = "Options" };
button.Flyout = new MenuFlyout
{
    Items =
    {
        new MenuItem { Header = "Refresh", Command = vm.RefreshCommand },
        new MenuItem { Header = "Export", Command = vm.ExportCommand }
    }
};
```

The object initializer syntax keeps the code close to the equivalent XAML while exposing full IntelliSense.

6. Diagnostics and testing for code-first layouts

Because no XAML compilation step validates your layout, lean on: - **Unit tests** using `Avalonia.Headless` to instantiate controls and assert layout bounds. - **DevTools** to inspect the visual tree (launch via

`AttachDevTools()` in debug builds). - **Logging** via property observers to catch binding mistakes early.

Example headless test snippet:

```
[Fact]
public void Summary_panel_contains_totals()
{
    using var app = AvaloniaApp();

    var view = DashboardViewFactory.Create(new FakeDashboardVm());
    var panel = view.GetLogicalDescendants().OfType<TextBlock>()
        .First(t => t.Classes.Contains("total"));

    panel.Text.Should().Be("$42,000");
}
```

`GetLogicalDescendants` is defined in `LogicalTreeExtensions`. Pair this with Chapter 38 for deeper testing patterns.

7. Practice lab

1. **StackPanel to Grid refactor** – Start with a simple `StackPanel` form built in code. Refactor it to a `Grid` with columns and auto-sizing rows using only C# helpers. Confirm layout parity via DevTools.
2. **Dashboard factory** – Implement a `DashboardViewFactory` that returns a `Grid` with cards arranged dynamically based on a view-model collection. Add fluent helpers for grid position, dock, and margin management.
3. **Attached property assertions** – Write a headless unit test that constructs your view, retrieves a control by name, and asserts attached properties (`Grid.GetRow`, `DockPanel.GetDock`) to prevent regressions.
4. **Dynamic modules** – Load modules at runtime that contribute layout fragments via `Func<Control>`. Merge their `Styles`/`ResourceDictionary` contributions when modules activate and remove them when deactivated.
5. **Performance profiling** – Use `RenderTimerDiagnostics` from DevTools to monitor layout passes. Compare baseline vs. dynamic code generation to ensure your factories don't introduce unnecessary measure/arrange churn.

Mastering these patterns means you can weave Avalonia's layout system into any C#-driven architecture—no XAML required, just the underlying property system and a toolbox of fluent helpers tailored to your project.

What's next - Next: Chapter35

35. Bindings, resources, and styles with fluent APIs

Goal - Compose data bindings, resource lookups, and styles from C# using the same primitives Avalonia's XAML markup wraps. - Harness indexer paths, compiled bindings, and validation hooks when no markup extensions are available. - Build reusable style/resource factories that keep code-first projects organized and themeable.

Why this matters - Binding expressions and resource dictionaries power MVVM regardless of markup language; code-first teams need ergonomic patterns to mirror XAML equivalents. - Explicit APIs (`Binding`, `CompiledBindingFactory`, `IResourceHost`, `Style`) remove stringly-typed errors and enable richer refactoring tools. - Once bindings and resources live in code, you can conditionally compose them, share helper libraries, and unit test your infrastructure without XML parsing.

Prerequisites - Chapter 7 (styling) and Chapter 10 (resources) to understand the conceptual model. - Chapter 33 (code-only startup) for service registration and theme initialization. - Chapter 34 (layout) for structuring controls that consume bindings/styles.

1. Binding essentials without markup

Avalonia's binding engine is expressed via `Binding` (`external/Avalonia/src/Avalonia.Base/Data/Binding.cs`). Construct bindings with property paths, modes, converters, and validation:

```
var binding = new Binding("Customer.Name")
{
    Mode = BindingMode.TwoWay,
    UpdateSourceTrigger = UpdateSourceTrigger.PropertyChanged,
    ValidatesOnExceptions = true
};

nameTextBox.Bind(TextBox.TextProperty, binding);
```

`Bind` is an extension method on `AvaloniaObject` (see `BindingExtensions`). The same API supports command bindings:

```
saveButton.Bind(Button.CommandProperty, new Binding("SaveCommand"));
```

For one-time assignments, use `BindingMode.OneTime`. When you need relative bindings (`RelativeSource` in XAML), use `RelativeSource` objects:

```
var binding = new Binding
{
    RelativeSource = new RelativeSource(RelativeSourceMode.FindAncestor)
    {
        AncestorType = typeof(Window)
    },
    Path = nameof(Window.Title)
};

header.Bind(TextBlock.TextProperty, binding);
```

Indexer bindings from code

Avalonia supports indexer paths (dictionary or list access) via the same `Binding.Path` syntax used in XAML.

```
var statusText = new TextBlock();
statusText.Bind(TextBlock.TextProperty, new Binding("Statuses[SelectedStatus]"));
```

Internally the binding engine uses `IndexerNode` (see `ExpressionNodes`). You still get change notifications when the indexer raises property change events (`INotifyPropertyChanged` + `IndexerName`). For dynamic

dictionaries, call `RaisePropertyChanged("Item[]")` on changes.

Typed bindings with `CompiledBindingFactory`

Compiled bindings avoid reflection at runtime. Create a factory and supply strongly-typed accessors, mirroring `{CompiledBinding}` usage.

```
var factory = new CompiledBindingFactory();
var compiled = factory.Create<DashboardViewModel, string>(
    vmGetter: static vm => vm.Header,
    vmSetter: static (vm, value) => vm.Header = value,
    name: nameof(DashboardViewModel.Header),
    mode: BindingMode.TwoWay);

headerText.Bind(TextBlock.TextProperty, compiled);
```

`CompiledBindingFactory` resides in `Avalonia.Data.Core`. Pass `BindingPriority` if you need to align with style triggers. Because compiled bindings capture delegates, they work well with source generators or analyzers.

Binding helpers for fluent composition

Create extension methods to reduce boilerplate:

```
public static class BindingHelpers
{
    public static T BindValue<T, TValue>(this T control, AvaloniaProperty<TValue> property, string path
        BindingMode mode = BindingMode.Default) where T : AvaloniaObject
    {
        control.Bind(property, new Binding(path) { Mode = mode });
        return control;
    }
}
```

Use them when composing views:

```
var searchBox = new TextBox()
    .BindValue(TextBox.TextProperty, nameof(SearchViewModel.Query), BindingMode.TwoWay);
```

2. Validation, converters, and multi-bindings

Validation feedback

Avalonia surfaces validation errors via `BindingNotification`. In code you set validation options on binding instances:

```
var amountBinding = new Binding("Amount")
{
    Mode = BindingMode.TwoWay,
    ValidatesOnDataErrors = true,
    ValidatesOnExceptions = true
};
amountTextBox.Bind(TextBox.TextProperty, amountBinding);
```

Listen for errors using `BindingObserver` or property change notifications on `DataValidationErrors` (see `external/Avalonia/src/Avalonia.Controls/DataValidationErrors.cs`). Example hooking into the attached property:

```
amountTextBox.GetObservable(DataValidationErrors.HasErrorsProperty)
    .Subscribe(hasErrors => amountTextBox.Classes.Set(":invalid", hasErrors));
```

Converters and converter parameters

Instantiate converters directly and assign them to `Binding.Converter`:

```
var converter = new BooleanToVisibilityConverter();
var binding = new Binding("IsBusy")
{
    Converter = converter
};

spinner.Bind(IsVisibleProperty, binding);
```

For inline converters, create lambda-based converter classes implementing `IValueConverter`. In code-first setups you can keep converter definitions close to usage.

Multi-binding composition

`MultiBinding` lives in `Avalonia.Base/Data/MultiBinding.cs`. Configure binding collection and converters directly.

```
var multi = new MultiBinding
{
    Bindings =
    {
        new Binding("FirstName"),
        new Binding("LastName")
    },
    Converter = FullNameConverter.Instance
};

fullNameText.Bind(TextBlock.TextProperty, multi);
```

`FullNameConverter` implements `IMultiValueConverter`. When multi-binding in code, consider static singletons to avoid allocations.

3. Commands and observables from code

Avalonia command support is just binding to `ICommand`. With code-first patterns, leverage `ReactiveCommand` or custom commands while still using `Bind`:

```
refreshButton.Bind(Button.CommandProperty, new Binding("RefreshCommand"));
```

To observe property changes for reactive flows, use `GetObservable` or `PropertyChanged` events. Combine with `ReactiveUI` by using `WhenAnyValue` inside view models—code-first views don't change this interop.

4. Resource dictionaries and lookup patterns

`ResourceDictionary` is just a C# collection (see `external/Avalonia/src/Avalonia.Base/Controls/ResourceDictionary` Create dictionaries and merge them programmatically.

```
var typographyResources = new ResourceDictionary
{
    ["Heading.FontSize"] = 24.0,
    ["Body.FontSize"] = 14.0
};
```

```
Application.Current!.Resources.MergedDictionaries.Add(typographyResources);
```

For per-control resources:

```
var card = new Border
{
    Resources =
    {
        ["CardBackground"] = Brushes.White,
        ["CardShadow"] = new BoxShadow { Color = Colors.Black, Opacity = 0.1, Blur = 8 }
    }
};
```

`Resources` property is itself a `ResourceDictionary`. Use strongly-typed wrapper classes to centralize resource keys:

```
public static class ResourceKeys
{
    public const string AccentBrush = nameof(AccentBrush);
    public const string AccentForeground = nameof(AccentForeground);
}

var accent = (IBrush)Application.Current!.Resources[ResourceKeys.AccentBrush];
```

Wrap lookups with helper methods to provide fallbacks:

```
public static TResource GetResource<TResource>(this IResourceHost host, string key, TResource fallback)
{
    return host.TryFindResource(key, out var value) && value is TResource typed
        ? typed
        : fallback;
}
```

`IResourceHost`/`IResourceProvider` interfaces are defined in `Avalonia.Styling`. Controls implement them, so you can call `control.TryFindResource` directly.

5. Building styles fluently

`Style` objects can be constructed with selectors and setters. The selector API mirrors XAML but uses lambda syntax.

```
var buttonStyle = new Style(x => x.OfType<Button>().Class("primary"))
{
    Setters =
    {
        new Setter(Button.BackgroundProperty, Brushes.MediumPurple),
        new Setter(Button.ForegroundProperty, Brushes.White),
        new Setter(Button.PaddingProperty, new Thickness(20, 10))
    },
    Triggers =
    {
        new Trigger
        {
            Property = Button.IsPointerOverProperty,
            Value = true,
            Setters = { new Setter(Button.BackgroundProperty, Brushes.DarkMagenta) }
        }
```

```
    }
};
```

Add styles to `Application.Current.Styles` or to a specific control's `Styles` collection. Remember to freeze brushes (call `ToImmutable()` or use static brushes) when reusing them widely.

Style includes and theme variants

You can still load existing `.axaml` resources via `StyleInclude`, or create purely code-based ones:

```
var theme = new Styles
{
    new StyleInclude(new Uri("avares://App/Styles"))
    {
        Source = new Uri("avares://App/Styles/Buttons.axaml")
    },
    buttonStyle
};

Application.Current!.Styles.AddRange(theme);
```

In pure C#, `Styles` is just a list. If you don't have `AddRange`, iterate:

```
foreach (var style in theme)
{
    Application.Current!.Styles.Add(style);
}
```

Theme variants (`ThemeVariant`) can be set directly on styles:

```
buttonStyle.Resources[ThemeVariant.Light] = Brushes.Black;
buttonStyle.Resources[ThemeVariant.Dark] = Brushes.White;
```

6. Code-first binding infrastructure patterns

Binding factories per view-model

Encapsulate binding creation in dedicated classes to avoid scattering strings:

```
public static class DashboardBindings
{
    public static Binding TotalSales => new(nameof(DashboardViewModel.TotalSales)) { Mode = BindingMode
    public static Binding RefreshCommand => new(nameof(DashboardViewModel.RefreshCommand));
}

salesText.Bind(TextBlock.TextProperty, DashboardBindings.TotalSales);
refreshButton.Bind(Button.CommandProperty, DashboardBindings.RefreshCommand);
```

Expression-based helpers

Use expression trees to produce path strings while maintaining compile-time checks:

```
public static class BindingFactory
{
    public static Binding Create<TViewModel, TValue>(Expression<Func<TViewModel, TValue>> expression,
        BindingMode mode = BindingMode.Default)
    {
        var path = ExpressionHelper.GetMemberPath(expression); // custom helper
        return new Binding(path) { Mode = mode };
```

```
    }
}
```

`ExpressionHelper` can walk the expression tree to build `Customer.Addresses[0].City` style paths, ensuring refactors update bindings.

Declarative resource builders

Provide factories for resource dictionaries similar to style factories:

```
public static class ResourceFactory
{
    public static ResourceDictionary CreateColors() => new()
    {
        [ResourceKeys.AccentBrush] = new SolidColorBrush(Color.Parse("#4F8EF7")),
        [ResourceKeys.AccentForeground] = Brushes.White
    };
}
```

Merge them in `App.Initialize()` or feature modules when needed.

7. Practice lab

1. **Binding library** – Implement a helper class that exposes strongly-typed bindings for a view-model using expression trees. Replace string-based paths in an existing code-first view.
2. **Indexer dashboards** – Build a dashboard card that binds to `Metrics["TotalRevenue"]` from a dictionary-backed view-model. Raise change notifications on dictionary updates and verify the UI refreshes.
3. **Validation styling** – Create a reusable style that applies an `:invalid` pseudo-class template to controls with validation errors. Trigger validation via a headless test.
4. **Resource fallback provider** – Write an extension method that locates a resource by key and throws a descriptive exception if missing, including current logical tree path. Use it in a headless test to catch missing theme registrations.
5. **Theme toggler** – Compose two `Styles` collections (light/dark) in code, swap them at runtime, and ensure all bindings to theme resources update automatically. Validate behaviour with a headless pixel test (Chapter 40).

With bindings, resources, and styles expressed in code, your Avalonia app gains powerful refactorability and testability. Embrace the fluent APIs and helper patterns to keep code-first UI as expressive as any XAML counterpart.

What's next - Next: Chapter36

36. Templates, indexers, and dynamic component factories

Goal - Compose control, data, and tree templates in pure C# using Avalonia's functional template APIs. - Harness indexer-driven bindings and template bindings to build dynamic, data-driven components. - Construct factories and selectors that swap templates at runtime without touching XAML.

Why this matters - Templates define how controls render. In code-first projects you still need `FuncControlTemplate`, `FuncDataTemplate`, and selectors to mirror the flexibility of XAML. - Indexer bindings and instanced bindings power advanced scenarios such as virtualization, item reuse, and hierarchical data. - Dynamic factories unlock plugin architectures, runtime theme changes, and feature toggles—all while keeping strong typing and testability.

Prerequisites - Chapter 34 (layouts) to place templated content within layouts. - Chapter 35 (bindings/resources) for binding syntax and helper patterns. - Chapter 23 (custom controls) if you plan to author templated controls that consume templates from code.

1. Control templates in code with `FuncControlTemplate`

`FuncControlTemplate<T>` (source: `external/Avalonia/src/Avalonia.Controls/Templates/FuncControlTemplate.cs`) produces a `ControlTemplate` that builds visuals from code. It takes a lambda that receives the templated parent and returns a `Control`/`IControl` tree.

```
public static ControlTemplate CreateCardTemplate()
{
    return new FuncControlTemplate<ContentControl>((parent, scope) =>
    {
        var border = new Border
        {
            Background = Brushes.White,
            CornerRadius = new CornerRadius(12),
            Padding = new Thickness(16),
            Child = new ContentPresenter
            {
                Name = "PART_ContentPresenter"
            }
        };

        scope?.RegisterNamed("PART_ContentPresenter", border.Child);
        return border;
    });
}
```

Attach the template to a control:

```
var card = new ContentControl
{
    Template = CreateCardTemplate(),
    Content = new TextBlock { Text = "Dashboard" }
};
```

Notes from the source implementation: - The second parameter (`INameScope scope`) lets you register named parts exactly like `<ControlTemplate>` does in XAML. Use it to satisfy template part lookups in your control's code-behind. - The lambda executes each time the control template is applied, so create new control instances inside the lambda—avoid caching across calls.

Template bindings and TemplatedParent

Use TemplateBinding helpers (TemplateBindingExtensions) to bind template visual properties to the templated control.

```
return new Border
{
    Background = Brushes.White,
    [!Border.BackgroundProperty] = parent.GetTemplateBinding(ContentControl.BackgroundProperty),
    Child = new ContentPresenter()
};
```

The [!Property] indexer syntax is shorthand for creating a template binding (enabled by the Avalonia.Markup.Declarative helpers). If you prefer explicit code, use TemplateBindingExtensions.Bind:

```
var presenter = new ContentPresenter();
presenter.Bind(ContentPresenter.ContentProperty, parent.GetTemplateBinding(ContentControl.ContentProper
```

TemplateBindingExtensions.cs shows this helper returns a lightweight binding linked to the templated parent's property value.

2. Data templates with FuncDataTemplate

FuncDataTemplate<T> (source: FuncDataTemplate.cs) creates visuals for data items. Often you assign it to ContentControl.ContentTemplate or ItemsControl.ItemTemplate.

```
var itemTemplate = new FuncDataTemplate<OrderItem>((item, _) =>
    new Border
    {
        Margin = new Thickness(0, 0, 0, 12),
        Child = new StackPanel
        {
            Orientation = Orientation.Horizontal,
            Spacing = 12,
            Children =
            {
                new TextBlock { Text = item.ProductName, FontWeight = FontWeight.SemiBold },
                new TextBlock { Text = item.Quantity.ToString() }
            }
        }
    }, recycle: true);
```

Pass recycle: true to participate in virtualization (controls are reused). Attach to an ItemsControl:

```
itemsControl.ItemTemplate = itemTemplate;
```

Binding inside data templates

Because the template receives the data item, you can access its properties directly or create bindings relative to the template context.

```
var template = new FuncDataTemplate<Customer>((item, scope) =>
{
    var balance = new TextBlock();
    balance.Bind(TextBlock.TextProperty, new Binding("Balance")
    {
        StringFormat = "{0:C}"
    });
```

```
    return new StackPanel
    {
        Children =
        {
            new TextBlock { Text = item.Name },
            balance
        }
    };
});
```

`FuncDataTemplate` sets the `DataContext` to the item automatically, so bindings with explicit paths work without additional setup.

Template selectors

`FuncDataTemplate` supports predicates for conditional templates. Use the overload that accepts a `Func<object?, bool>` predicate.

```
var positiveTemplate = new FuncDataTemplate<Transaction>((item, _) => CreateTransactionRow(item));
var negativeTemplate = new FuncDataTemplate<Transaction>((item, _) => CreateTransactionRow(item, isDebi

var selector = new FuncDataTemplate<Transaction>((item, _) =>
    (item.Amount >= 0 ? positiveTemplate.Build(item) : negativeTemplate.Build(item))!,
    supportsRecycling: true);
```

For more complex selection logic, implement `IDataTemplate` manually or use `DataTemplateSelector` base classes from community packages.

3. Hierarchical templates with `FuncTreeDataTemplate`

`FuncTreeDataTemplate<T>` builds item templates for hierarchical data such as tree views. It receives the item and a recursion function.

```
var treeTemplate = new FuncTreeDataTemplate<DirectoryNode>((item, _) =>
    new StackPanel
    {
        Orientation = Orientation.Horizontal,
        Children =
        {
            new TextBlock { Text = item.Name }
        }
    },
    x => x.Children,
    true);

var treeView = new TreeView
{
    Items = fileSystem.RootNodes,
    ItemTemplate = treeTemplate
};
```

The third argument is `supportsRecycling`. The second argument is the accessor returning child items. This mirrors XAML's `<TreeDataTemplate ItemsSource="{Binding Children}">`.

`FuncTreeDataTemplate` internally wires `TreeDataTemplate` with lambda-based factories, so you get the same virtualization behaviour as XAML templates.

4. Instanced bindings and indexer tricks

`InstancedBinding` (source: `external/Avalonia/src/Avalonia.Data/Core/InstancedBinding.cs`) lets you precompute a binding for a known source. It's powerful when a template needs to bind to an item-specific property or when you assemble UI from graphs.

```
var binding = new Binding("Metrics[\"Total\"]") { Mode = BindingMode.OneWay };
var instanced = InstancedBinding.OneWay(binding, metricsDictionary);

var text = new TextBlock();
text.Bind(text.TextProperty, instanced);
```

Because you supply the source (`metricsDictionary`), the binding bypasses `DataContext`. This is useful in templates where you juggle multiple sources (e.g., templated parent + external service).

Binding to template parts via indexers

Within templates you can reference named parts registered through `scope.RegisterNamed`. After applying the template, resolve them via `TemplateAppliedEventArgs`.

```
protected override void OnApplyTemplate(TemplateAppliedEventArgs e)
{
    base.OnApplyTemplate(e);
    _presenter = e.NameScope.Find<ContentPresenter>("PART_ContentPresenter");
}
```

From code-first templates, ensure the name scope registration occurs inside the template lambda as shown earlier.

5. Swapping templates at runtime

Because templates are just CLR objects, you can replace them dynamically to support different visual representations.

```
public void UseCompactTemplates(Window window)
{
    window.Resources["CardTemplate"] = Templates.CompactCard;
    window.Resources["ListItemTemplate"] = Templates.CompactListItem;

    foreach (var presenter in window.GetVisualDescendants().OfType<ContentPresenter>())
    {
        presenter.UpdateChild(); // apply new template
    }
}
```

`ContentPresenter.UpdateChild()` forces the presenter to re-evaluate its template. `GetVisualDescendants` comes from `VisualTreeExtensions`. Consider performance: only call on affected presenters.

Use `IStyle` triggers or the view-model to change templates automatically. Example using a binding:

```
contentControl.Bind(ContentControl.ContentTemplateProperty, new Binding("SelectedTemplate")
{
    Mode = BindingMode.OneWay
});
```

The view-model exposes `IDataTemplate SelectedTemplate`, and your code-first view updates this property to switch visuals.

6. Component factories and virtualization

Control factories

Wrap template logic in factories that accept data and return controls, useful for plugin systems.

```
public interface IWidgetFactory
{
    bool CanHandle(string widgetType);
    Control Create(IWidgetContext context);
}

public sealed class ChartWidgetFactory : IWidgetFactory
{
    public bool CanHandle(string widgetType) => widgetType == "chart";

    public Control Create(IWidgetContext context)
    {
        return new Border
        {
            Child = new ChartControl { DataContext = context.Data }
        };
    }
}
```

Register factories and pick one at runtime:

```
var widget = factories.First(f => f.CanHandle(config.Type)).Create(context);
panel.Children.Add(widget);
```

Factories can also emit data templates instead of controls. For virtualization, return a `FuncDataTemplate` that participates in recycling.

Items panel factories

`ItemsControl` allows specifying the `ItemsPanel` with `FuncTemplate<Panel?>`. Build them from code to align virtualization mode with runtime options.

```
itemsControl.ItemsPanel = new FuncTemplate<Panel?>(() =>
    new VirtualizingStackPanel
    {
        Orientation = Orientation.Vertical,
        VirtualizationMode = ItemVirtualizationMode.Simple
    });
```

`FuncTemplate<T>` lives in `external/Avalonia/src/Avalonia.Controls/Templates/FuncTemplate.cs` and returns a new panel per items presenter.

Recycling with `RecyclingElementFactory`

Avalonia's element factories provide direct control over virtualization (see `external/Avalonia/src/Avalonia.Controls/Gene`
You can use `RecyclingElementFactory` and supply templates via `IDataTemplate` implementations defined in code.

```
var factory = new RecyclingElementFactory
{
    RecycleKey = "Widget",
    Template = new FuncDataTemplate<IWidgetViewModel>((item, _) => WidgetFactory.CreateControl(item))
};
```

```
var items = new ItemsRepeater { ItemTemplate = factory };
```

`ItemsRepeater` (in `Avalonia.Controls`) mirrors WinUI's control. Providing a factory integrates with virtualization surfaces better than raw `ItemsControl` in performance-sensitive scenarios.

7. Testing templates and factories

- **Unit tests**: Use `FuncDataTemplate.Build(item)` to materialize the control tree in memory and assert shape/values.

```
[Fact]
public void Order_item_template_renders_quantity()
{
    var template = Templates.OrderItem;
    var control = (Control)template.Build(new OrderItem { Quantity = 5 }, null)!;

    control.GetVisualDescendants().OfType<TextBlock>().Should().Contain(t => t.Text == "5");
}
```

- **Headless rendering**: Combine with Chapter 40 to capture template output bitmaps.
- **Name scope checks**: After applying control templates, call `TemplateAppliedEventArgs.NameScope.Find` in tests to guarantee required parts exist.

8. Practice lab

1. **Card control template** – Build a `FuncControlTemplate` for a `CardControl` that registers named parts, uses template bindings for background/content, and applies to multiple instances with different content.
2. **Conditional data templates** – Create templates for `IssueViewModel` that render differently based on `IsClosed`. Swap templates dynamically by changing a property on the view-model.
3. **Hierarchical explorer** – Compose a `TreeView` for file system data using `FuncTreeDataTemplate`, including icons and lazy loading. Ensure child collections load on demand.
4. **Template factory registry** – Implement a registry of `IDataTemplate` factories keyed by type names. Resolve templates at runtime and verify virtualization with an `ItemsRepeater` in a headless test.
5. **Template swap diagnostics** – Write a helper that re-applies templates when theme changes occur, logging how many presenters were updated. Ensure the log stays small by limiting scope to affected regions.

By mastering code-based templates, indexers, and factories, you gain full control over Avalonia's presentation layer without depending on XAML. Combine these techniques with the binding and layout patterns from earlier chapters to build highly dynamic, testable UI modules in pure C#.

What's next - Next: Chapter37

37. Reactive patterns, helpers, and tooling for code-first teams

Goal - Combine Avalonia's property system with reactive libraries (ReactiveUI, DynamicData) entirely from C#. - Build helper extensions for behaviours, pseudo-classes, transitions, and animation triggers without XAML. - Integrate diagnostics and hot-reload-style tooling that keeps developer loops tight in code-first workflows.

Why this matters - Code-first projects often favour reactive patterns to keep UI logic composable and testable. - Avalonia exposes rich helper APIs (`Classes`, `PseudoClasses`, `Transitions`, `Interaction`) that work perfectly in C# once you know where to look. - Tooling such as DevTools, live reload, and logging remain essential even without XAML; wiring them programmatically ensures parity with markup-heavy projects.

Prerequisites - Chapter 33–36 for code-first startup, layouts, bindings, and templates. - Chapter 29 (animations) and Chapter 24 (DevTools) for background on transitions and diagnostics. - Working familiarity with ReactiveUI/DynamicData if you plan to reuse those patterns.

1. Reactive building blocks in Avalonia

Avalonia's property system already supports observables. `AvaloniaObject` exposes `GetObservable` and `GetPropertyChangedObservable` so you can build reactive pipelines without XAML triggers.

```
var textBox = new TextBox();
textBox.GetObservable(TextBox.TextProperty)
    .Throttle(TimeSpan.FromMilliseconds(250), RxApp.MainThreadScheduler)
    .DistinctUntilChanged()
    .Subscribe(text => _search.Execute(text));
```

Use `ObserveOn(RxApp.MainThreadScheduler)` to marshal onto the UI thread when subscribing. For non-ReactiveUI projects, use `DispatcherScheduler.Current` (from `Avalonia.Reactive`) or `Dispatcher.UIThread.InvokeAsync` inside the observer.

Connecting to ReactiveUI view-models

ReactiveUI view-models usually expose `ReactiveCommand` and `ObservableAsPropertyHelper`. Bind them as usual, but you can also subscribe directly:

```
var vm = new DashboardViewModel();
vm.WhenAnyValue(x => x.IsLoading)
  .ObserveOn(RxApp.MainThreadScheduler)
  .Subscribe(isLoading => spinner.IsVisible = isLoading);
```

`WhenAnyValue` is extension from `ReactiveUI`. For code-first views, you may bridge them via constructor injection, ensuring the view wires observable pipelines in its constructor or `OnAttachedToVisualTree` lifecycle methods.

DynamicData for collections

DynamicData shines when projecting observable collections into UI-friendly lists.

```
var source = new SourceList<ItemViewModel>();
var bindingList = source.Connect()
    .Filter(item => item.IsEnabled)
    .Sort(SortExpressionComparer<ItemViewModel>.Descending(x => x.CreatedAt))
    .ObserveOn(RxApp.MainThreadScheduler)
    .Bind(out var items)
    .Subscribe();
```

```
listBox.Items = items;
```

Dispose the subscription when the control unloads to prevent leaks (e.g., store `IDisposable` and dispose in `DetachedFromVisualTree`).

2. Working with `Classes` and `PseudoClasses`

`Classes` and `PseudoClasses` collections (defined in `Avalonia.Styling`) let you toggle CSS-like states entirely from C#.

```
var panel = new Border();
panel.Classes.Add("card"); // corresponds to :class selectors in styles

panel.PseudoClasses.Set(":active", true);
```

Use helpers to line up state changes with view-model events:

```
vm.WhenAnyValue(x => x.IsSelected)
  .Subscribe(selected => panel.Classes.Toggle("selected", selected));
```

`Toggle` is an extension you can write:

```
public static class ClassExtensions
{
    public static void Toggle(this Classes classes, string name, bool add)
    {
        if (add)
            classes.Add(name);
        else
            classes.Remove(name);
    }
}
```

Behaviours from `Avalonia.Interactivity`

`Interaction` (in `external/Avalonia/src/Avalonia.Interactivity/Interaction.cs`) provides behaviour collections similar to WPF. You can attach behaviours programmatically via `Interaction.SetBehaviors`.

```
Interaction.SetBehaviors(listBox, new BehaviorCollection
{
    new SelectOnPointerOverBehavior()
});
```

Behaviours are regular classes implementing `IBehavior`. Author your own to encapsulate complex logic like drag-to-reorder.

3. Transitions, animations, and reactive triggers

`Transitions` collection (from `Avalonia.Animation`) lives on `Control`. Build transitions and hook them dynamically.

```
panel.Transitions = new Transitions
{
    new DoubleTransition
    {
        Property = Border.OpacityProperty,
        Duration = TimeSpan.FromMilliseconds(200),
        Easing = new CubicEaseOut()
```

```
    }
};
```

Activate transitions via property setters:

```
vm.WhenAnyValue(x => x.ShowDetails)
  .Subscribe(show => panel.Opacity = show ? 1 : 0);
```

The change triggers the transition. Because transitions live on the control, you can swap them per theme or feature by replacing the `Transitions` collection at runtime.

Animation helpers

`Animatable.BeginAnimation` (from `AnimationExtensions`) lets you trigger storyboards without styles:

```
panel.BeginAnimation(Border.OpacityProperty, new Animation
{
    Duration = TimeSpan.FromMilliseconds(400),
    Easing = new SineEaseInOut(),
    Children =
    {
        new KeyFrames
        {
            new KeyFrame { Cue = new Cue(0d), Setters = { new Setter(Border.OpacityProperty, 0d) } },
            new KeyFrame { Cue = new Cue(1d), Setters = { new Setter(Border.OpacityProperty, 1d) } }
        }
    }
});
```

Encapsulate animations into factory methods for reuse across views.

4. Hot reload and state persistence helpers

While Avalonia's XAML Previewer focuses on markup, code-first workflows can approximate hot reload using: - **`DevTools`**: `AttachDevTools()` on the main window or `AppBuilder` (see `ApplicationLifetimes`). - **`Avalonia.ReactiveUI` HotReload** packages or community tooling for reloading compiled assemblies. - **State persistence**: store view-model state in services to rehydrate UI after code changes.

Enable DevTools programmatically in debug builds:

```
if (Debugger.IsAttached)
{
    this.AttachDevTools();
}
```

For headless tests, log control trees after creation to confirm state without UI.

5. Diagnostics pipelines

Integrate logging by observing key properties and commands.

```
var subscription = panel.GetPropertyChangedObservable(Border.OpacityProperty)
    .Subscribe(args => _logger.Debug("Opacity changed from {Old} to {New}", args.OldValue, args.NewValu
```

Tie into Avalonia's diagnostics overlays (Chapter 24) by enabling them in code-first startup:

```
if (Debugger.IsAttached)
{
    RenderOptions.ProcessRenderOperations = true;
```

```
    RendererDiagnostics.DebugOverlays = RendererDebugOverlays.Fps | RendererDebugOverlays.Layout;
}
```

6. Putting it together: Building reusable helper libraries

Create a shared library of helpers tailored to your code-first patterns:

```
public static class ReactiveControlHelpers
{
    public static IDisposable BindState<TViewModel>(this TViewModel vm, Control control,
        Expression<Func<TViewModel, bool>> property, string pseudoClass)
    {
        return vm.WhenAnyValue(property)
            .ObserveOn(RxApp.MainThreadScheduler)
            .Subscribe(value => control.PseudoClasses.Set(pseudoClass, value));
    }
}
```

Use it in views:

```
_disposables.Add(vm.BindState(this, x => x.IsActive, ":active"));
```

Maintain a `CompositeDisposable` on the view to dispose subscriptions when the view unloads. Override `OnAttachedToVisualTree`/`OnDetachedFromVisualTree` to manage lifetime.

7. Practice lab

1. **Reactive state toggles** – Implement a helper that watches `WhenAnyValue` on a view-model and toggles `Classes` on a panel. Verify with headless tests that pseudo-class changes propagate to styles.
2. **Transition kit** – Build a factory returning `Transitions` configured per theme (e.g., fast vs. slow). Swap collections at runtime and instrument the effect with property observers.
3. **Behavior registry** – Create a behaviour that wires `PointerMoved` events into an observable stream. Use it to implement drag selection without code-behind duplication.
4. **Diagnostic dashboard** – Add DevTools and renderer overlays programmatically. Expose a keyboard shortcut (ReactiveCommand) that toggles them during development.
5. **Hot reload simulation** – Persist view-model state to a service, tear down the view, rebuild it from code, and reapply state to mimic live-edit workflows. Assert via unit test that state survives the rebuild.

Reactive helper patterns ensure code-first Avalonia apps stay expressive, maintainable, and observable. By leveraging observables, behaviours, transitions, and tooling APIs directly from C#, your team keeps the productivity of markup-driven workflows while embracing the flexibility of a single-language stack.

What's next - Next: Chapter38

38. Headless platform fundamentals and lifetimes

Goal - Run Avalonia apps without a windowing system so tests, previews, and automation can execute in CI. - Configure headless lifetimes, services, and render loops to mimic production behaviour while remaining deterministic. - Understand the knobs provided by `Avalonia.Headless` so you can toggle Skia rendering, timers, and focus/input handling on demand.

Why this matters - Headless execution unlocks fast feedback loops: BDD/UI unit tests, snapshot rendering, and tooling all rely on it. - CI agents rarely expose desktops or GPUs; the headless backend gives you a predictable environment across Windows, macOS, and Linux. - Knowing the lifetimes and options ensures app startup mirrors real targets—preventing bugs that only appear when the full desktop lifetime runs.

Prerequisites - Chapter 4 (startup and lifetimes) for the `AppBuilder` pipeline. - Chapter 33 (code-first startup) for wiring services/resources without XAML. - Chapter 21 (Headless and testing overview) for the bigger picture of test tooling.

1. Meet the headless platform

The headless backend lives in `external/Avalonia/src/Headless/Avalonia.Headless`. You enable it by calling `UseHeadless()` on `AppBuilder`.

```
using Avalonia;
using Avalonia.Headless;
using Avalonia.Themes.Fluent;

public static class Program
{
    public static AppBuilder BuildAvaloniaApp(bool enableSkia = false)
        => AppBuilder.Configure<App>()
            .UseHeadless(new AvaloniaHeadlessPlatformOptions
            {
                UseHeadlessDrawing = !enableSkia,
                UseSkia = enableSkia,
                AllowEglInitialization = false,
                PreferDispatcherScheduling = true
            })
            .LogToTrace();
}
```

Key extension: `AvaloniaHeadlessAppBuilderExtensions.UseHeadless` registers platform services, render loop, and input plumbing. Options: - `UseHeadlessDrawing`: if `true`, renders to an in-memory framebuffer without Skia. - `UseSkia`: when `true`, create a Skia GPU context (requires `UseHeadlessDrawing = false`). - `AllowEglInitialization`: opt-in to EGL for hardware acceleration when available. - `PreferDispatcherScheduling`: ensures timers queue work via `Dispatcher` instead of busy loops.

Because `UseHeadless()` skips `UsePlatformDetect()`, call it explicitly in tests. For hybrid apps, provide a `BuildAvaloniaApp` overload that chooses headless vs. desktop based on environment.

2. Lifetimes built for tests

Headless apps use `HeadlessLifetime` (see `Avalonia.Headless/HeadlessLifetime.cs`). It mimics `IClassicDesktopStyleApplicationLifetime` but never opens OS windows.

```
public sealed class TestApp : Application
{
    public override void OnFrameworkInitializationCompleted()
    {
```

```
        if (ApplicationLifetime is HeadlessLifetime lifetime)
        {
            lifetime.MainView = new MainView { DataContext = new MainViewModel() };
        }

        base.OnFrameworkInitializationCompleted();
    }
}
```

`HeadlessLifetime` exposes: - `MainView`: root visual displayed inside the headless window implementation.
- `Start()`, `Stop()`: manual control for test harnesses. - `Parameters`: mirrors command-line args.

You can also use `SingleViewLifetime` (`Avalonia.Controls/ApplicationLifetimes/ISingleViewApplicationLifetime.c` for mobile-like scenarios. Headless tests frequently wire both so code mirrors production flows.

Switching lifetimes per environment

```
var builder = Program.BuildAvaloniaApp(enableSkia: true);

if (RuntimeInformation.IsOSPlatform(OSPlatform.Linux) && IsCiAgent)
{
    builder.SetupWithoutStarting();
    using var lifetime = new HeadlessLifetime();
    builder.Instance?.ApplicationLifetime = lifetime;
    lifetime.Start();
}
else
{
    builder.StartWithClassicDesktopLifetime(args);
}
```

`SetupWithoutStarting()` (from `AppBuilderBase`) initializes the app without running the run loop, allowing you to plug in custom lifetimes.

3. Headless application sessions for test frameworks

`HeadlessUnitTestSession` (source: `Avalonia.Headless/HeadlessUnitTestSession.cs`) coordinates app startup across tests so each fixture doesn't rebuild the runtime.

NUnit integration

`Avalonia.Headless.NUnit` ships attributes (`[AvaloniaTest]`, `[AvaloniaTheory]`) that wrap tests in a session. Example test fixture:

```
[AvaloniaTest(Application = typeof(TestApp))]
public class CounterTests
{
    [Test]
    public void Clicking_increment_updates_label()
    {
        using var app = HeadlessUnitTestSession.Start<App>();
        var window = new MainWindow { DataContext = new MainViewModel() };
        window.Show();

        window.FindControl<Button>("IncrementButton")!.RaiseEvent(new RoutedEventArgs(Button.ClickEvent

        window.FindControl<TextBlock>("CounterLabel")!.Text.Should().Be("1");
```

```
    }
}
```

`HeadlessUnitTestSession.Start<TApp>()` spins up the shared app and dispatcher. `FindControl` works because the visual tree exists even though no OS window renders.

xUnit integration

`Avalonia.Headless.XUnit` provides `[AvaloniaFact]` and `[AvaloniaTheory]` attributes. Decorate your test class with `[CollectionDefinition]` to ensure single app instance per collection when running in parallel.

4. Dispatcher, render loops, and timing

Headless rendering still uses Avalonia's dispatcher and render loop. `HeadlessWindowImpl` (source: `Avalonia.Headless/HeadlessWindowImpl.cs`) implements `IWindowImpl` with an in-memory framebuffer. Understanding its behaviour is crucial for deterministic tests.

Forcing layout/render ticks

Headless tests don't run an infinite loop unless you start it. Use `AvaloniaHeadlessPlatform.ForceRenderTimerTick()` to advance timers manually.

```
public static void RenderFrame(TopLevel topLevel)
{
    AvaloniaHeadlessPlatform.ForceRenderTimerTick();
    topLevel.RunJobsOnMainThread();
}
```

`RunJobsOnMainThread()` is a helper extension defined in `HeadlessWindowExtensions`. It drains pending dispatcher work and ensures layout/render happens before assertions.

Simulating async work

Combine `Dispatcher.UIThread.InvokeAsync` with `ForceRenderTimerTick` to await UI updates:

```
await Dispatcher.UIThread.InvokeAsync(() => viewModel.LoadAsync());
AvaloniaHeadlessPlatform.ForceRenderTimerTick();
```

In tests, call `Dispatcher.UIThread.RunJobs()` to flush pending tasks (extension in `Avalonia.Headless` as well).

5. Input, focus, and window services

`HeadlessWindowImpl` implements `IHeadlessWindow`, exposing methods to simulate input:

```
var topLevel = new Window();
var headless = (IHeadlessWindow)topLevel.PlatformImpl!;

headless.MouseMove(new Point(50, 30), RawInputModifiers.None);
headless.MouseDown(new Point(50, 30), MouseButton.Left, RawInputModifiers.LeftMouseButton);
headless.MouseUp(new Point(50, 30), MouseButton.Left, RawInputModifiers.LeftMouseButton);
```

Use extension methods from `HeadlessWindowExtensions` (e.g., `Click(Point)`) to simplify. Focus management works: call `topLevel.Focus()` or `KeyboardDevice.Instance.SetFocusedElement`.

Services like storage providers or dialogs aren't available by default. If your app depends on them, register test doubles in `Application.RegisterServices()`:

```
protected override void RegisterServices()
{
    var services = AvaloniaLocator.CurrentMutable;
    services.Bind<IPlatformLifetimeEvents>().ToConstant(new TestLifetimeEvents());
    services.Bind<IClipboard>().ToSingleton<HeadlessClipboard>();
}
```

`Avalonia.Headless` already provides `HeadlessClipboard`, `HeadlessCursorFactory`, and other minimal implementations; inspect `Avalonia.Headless` folder for available services before writing your own.

6. Rendering options and Skia integration

By default headless renders via CPU copy. To generate bitmaps (Chapter 40), enable Skia:

```
var builder = Program.BuildAvaloniaApp(enableSkia: true);
var options = AvaloniaLocator.Current.GetService<AvaloniaHeadlessPlatformOptions>();
```

When `UseSkia` is true, the backend creates a Skia surface per frame. Ensure the CI environment has the necessary native dependencies (`libSkiaSharp`). If you stick with `UseHeadlessDrawing = true`, `RenderTargetBitmap` still works but without GPU acceleration.

`HeadlessWindowExtensions.CaptureRenderedFrame(topLevel)` captures an `IBitmap` of the latest frame—use it for snapshot tests.

7. Troubleshooting common issues

- **App not initialized**: Ensure `AppBuilder.Configure<App>()` runs before calling `HeadlessUnitTestSession.Start`. Missing static constructor often stems from trimming or linking; mark entry point classes with `[assembly: RequiresUnreferencedCode]` if needed.
- **Dispatcher deadlocks**: Always schedule UI work via `Dispatcher.UIThread`. If a test blocks the UI thread, there's no OS event loop to bail you out.
- **Missing services**: Headless backend only registers core services. Provide mocks for file dialogs, storage, or notifications.
- **Time-dependent tests**: When using timers, call `ForceRenderTimerTick` repeatedly or provide deterministic scheduler wrappers.
- **Memory leaks**: Dispose windows (`window.Close()`) and subscriptions (`CompositeDisposable`) after each test—headless sessions persist across multiple tests by default.

8. Practice lab

1. **Headless bootstrap** – Build a reusable `HeadlessTestApplication` that mirrors your production `App` styles/resources. Verify service registration via unit tests that resolve dependencies from `AvaloniaLocator`.
2. **Lifetime switcher** – Write a helper that starts your app with `HeadlessLifetime` when `DOTNET_RUNNING_IN_CONTAINER` is set. Assert via tests that both classic desktop and headless lifetimes share the same `OnFrameworkInitializationCompleted` flow.
3. **Deterministic render loop** – Create a headless fixture that mounts a view, updates the view-model, calls `ForceRenderTimerTick`, and asserts layout/visual changes with zero sleeps.
4. **Input harness** – Implement extensions wrapping `IHeadlessWindow` for click, drag, and keyboard simulation. Use them to test complex interactions (drag-to-reorder list) without real input devices.
5. **Service fallback** – Provide headless implementations for storage provider and clipboard, inject them in `RegisterServices`, and write tests asserting your UI handles success/failure cases.

Mastering the headless platform ensures Avalonia apps stay testable, portable, and CI-friendly. With lifetimes, options, and input surfaces under your control, you can script rich UI scenarios without ever opening an OS window.

What's next - Next: Chapter39

39. Unit testing view-models and controls headlessly

Goal - Exercise your UI and view-model logic inside real `Dispatcher` loops without opening desktop windows. - Share fixtures and app configuration across xUnit and NUnit by wiring `AvaloniaHeadless` runners correctly. - Simulate input, state changes, and async updates deterministically so assertions stay reliable in CI.

Why this matters - Headless UI tests catch regressions that unit tests miss while remaining fast enough for continuous builds. - Avalonia's dispatcher and property system require a running application instance—adapters handle that for you. - Framework-provided attributes eliminate flaky cross-thread failures and keep tests close to production startup paths.

Prerequisites - Chapter 4 for lifetime selection and `AppBuilder` basics. - Chapter 21 for the bird's-eye view of headless testing capabilities. - Chapter 38 for platform options, dispatcher control, and input helpers.

1. Pick the headless harness

Avalonia ships runner glue for xUnit and NUnit so your test bodies always execute on the UI dispatcher.

xUnit: opt into the Avalonia test framework

Add the assembly-level attribute once and then decorate tests with `[AvaloniaFact]`/`[AvaloniaTheory]`.

```
// AssemblyInfo.cs
using Avalonia.Headless;
using Avalonia.Headless.XUnit;

[assembly: AvaloniaTestApplication(typeof(TestApp))]
[assembly: AvaloniaTestFramework]
```

`AvaloniaTestFramework` (see `external/Avalonia/src/Headless/Avalonia.Headless.XUnit/AvaloniaTestFramework.cs`) installs a custom executor that spawns a `HeadlessUnitTestSession` for the assembly. Each `[AvaloniaFact]` routes through `AvaloniaTestCaseRunner`, ensuring awaited continuations re-enter the dispatcher thread.

NUnit: wrap commands via `[AvaloniaTest]`

```
using Avalonia.Headless;
using Avalonia.Headless.NUnit;

[assembly: AvaloniaTestApplication(typeof(TestApp))]

public class ButtonSpecs
{
    [SetUp]
    public void OpenApp() => Dispatcher.UIThread.VerifyAccess();

    [AvaloniaTest, Timeout(10000)]
    public void Click_updates_counter()
    {
        var window = new Window();
        // ...
    }
}
```

`AvaloniaTestAttribute` swaps NUnit's command pipeline with `AvaloniaTestMethodCommand` (`external/Avalonia/src/He` capturing `SetUp`/`TearDown` delegates and executing them inside the shared dispatcher.

2. Bootstrap the application under test

The harness needs an entry point that mirrors production startup. Reuse your `BuildAvaloniaApp` method or author a lightweight test shell.

```
public class TestApp : Application
{
    public override void OnFrameworkInitializationCompleted()
    {
        Styles.Add(new SimpleTheme());
        base.OnFrameworkInitializationCompleted();
    }

    public static AppBuilder BuildAvaloniaApp() =>
        AppBuilder.Configure<TestApp>()
            .UseSkia()
            .UseHeadless(new AvaloniaHeadlessPlatformOptions
            {
                UseHeadlessDrawing = false, // enable Skia-backed surfaces for rendering checks
                PreferDispatcherScheduling = true
            });
}
```

This pattern matches Avalonia's own tests (`external/Avalonia/tests/Avalonia.Headless.UnitTests/TestApplication.`. When the runner detects `BuildAvaloniaApp`, it invokes it before each dispatch, so your services, themes, and dependency injection mirror the real app. If your production bootstrap already includes `UseHeadless`, the harness respects it; otherwise `HeadlessUnitTestSession.StartNew` injects defaults.

3. Understand session lifetime and dispatcher flow

`HeadlessUnitTestSession` (`external/Avalonia/src/Headless/Avalonia.Headless/HeadlessUnitTestSession.cs`) is the engine behind both harnesses. Highlights:

- `GetOrStartForAssembly` caches a session per test assembly, honoring `[AvaloniaTestApplication]`.
- `Dispatch`/`Dispatch<TResult>` queue work onto the UI thread while keeping NUnit/xUnit's thread blocked until completion.
- `EnsureApplication()` recreates the `AppBuilder` scope for every dispatched action, resetting `Dispatcher` state so tests remain isolated.

You can opt into manual session control when writing custom runners or diagnostics:

```
using var session = HeadlessUnitTestSession.StartNew(typeof(TestApp));
await session.Dispatch(async () =>
{
    var window = new Window();
    window.Show();
    await Dispatcher.UIThread.InvokeAsync(() => window.Close());
}, CancellationToken.None);
```

Dispose the session at the end of a run to stop the dispatcher loop and release the blocking queue.

4. Mount controls and bind view-models

With the dispatcher in place, tests can instantiate real controls, establish bindings, and observe Avalonia's property system.

```
public class CounterTests
{
```

```
    [AvaloniaFact]
    public void Button_click_updates_label()
    {
        var vm = new CounterViewModel();
        var window = new Window
        {
            DataContext = vm,
            Content = new StackPanel
            {
                Children =
                {
                    new Button { Name = "IncrementButton", Command = vm.IncrementCommand },
                    new TextBlock { Name = "CounterLabel", [!TextBlock.TextProperty] = vm.CounterBindin
                }
            }
        };

        window.Show();
        window.MouseDown(new Point(20, 20), MouseButton.Left);
        window.MouseUp(new Point(20, 20), MouseButton.Left);

        window.FindControl<TextBlock>("CounterLabel")!.Text.Should().Be("1");
        window.Close();
    }
}
```

The mouse helpers come from `HeadlessWindowExtensions` (`external/Avalonia/src/Headless/Avalonia.Headless/Headl
They flush pending dispatcher work before delivering input, then run jobs again afterward so bindings update before the assertion. Always `Close()` windows when you finish to keep the session clean.

5. Share fixtures with setup/teardown hooks

Both frameworks let you prepare windows or services per test while staying on the UI thread.

```
public class InputHarness
#if XUNIT
    : IDisposable
#endif
{
    private readonly Window _window;

#if NUNIT
    [SetUp]
    public void SetUp()
#elif XUNIT
    public InputHarness()
#endif
    {
        Dispatcher.UIThread.VerifyAccess();
        _window = new Window { Width = 100, Height = 100 };
    }

#if NUNIT
    [AvaloniaTest]
#elif XUNIT
```

```
    [AvaloniaFact]
#endif
    public void Drag_updates_position()
    {
        _window.Show();
        _window.MouseDown(new Point(10, 10), MouseButton.Left);
        _window.MouseMove(new Point(60, 40));
        _window.MouseUp(new Point(60, 40), MouseButton.Left);
        _window.Position.Should().Be(new PixelPoint(0, 0)); // headless doesn't move windows automatica
    }

#if NUNIT
    [TearDown]
    public void TearDown()
#elif XUNIT
    public void Dispose()
#endif
    {
        Dispatcher.UIThread.VerifyAccess();
        _window.Close();
    }
}
```

The sample mirrors Avalonia's own `InputTests` (`external/Avalonia/tests/Avalonia.Headless.UnitTests/InputTests.`
Use preprocessor guards if you cross-compile the same tests between xUnit and NUnit packages.

6. Keep async work deterministic

Headless tests still depend on Avalonia's dispatcher and timers. Prefer structured helpers over `Task.Delay`.

- `Dispatcher.UIThread.RunJobs()` drains queued operations immediately.
- `AvaloniaHeadlessPlatform.ForceRenderTimerTick()` advances layout and render timers—pair it with `RunJobs()` when you expect visuals to update.
- `DispatcherTimer.RunOnce` works inside tests; the runner ensures the callback fires on the same thread, as shown in `ThreadingTests` (`external/Avalonia/tests/Avalonia.Headless.UnitTests/ThreadingTests.cs`).

```
[AvaloniaFact]
public async Task Loader_raises_progress()
{
    var progress = 0;
    var loader = new AsyncLoader();

    await Dispatcher.UIThread.InvokeAsync(() => loader.Start());

    while (progress < 100)
    {
        AvaloniaHeadlessPlatform.ForceRenderTimerTick();
        Dispatcher.UIThread.RunJobs();
        progress = loader.Progress;
    }

    progress.Should().Be(100);
}
```

If your view-model uses `DispatcherTimer`, expose a hook that ticks manually so tests avoid clock-based flakiness.

7. Theories, collections, and parallelism

`[AvaloniaTheory]` supports data-driven tests while staying on the dispatcher. For xUnit, decorate a collection definition to run related fixtures sequentially:

```
[AvaloniaCollection] // custom marker
public class DialogTests
{
    [AvaloniaTheory]
    [InlineData(false)]
    [InlineData(true)]
    public void Dialog_lifecycle(bool useAsync)
    {
        // ...
    }
}

[CollectionDefinition("AvaloniaCollection", DisableParallelization = true)]
public class AvaloniaCollection : ICollectionFixture<HeadlessFixture> { }
```

The custom fixture can preload services or share the `MainView`. NUnit users can rely on `[Apartment(ApartmentState.STA)]` plus `[AvaloniaTest]` when mixing with other UI frameworks, but remember Avalonia already enforces a single dispatcher thread.

8. Troubleshooting failures

- **Test never finishes** – ensure you awaited async work through `Dispatcher.UIThread` or `HeadlessUnitTestSession.Dispatch`. Background tasks without dispatcher access will hang because the harness blocks the originating test thread.
- **Missing services** – register substitutes in `Application.RegisterServices()` before calling base initialization. Clipboard, dialogs, or storage require headless-friendly implementations (see Chapter 38).
- **State bleed between tests** – close all `TopLevel`s, dispose `CompositeDisposable`s, and avoid static view-model singletons. Each dispatched action gets a fresh `Application` scope, but stray static caches persist.
- **Random `InvalidOperationException: VerifyAccess`** – a test ran code on a thread pool thread. Wrap the block in `Dispatcher.UIThread.InvokeAsync` or use `await session.Dispatch(...)` in custom helpers.
- **Parallel collection deadlocks** – turn off test parallelism when fixtures share windows. xUnit: `[assembly: CollectionBehavior(DisableTestParallelization = true)]`; NUnit: `--workers=1` or `[NonParallelizable]` per fixture.

Practice lab

1. **Session helper** – Write a reusable `HeadlessTestSessionFixture` exposing `Dispatch(Func<Task>)` so plain unit tests can invoke dispatcher-bound code without attributes.
2. **View-model assertions** – Mount a form with compiled bindings, trigger `BindingOperations` updates, and assert validation errors surface via `DataValidationErrors.GetErrors`.
3. **Keyboard automation** – Use `HeadlessWindowExtensions.KeyPressQwerty` to simulate typing into a `TextBox`, verify selection state, then assert command execution when pressing Enter.
4. **Timer-driven UI** – Create a progress dialog using `DispatcherTimer`. In tests, tick the timer manually and assert the dialog closes itself at 100% without sleeping.
5. **Theory matrix** – Build a `[AvaloniaTheory]` test that runs the same control suite using Classic Desktop vs. Single View lifetimes by swapping `HeadlessLifetime.MainView`. Confirm both paths render identical text through `GetLastRenderedFrame()`.

What's next - Next: Chapter40

40. Rendering verification and pixel assertions

Goal - Capture deterministic frames from controls and windows so UI regressions show up as image diffs. - Re-render visuals off-screen with `RenderTargetBitmap` for pipeline-level validation without a running window. - Build comparison utilities that tolerate minor noise while still failing on real regressions.

Why this matters - Visual bugs rarely surface through property assertions alone; pixel diffs make style and layout drift obvious. - CI agents run headless—leveraging Avalonia's off-screen renderers keeps comparison workflows portable. - Consistent capture pipelines simplify storing baselines, reviewing diffs, and onboarding QA to UI automation.

Prerequisites - Chapter 21 for the overview of headless testing options. - Chapter 38 for dispatcher control and headless render ticks. - Chapter 39 for running xUnit/NUnit fixtures on the Avalonia dispatcher.

1. Capture frames from headless top levels

`HeadlessWindowExtensions.CaptureRenderedFrame` (`external/Avalonia/src/Headless/Avalonia.Headless/Headless`
flushes the dispatcher, ticks the headless timer, and returns a `WriteableBitmap` of the latest frame. The helper delegates to `GetLastRenderedFrame`, which requires Skia-backed rendering—set `UseHeadlessDrawing = false` and `UseSkia = true` in your test app:

```
public static AppBuilder BuildAvaloniaApp() =>
    AppBuilder.Configure<TestApp>()
        .UseHeadless(new AvaloniaHeadlessPlatformOptions
        {
            UseHeadlessDrawing = false,
            UseSkia = true,
            PreferDispatcherScheduling = true
        });
```

Once configured, capture snapshots straight from a headless window:

```
var window = new Window
{
    Content = new ControlCatalogPage(),
    SizeToContent = SizeToContent.WidthAndHeight
};

window.Show();
var frame = window.CaptureRenderedFrame();
Assert.NotNull(frame);
```

Avalonia's own regression tests follow this pattern (`external/Avalonia/tests/Avalonia.Headless.UnitTests/RenderingT`
Use `CaptureRenderedFrame` when you want the helper to tick timers for you; call `GetLastRenderedFrame` if you have already driven the dispatcher manually.

2. Render visuals off-screen with `RenderTargetBitmap`

To avoid constructing full windows, target the visual tree directly. `RenderTargetBitmap` uses `ImmediateRenderer.Render` under the hood (`external/Avalonia/src/Avalonia.Base/Media/Imaging/RenderTargetBitr`

```
var root = new Border
{
    Width = 200,
    Height = 120,
    Background = Brushes.CornflowerBlue,
    Child = new TextBlock
    {
```

```
        Text = "Hello Avalonia",
        FontSize = 24,
        HorizontalAlignment = HorizontalAlignment.Center,
        VerticalAlignment = VerticalAlignment.Center
    }
};

await Dispatcher.UIThread.InvokeAsync(() => root.Measure(new Size(double.PositiveInfinity, double.Positi
root.Arrange(new Rect(root.DesiredSize));

using var rtb = new RenderTargetBitmap(new PixelSize(200, 120));
rtb.Render(root);
```

The bitmap implements `IBitmap`, so you can save it, compare pixels, or embed it in diagnostics emails. For complex compositions, grab a `DrawingContext` from `RenderTargetBitmap.CreateDrawingContext` to draw primitive overlays before comparison.

3. Compare pixels with configurable tolerances

Whether you use `CaptureRenderedFrame` or `RenderTargetBitmap`, lock the frame buffer to access raw bytes. `WriteableBitmap.Lock()` exposes an `ILockedFramebuffer` with stride, format, and a pointer into the pixel buffer (`external/Avalonia/src/Avalonia.Base/Media/Imaging/WriteableBitmap.cs:59`).

```
public static PixelDiffResult CompareBitmaps(IBitmap expected, IBitmap actual, byte tolerance = 2)
{
    using var left = expected.Lock();
    using var right = actual.Lock();

    if (left.Size != right.Size)
        return PixelDiffResult.SizeMismatch(left.Size, right.Size);

    var failures = new List<PixelDiff>();

    unsafe
    {
        for (var y = 0; y < left.Size.Height; y++)
        {
            var pLeft = (byte*)left.Address + y * left.RowBytes;
            var pRight = (byte*)right.Address + y * right.RowBytes;

            for (var x = 0; x < left.Size.Width; x++)
            {
                var idx = x * 4; // BGRA
                var delta = Math.Max(
                    Math.Abs(pLeft[idx] - pRight[idx]),
                    Math.Max(Math.Abs(pLeft[idx + 1] - pRight[idx + 1]),
                             Math.Abs(pLeft[idx + 2] - pRight[idx + 2])));

                if (delta > tolerance)
                    failures.Add(new PixelDiff(x, y, delta));
            }
        }
    }

    return PixelDiffResult.FromList(failures);
```

```
}
```

Tune the tolerance to absorb small antialiasing differences. Consider summing absolute channel differences or using the Delta-E metric when gradients highlight sub-pixel drift.

Produce diagnostic overlays

When differences occur, create an error bitmap that highlights only changed pixels:

```
public static WriteableBitmap CreateDiffMask(IBitmap baseline, PixelDiffResult result)
{
    var size = baseline.PixelSize;
    var diff = new WriteableBitmap(size, baseline.Dpi); // default BGRA32

    using var target = diff.Lock();
    var buffer = new Span<byte>((void*)target.Address, target.RowBytes * size.Height);
    buffer.Clear();

    foreach (var pixel in result.Failures)
    {
        var idx = pixel.Y * target.RowBytes + pixel.X * 4;
        buffer[idx + 0] = 0;          // B
        buffer[idx + 1] = 0;          // G
        buffer[idx + 2] = 255;        // R highlights
        buffer[idx + 3] = 255;        // A
    }

    return diff;
}
```

Attach the original frame, baseline, and diff mask to CI artifacts so reviewers can inspect regressions quickly.

4. Manage baselines and golden images

Golden images can live alongside tests as embedded resources. Load them via `WriteableBitmap.Decode` and normalize configuration before comparison:

```
await using var stream = manifestAssembly.GetManifestResourceStream("Tests.Baselines.Dialog.png");
var baseline = WriteableBitmap.Decode(stream!);
```

When baselines must be refreshed, capture a new frame and save it to disk using `frame.Save(fileStream)`. Normalize DPI and render scaling so new baselines remain cross-platform:

```
var normalized = new RenderTargetBitmap(new PixelSize(800, 600), new Vector(96, 96));
normalized.Render(window);
await using var file = File.Create("Baselines/Dialog.png");
normalized.Save(file);
```

`RenderTargetBitmapImpl` uses Skia surfaces (`external/Avalonia/src/Skia/Avalonia.Skia/RenderTargetBitmapImpl.cs`) so CI agents must have the Skia native bundle available. If you target platforms without GPU support, stick to headless captures with `UseHeadlessDrawing = true` and fall back to `WriteableBitmap` comparisons.

5. Handle DPI, alpha, and layout variability

Visual tests are sensitive to device-independent rounding. Lock down inputs:

- Set explicit window sizes and call `SizeToContent = WidthAndHeight` to avoid layout fluctuations.
- Fix `RenderScaling` by pinning `UseHeadlessDrawing` and Skia DPI to 96.

- Strip alpha when comparing controls that rely on transparency to avoid background differences. Copy pixels into a new bitmap with an opaque fill before diffing.

For dynamic content (animations, timers), tick the dispatcher deterministically: call `AvaloniaHeadlessPlatform.ForceRende` between each capture, and pause transitions via `IClock` injection so frames stay stable.

Leverage composition snapshots when you need sub-tree captures: `Compositor.CreateCompositionVisualSnapshot` returns a GPU-rendered image of any `Visual` (`external/Avalonia/tests/Avalonia.Headless.UnitTests/RenderingTest` Convert the snapshot to `WriteableBitmap` for comparisons if you want to isolate specific effects layers.

6. Troubleshooting

- **`GetLastRenderedFrame` throws** – ensure Skia is active; the helper checks for `HeadlessPlatformRenderInterface` and fails when only headless drawing is enabled.
- **Alpha mismatches** – multiply against a known background before diffing. Render your control inside a `Border` with a solid color or premultiply the buffer before comparison.
- **Different stride values** – always use `ILockedFramebuffer.RowBytes` instead of assuming width × 4 bytes.
- **Platform font differences** – embed test fonts or ship them with the test harness so text metrics remain identical across agents.
- **Large golden files** – compress PNGs with `optipng` or generate vector baselines by storing the render input (XAML/data) alongside the image for easier review.

Practice lab

1. **Snapshot harness** – Build a `PixelAssert.Capture(window)` helper that returns baseline, actual, and diff images, then integrates them with your test framework's logging.
2. **Tolerance sweeper** – Write a diagnostic that runs the same render with multiple tolerances, reporting how many pixels fail each threshold to help pick a sensible default.
3. **Golden management** – Implement a CLI command that regenerates baselines from the latest controls, writes them to disk, and updates a manifest listing checksum + control name.
4. **Alpha neutralization** – Add a utility that composites captured frames over a configurable background color before comparison, and verify it fixes regressions caused by transparent overlays.
5. **Snapshot localization** – Capture the same view under different resource cultures and ensure your comparison harness accepts localized text while still flagging layout drift.

What's next - Next: Chapter41

41. Simulating input and automation in headless runs

Goal - Drive Avalonia UI interactions programmatically inside headless tests, mirroring real user gestures. - Coordinate keyboard, pointer, and text input events through `HeadlessWindowExtensions` so focus and routing behave exactly as on desktop. - Assert downstream automation effects—commands, behaviors, drag/drop—without launching OS-level windows.

Why this matters - Interactive flows (menus, drag handles, keyboard shortcuts) break easily if you only test bindings or view-models; simulated input keeps coverage honest. - CI agents lack real hardware. The headless platform proxies devices so you can rehearse full user journeys deterministically. - Automation/UIP frameworks often rely on routed events and focus transitions; reproducing them in tests prevents last-minute surprises.

Prerequisites - Chapter 38 for headless dispatcher control and platform options. - Chapter 39 for integrating Avalonia's headless test attributes in xUnit or NUnit. - Chapter 40 if you plan to pair input simulation with pixel verification.

1. Meet the headless input surface

Every headless `TopLevel` implements `IHeadlessWindow` (`external/Avalonia/src/Headless/Avalonia.Headless/IHeadles` exposing methods for keyboard, pointer, wheel, and drag/drop events. `HeadlessWindowExtensions` (`external/Avalonia/src/Headless/Avalonia.Headless/HeadlessWindowExtensions.cs:20`) wraps those APIs, handling dispatcher ticks before and after each gesture so routed events fire on time.

```
var window = new Window { Content = new Button { Content = "Click me" } };
window.Show();

window.MouseMove(new Point(20, 20));
window.MouseDown(new Point(20, 20), MouseButton.Left);
window.MouseUp(new Point(20, 20), MouseButton.Left);
```

Under the hood the extension flushes outstanding work (`Dispatcher.UIThread.RunJobs()`), triggers the render timer (`AvaloniaHeadlessPlatform.ForceRenderTimerTick()`), invokes the requested gesture on the `IHeadlessWindow`, and drains the dispatcher again. This ensures property changes, focus updates, and automation events complete before your assertions run.

2. Keyboard and text input

`HeadlessWindowExtensions` provides multiple helpers for synthesizing key strokes:

- `KeyPress`/`KeyRelease` accept logical `Key` values plus `RawInputModifiers`.
- `KeyPressQwerty`/`KeyReleaseQwerty` map physical scan codes to logical keys using a QWERTY layout.
- `KeyTextInput` sends text composition events directly to controls that listen for `TextInput`.

```
var textBox = new TextBox { AcceptsReturn = true };
var window = new Window { Content = textBox };
window.Show();
textBox.Focus();

window.KeyPressQwerty(PhysicalKey.KeyH, RawInputModifiers.Shift);
window.KeyPressQwerty(PhysicalKey.KeyI, RawInputModifiers.None);
window.KeyReleaseQwerty(PhysicalKey.Enter, RawInputModifiers.None);
window.KeyTextInput("!");

textBox.Text.Should().Be("Hi!\n");
```

Avalonia routes the events through `KeyboardDevice` so controls experience the same bubbling/tunneling as in production. Remember to set focus explicitly (`textBox.Focus()` or `KeyboardDevice.Instance.SetFocusedElement`)

before typing—headless windows do not auto-focus when shown.

3. Pointer gestures and drag/drop

Mouse helpers cover move, button transitions, wheel scrolling, and drag/drop scenarios. The headless platform maintains a single virtual pointer (`HeadlessWindowImpl` uses `PointerDevice`, see `external/Avalonia/src/Headless/Avalonia.Headless/HeadlessWindowImpl.cs:34`).

```
var listBox = new ListBox
{
    ItemsSource = new[] { "Alpha", "Beta", "Gamma" }
};
var window = new Window { Content = listBox };
window.Show();

// Click first item
window.MouseMove(new Point(10, 20));
window.MouseDown(new Point(10, 20), MouseButton.Left);
window.MouseUp(new Point(10, 20), MouseButton.Left);
listBox.SelectedIndex.Should().Be(0);

// Scroll down
window.MouseWheel(new Point(10, 20), new Vector(0, -120));
```

For drag/drop, build a `DataObject` and send a sequence of drag events:

```
var data = new DataObject();
data.Set(DataFormats.Text, "payload");
window.DragDrop(new Point(10, 20), RawDragEventType.DragEnter, data, DragDropEffects.Copy);
window.DragDrop(new Point(80, 40), RawDragEventType.DragOver, data, DragDropEffects.Copy);
window.DragDrop(new Point(80, 40), RawDragEventType.Drop, data, DragDropEffects.Copy);
```

Your controls will receive `DragEventArgs`, invoke drop handlers, and update view-models just as they would with real user input.

4. Focus, capture, and multi-step workflows

Headless tests still rely on Avalonia's focus and capture services:

- Call `control.Focus()` or `FocusManager.Instance.Focus(control)` before keyboard entry.
- Pointer capture happens automatically when a control handles `PointerPressed` and calls `e.Pointer.Capture(control)`. To assert capture, inspect `Pointer.Captured` inside your test after dispatching input.
- Release capture manually with `pointer.Capture(null)` when simulating complex gestures to avoid stale state.

Example: testing a custom drag handle that requires capture and modifier keys.

```
[AvaloniaFact]
public void DragHandle_updates_offset()
{
    var handle = new DragHandleControl();
    var window = new Window { Content = handle };
    window.Show();

    window.MouseMove(new Point(5, 5));
    window.MouseDown(new Point(5, 5), MouseButton.Left, RawInputModifiers.LeftMouseButton);
    handle.PointerIsCaptured.Should().BeTrue();
```

```
    window.MouseMove(new Point(45, 5), RawInputModifiers.LeftMouseButton | RawInputModifiers.Shift);
    window.MouseUp(new Point(45, 5), MouseButton.Left);

    handle.Offset.Should().BeGreaterThan(0);
}
```

Because `HeadlessWindowExtensions` executes all gestures on the UI thread, your control can update dependency properties, trigger animations, and publish events synchronously within the test.

5. Compose higher-level automation helpers

Most suites wrap common interaction patterns in reusable functions to keep tests declarative:

```
public sealed class HeadlessUser
{
    private readonly Window _window;
    public HeadlessUser(Window window) => _window = window;

    public void Click(Control control)
    {
        var point = control.TranslatePoint(new Point(control.Bounds.Width / 2, control.Bounds.Height / 
        _window.MouseMove(point);
        _window.MouseDown(point, MouseButton.Left);
        _window.MouseUp(point, MouseButton.Left);
    }

    public void Type(string text)
    {
        foreach (var ch in text)
            _window.KeyTextInput(ch.ToString());
    }
}
```

Pair these helpers with assertions against `AutomationProperties` to verify accessibility metadata as you drive the UI. Tests in `external/Avalonia/tests/Avalonia.Headless.UnitTests/InputTests.cs:29` demonstrate structuring fixtures that open a window in `[SetUp]`/constructor, execute gestures, and dispose deterministically.

6. Raw input modifiers and multiple devices

`RawInputModifiers` combines buttons, keyboard modifiers, and touch states into a single bit field. Use it to emulate complex shortcuts:

```
window.MouseDown(point, MouseButton.Left, RawInputModifiers.LeftMouseButton | RawInputModifiers.Control
window.KeyPress(Key.S, RawInputModifiers.Control, PhysicalKey.KeyS, "s");
```

Headless currently exposes a single mouse pointer and keyboard. To simulate multi-pointer scenarios (e.g., pinch gestures), create custom `RawPointerEventArgs` and push them through `InputManager.Instance.ProcessInput`. That advanced path uses `IInputRoot.Input` (hook available via `HeadlessWindowImpl.Input`), giving you full control when default helpers are insufficient.

7. Troubleshooting

- **No events firing** – confirm you called `window.Show()` and that the target control is in the visual tree. Without showing, the platform impl doesn't attach an `InputRoot`.

- **Focus lost between gestures** – check whether your control closes popups or dialogs. Re-focus before continuing or assert against `FocusManager.Instance.Current`.
- **Pointer coordinates off** – convert control-relative coordinates to window coordinates (`TranslatePoint`) and double-check logical vs. visual point units (headless always uses logical units, scaling = 1 unless you override).
- **Keyboard text missing** – some controls ignore `KeyTextInput` without focus or when `AcceptsReturn` is false. Set the right properties or use `TextInputOptions` when testing IME handling (`external/Avalonia/src/Avalonia.Base/Input/TextInput/TextInputOptions.cs`).
- **Drag/drop crashes** – make sure Skia is enabled for capture-heavy tests and that you dispose `DataObject` content streams after the drop completes.

Practice lab

1. **User DSL** – Build a `HeadlessUser` helper that supports click, double-click, context menu, typing, and modifier-aware shortcuts. Use it to script multi-page navigation flows.
2. **Pointer capture assertions** – Write a test that verifies a custom canvas captures the pointer during drawing and releases it when `PointerReleased` fires, asserting against `Pointer.Captured`.
3. **Keyboard navigation** – Simulate `Tab`/`Shift+Tab` sequences across a dialog and assert `FocusManager.Instance.Current` to ensure accessibility order is correct.
4. **Drag/drop harness** – Create reusable helpers for `DragEnter`/`DragOver`/`Drop` with specific `IDataObject` payloads. Verify your view-model receives the right data and that effects (`DragDropEffects`) match expectations.
5. **IME/text services** – Toggle `TextInputOptions` on a `TextBox`, send mixed `KeyPress` and `KeyTextInput` events, and confirm composition events surface in your view-model for languages requiring IME support.

What's next - Next: Chapter42

42. CI pipelines, diagnostics, and troubleshooting

Goal - Run Avalonia headless and automation suites reliably in CI across Windows, macOS, and Linux agents. - Capture logs, screenshots, and diagnostics artifacts so UI regressions are easy to triage. - Detect hangs or ordering issues proactively and keep runs deterministic even under heavy concurrency.

Why this matters - UI regressions usually surface first in automation—if the pipeline flakes, the team stops trusting the signal. - Headless tests rely on the dispatcher and render loop; CI environments with limited GPUs or desktops need deliberate setup. - Rich artifacts (logs, videos, dumps) turn red builds into actionable bug reports instead of mystery failures.

Prerequisites - Chapter 38 for configuring `UseHeadless` and driving the dispatcher. - Chapter 39 for integrating the headless test session into xUnit or NUnit. - Chapter 41 for scripting complex input sequences that your pipeline will exercise.

1. Pick a CI host and bootstrap prerequisites

Avalonia's own integration pipeline (see `external/Avalonia/azure-pipelines-integrationtests.yml:1`) demonstrates the moving parts for Appium + headless test runs:

- Install the correct .NET runtimes/SDKs via `UseDotNet@2`.
- Prepare platform dependencies (e.g., select Xcode, kill stray `node` processes, start Appium on macOS; start WinAppDriver on Windows).
- Build the test app and run `dotnet test` against `Avalonia.IntegrationTests.Appium.csproj`.
- Publish artifacts—`appium.out` on failure and TRX results on all outcomes.

For GitHub Actions, mirror that setup with runner-specific steps:

```
jobs:
  ui-tests:
    strategy:
      matrix:
        os: [windows-latest, macos-13]
    runs-on: ${{ matrix.os }}
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-dotnet@v3
        with:
          global-json-file: global.json
      - name: Start WinAppDriver
        if: runner.os == 'Windows'
        run: Start-Process -FilePath 'C:\\Program Files (x86)\\Windows Application Driver\\WinAppDriver
      - name: Restore
        run: dotnet restore tests/Avalonia.Headless.UnitTests
      - name: Test headless suite
        run: dotnet test tests/Avalonia.Headless.UnitTests --logger "trx;LogFileName=headless.trx" --bla
      - name: Publish results
        if: always()
        uses: actions/upload-artifact@v4
        with:
          name: headless-results
          path: '**/*.trx'
```

Adjust the matrix for Linux when you only need headless tests (no Appium). Use the same `dotnet test` command locally to validate pipeline scripts.

2. Configure deterministic test execution

Headless suites should run with parallelism disabled unless every fixture is isolation-safe. xUnit supports assembly-level configuration:

```
// AssemblyInfo.cs
[assembly: CollectionBehavior(DisableTestParallelization = true)]
[assembly: AvaloniaTestFramework]
```

Pair the attribute with `AvaloniaTestApplication` so a single `HeadlessUnitTestSession` drives the whole assembly. For NUnit, launch the test runner with `--workers=1` or mark fixtures `[NonParallelizable]`. This avoids fighting over the singleton dispatcher and ensures actions happen in the same order on developer machines and CI bots.

Within tests, drain work deterministically. `HeadlessWindowExtensions` already wraps each gesture with `Dispatcher.UIThread.RunJobs()` and `AvaloniaHeadlessPlatform.ForceRenderTimerTick()`; call those directly from helpers when you schedule background tasks outside the provided wrappers.

3. Capture logs, screenshots, and videos

Collect evidence automatically so failing builds are actionable:

- Turn on Avalonia's trace logging by chaining `.LogToTrace()` in your `AppBuilder`. Redirect stderr to a file in CI (`dotnet test … 2> headless.log`) and upload it as an artifact.
- Use `CaptureRenderedFrame` (Chapter 40) to grab before/after bitmaps on failure. Save them with a timestamp inside `TestContext.CurrentContext.WorkDirectory` (NUnit) or `ITestOutputHelper` attachments (xUnit).
- On Windows, record screen captures with MSTest data collectors. Avalonia ships `record-video.runsettings` (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/record-video.runsettings:1`) to capture Appium sessions; reuse it by passing `/Settings:record-video.runsettings` to VSTest or `--settings` to `dotnet test`.
- For Appium runs, write driver logs to disk. The macOS pipeline publishes `appium.out` when a job fails (`external/Avalonia/azure-pipelines-integrationtests.yml:27`).

4. Diagnose hangs and deadlocks

UI tests occasionally hang because outstanding work blocks the dispatcher. Harden your pipeline with diagnosis options:

- Use `dotnet test --blame-hang-timeout 5m --blame-hang-dump-type full` to trigger crash dumps when a test exceeds the timeout.
- Wrap long-running awaits inside `HeadlessUnitTestSession.Dispatch` so the framework can pump the dispatcher (`external/Avalonia/src/Headless/Avalonia.Headless/HeadlessUnitTestSession.cs:54`).
- Expose a helper that runs `Dispatcher.UIThread.RunJobs()` and `AvaloniaHeadlessPlatform.ForceRenderTimerTic` in a loop until a condition is met. Fail the test if the condition never becomes true to avoid infinite waits.
- When debugging locally, attach a logger to `DispatcherTimer` callbacks or set `DispatcherTimer.Tag` to identify timers causing hangs; the headless render timer is labeled `HeadlessRenderTimer` (`external/Avalonia/src/Headless/Avalonia.Headless/AvaloniaHeadlessPlatform.cs:21`).

Analyze captured dumps with `dotnet-dump analyze` to inspect managed thread stacks and spot blocked tasks.

5. Environment hygiene on shared agents

CI agents often reuse workspaces. Add cleanup steps before running UI automation:

- Kill straggling processes (`pkill IntegrationTestApp`, `pkill node`) as the macOS pipeline does (`external/Avalonia/azure-pipelines-integrationtests.yml:21`).
- Remove stale app bundles or temporary data to guarantee a clean run.
- Reset environment variables that influence Avalonia behavior (e.g., `AVALONIA_RENDERER` overrides). Keep your scripts explicit to avoid surprises when infra engineers tweak images.

For cross-platform Appium tests, encapsulate capability setup in fixtures. `DefaultAppFixture` (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/DefaultAppFixture.cs:9`) configures Windows and macOS sessions differently while exposing a consistent driver to tests.

6. Build health dashboards and alerts

Publish TRX or NUnit XML outputs to your CI system so failures appear in dashboards. Azure Pipelines uses `PublishTestResults@2` to ingest xUnit results even when the job succeeds with warnings (`external/Avalonia/azure-pipelines-integrationtests.yml:67`). GitHub Actions can read TRX via `dorny/test-reporter` or similar actions.

Send critical logs to observability tools if your team maintains telemetry infrastructure. A simple approach is to push structured log lines to stdout in JSON—CI services preserve the console by default.

7. Troubleshooting checklist

- **Tests fail only on CI** – compare fonts, localization, and DPI. Ensure custom fonts are deployed with the test app and `CultureInfo.DefaultThreadCurrentUICulture` is set for deterministic layouts.
- **Intermittent hangs** – add `--blame` dumps, then review stuck threads. Often a test awaited `Task.Delay` without advancing the render timer; replace with deterministic loops.
- **Missing screenshots** – confirm Skia is enabled (`UseHeadlessDrawing = false`) so `CaptureRenderedFrame` works in pipelines.
- **Appium session errors** – verify the automation server is running (WinAppDriver/Appium) before tests start, and stop it in a final step to avoid port conflicts next run.
- **Resource leaks across tests** – always close windows (`window.Close()`), dispose `CompositeDisposable`, and tear down Appium sessions in `Dispose`. Lingering windows keep the dispatcher alive and can cause later tests to inherit state.

Practice lab

1. **Pipeline parity** – Create a local script that mirrors your CI job (`dotnet restore`, `dotnet test`, artifact copy). Run it before pushing so pipeline failures never surprise you.
2. **Hang detector** – Wire `dotnet test --blame` into your CI job and practice analyzing the generated dumps for a deliberately hung test.
3. **Artifact triage** – Extend your test harness to save headless screenshots and logs into an output directory, then configure your pipeline to upload them on failure.
4. **Parallelism audit** – Temporarily enable test parallelization to identify fixtures that rely on global state. Fix the offenders or permanently disable parallel runs via assembly attributes.
5. **Cross-platform dry run** – Use a GitHub Actions matrix or Azure multi-job pipeline to run headless tests on Windows and Linux simultaneously, comparing logs for environment-specific quirks.

What's next - Next: Chapter43

43. Appium fundamentals for Avalonia apps

Goal - Stand up Appium-based UI tests that drive Avalonia desktop apps on Windows and macOS. - Reuse the built-in integration harness (`Avalonia.IntegrationTests.Appium`) to spin sessions, navigate the sample app, and locate controls reliably. - Understand the accessibility surface Avalonia exposes so selectors stay stable across platforms and Appium versions.

Why this matters - End-to-end coverage validates window chrome, dialogs, and platform behaviors that headless tests can't touch. - Appium works with the same accessibility tree users rely on—tests that pass here give confidence in automation readiness. - A disciplined harness keeps session setup, synchronization, and cleanup consistent across operating systems.

Prerequisites - Chapter 12 for windowing concepts referenced by Appium tests. - Chapter 13 for menus/dialogs—the automation harness exercises them heavily. - Chapter 42 for CI orchestration once your Appium suite is green locally.

1. Anatomy of the Avalonia Appium harness

Avalonia ships an Appium test suite in `external/Avalonia/tests/Avalonia.IntegrationTests.Appium`. Key parts:

- `DefaultAppFixture` builds and launches the sample `IntegrationTestApp`, creating an `AppiumDriver` for Windows or macOS sessions (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/DefaultAppFixtu`
- `TestBase` accepts the fixture and navigates the ControlCatalog-style pager. It retries the navigation click to absorb macOS animations (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/TestBase.cs:6`
- `CollectionDefinitions` wires fixtures into xUnit collections so sessions are shared per test class (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/CollectionDefinitions.cs:4`).

Reuse this structure in your own project: create a fixture that launches your app (packaged exe/bundle), expose the `AppiumDriver`, and derive page-specific test classes from a `TestBase` that performs navigation.

2. Configure sessions per platform

`DefaultAppFixture` populates capability sets tailored to each OS:

```
var options = new AppiumOptions();
if (OperatingSystem.IsWindows())
{
    options.AddAdditionalCapability(MobileCapabilityType.App, TestAppPath);
    options.AddAdditionalCapability(MobileCapabilityType.PlatformName, MobilePlatform.Windows);
    options.AddAdditionalCapability(MobileCapabilityType.DeviceName, "WindowsPC");
    Session = new WindowsDriver(new Uri("http://127.0.0.1:4723"), options);
}
else if (OperatingSystem.IsMacOS())
{
    options.AddAdditionalCapability("appium:bundleId", "net.avaloniaui.avalonia.integrationtestapp");
    options.AddAdditionalCapability(MobileCapabilityType.PlatformName, MobilePlatform.MacOS);
    options.AddAdditionalCapability(MobileCapabilityType.AutomationName, "mac2");
    Session = new MacDriver(new Uri("http://127.0.0.1:4723/wd/hub"), options);
}
```

The fixture also foregrounds the window on Windows via `SetForegroundWindow` to avoid focus issues. Always close the session in `Dispose` even if Appium errors—macOS' `mac2` driver may throw on shutdown, so wrap in try/catch like the sample.

TIP: keep Appium/WAD endpoints configurable via environment variables so your CI scripts can point to remote device clouds.

3. Navigating the sample app

`TestBase` selects a page by finding the pager control and clicking the relevant button. The same pattern applies to your app:

```
public class WindowTests : TestBase
{
    public WindowTests(DefaultAppFixture fixture) : base(fixture, "Window") { }

    [Fact]
    public void Can_toggle_window_state()
    {
        var windowStateCombo = Session.FindElementByAccessibilityId("CurrentWindowState");
        windowStateCombo.Click();
        Session.FindElementByAccessibilityId("WindowStateMaximized").SendClick();
        Assert.Equal("Maximized", windowStateCombo.GetComboBoxValue());
    }
}
```

The `pageName` passed to `TestBase` must match the accessible name exposed by the pager button. Avalonia's sample ControlCatalog sets these via `AutomationProperties.Name`, so always annotate navigation controls in your app for consistent selectors.

4. Element discovery and helper APIs

Selectors vary subtly across platforms. Avalonia's helpers hide those differences:

- `AppiumDriverEx` defines `FindElementByAccessibilityId`, `FindElementByName`, and other convenience methods to work with both Appium 1 and 2 (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/`
- `ElementExtensions` centralizes common queries such as chrome buttons and combo box value extraction. For example, `GetComboBoxValue` uses `Text` on Windows and `value` attributes elsewhere (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/ElementExtensions.cs:34`).
- `GetCurrentSingleWindow` hides the extra wrapper window present in macOS accessibility trees (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/ElementExtensions.cs:60`).

When building your suite, add similar extension methods instead of hard-coding XPath per test. Keep selectors rooted in `AutomationId` or names you control via `AutomationProperties.AutomationId` and `Name` to minimize brittleness.

5. Synchronization and retries

Appium commands are asynchronous relative to the app. Avalonia tests mix explicit waits, retries, and timeouts:

- `TestBase` retries page navigation three times with a 1s delay to survive macOS transitions.
- `ElementExtensions.OpenWindowWithClick` polls for either a new window handle or child window to appear, retrying up to ten times (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/ElementExtensio`
- For transitions with animations (e.g., exiting full screen), tests call `Thread.Sleep` after sending commands—note the cleanup block in `WindowTests` that waits 1 second on macOS before asserting (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/WindowTests.cs:53`).

Wrap these patterns in helper methods so timing tweaks stay centralized. For more resilient waits, use Appium's `WebDriverWait` with conditions such as `driver.FindElementByAccessibilityId(...)` or `element.Displayed`.

6. Cross-platform control with attributes and collections

Automation suites often need OS-specific assertions. Avalonia uses:

- `[PlatformFact]` / `[PlatformTheory]` to skip tests on unsupported OSes (`external/Avalonia/tests/Avalonia.Integ`
- Collection definitions to isolate fixtures for specialized apps (e.g., overlay popups vs. default ControlCatalog) (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/CollectionDefinitions.cs:10`).

Follow suit by tagging tests with custom attributes that read environment variables or capability flags. This keeps your suite from failing on agents lacking certain features (e.g., Win32-only APIs).

7. Exposing automation IDs in Avalonia

Appium relies on the accessibility tree. Avalonia maps these properties as follows:

- `AutomationProperties.AutomationId` and control `Name` become accessibility IDs (`AutomationTests.AutomationId`, `external/Avalonia/tests/Avalonia.IntegrationTests.Appium/AutomationTests.cs:12`).
- `AutomationProperties.Name` populates the element name in both Windows UIA and macOS accessibility APIs.
- `AutomationProperties.LabeledBy` and other metadata surface via Appium attributes so you can assert associations (`AutomationTests.LabeledBy`).

Ensure the controls you interact with set both `AutomationId` and `Name`; for templated controls expose IDs through `x:Name` or `Automation.Id`. Without these properties, selectors fall back to fragile XPath queries.

8. Running the suite

Windows

1. Install WinAppDriver (ships with Visual Studio workloads) and start it on port 4723.
2. Build your Avalonia app for `net8.0-windows` with `UseWindowsForms` disabled (the sample uses `IntegrationTestApp`).
3. Launch Appium tests: `dotnet test tests/Avalonia.IntegrationTests.Appium --logger "trx;LogFileName=appium.trx"`.

macOS

1. Install Appium 2 with the `mac2` driver and run `appium --base-path /wd/hub`.
2. Ensure the test runner has accessibility permissions; the pipeline script resets them via `pkill` and `osascript` (`external/Avalonia/azure-pipelines-integrationtests.yml:17`).
3. Bundle the app (`samples/IntegrationTestApp/bundle.sh`) so Appium can reference it by bundle ID.

Use the provided `macos-clean-build-test.sh` as a reference for orchestrating builds locally or in CI.

9. Troubleshooting

- **Session fails to start** – Verify the Appium server is running and that the path/bundle ID is correct. On Windows, ensure the test app exists relative to the test project (`DefaultAppFixture.TestAppPath`).
- **Elements not found** – Inspect the accessibility tree with tools such as Windows Inspect or macOS Accessibility Inspector. Add missing `AutomationId` values to the Avalonia XAML.
- **Focus issues after fullscreen** – Mirror Avalonia's retry `Thread.Sleep` or use explicit waits; macOS may animate transitions for up to a second.
- **Multiple windows** – Use `OpenWindowWithClick` helper to track handles. Remember to dispose the returned `IDisposable` so the new window closes after the test.
- **Driver shutdown crashes** – Wrap `Session.Close()` in try/catch like `DefaultAppFixture.Dispose` to shield flaky platform drivers.

Practice lab

1. **Custom fixture** – Implement a fixture that launches your app under test, parameterized by environment variables for executable path and Appium endpoint.
2. **Navigation helper** – Create a `TestBase` that navigates your shell's menu/pager via automation IDs, then write a smoke test asserting window title, version label, or status bar text.
3. **Selector audit** – Add `AutomationId` attributes to controls in a sample page, write tests that locate them by accessibility ID, and verify they remain stable after theme changes.
4. **Cross-platform skip logic** – Introduce `[PlatformFact]`-style attributes that read from `RuntimeInformation` and feature flags (e.g., skip tray icon tests on macOS), then apply them to OS-specific suites.
5. **Wait strategy** – Replace any `Thread.Sleep` in your tests with a reusable wait helper that polls for element state using Appium's `WebDriverWait`, ensuring the helper raises descriptive timeout errors.

What's next - Next: Chapter44

44. Environment setup, drivers, and device clouds

Goal - Stand up reliable Appium infrastructure for Avalonia desktop automation on Windows and macOS. - Package and register test apps so automation servers can launch them locally or on remote device clouds. - Script build/start/stop flows that keep CI agents clean while preserving diagnostics.

Why this matters - Incorrect driver versions or unregistered bundles are the top causes of flaky Appium runs. - Avalonia apps often ship custom arguments (overlay popups, experimental features); tests need a repeatable way to pass them to the harness. - Device-cloud execution magnifies small misconfigurations—locking your setup locally prevents expensive remote reruns.

Prerequisites - Chapter 43 for the fundamentals of Avalonia's Appium test harness. - Chapter 42 for CI orchestration patterns and artifact capture. - Base familiarity with platform build tooling (PowerShell, bash, Xcode command-line tools).

1. Install automation servers and drivers

Windows

1. Install **WinAppDriver** (https://github.com/microsoft/WinAppDriver). It registers itself in the Start menu and listens on `http://127.0.0.1:4723`.
2. Ensure the machine is running in **desktop interactive** mode—WinAppDriver cannot interact with headless Windows Server sessions.
3. Optional: pin the service to auto-start via `schtasks` or a Windows Service wrapper so CI agents bring it up automatically.

macOS

1. Install **Appium** (`npm install -g appium`). For Appium 1, the built-in mac driver is sufficient; for Appium 2 install the `mac2` driver (`appium driver install mac2`).
2. Grant Xcode helper the accessibility permissions required to drive UI (see harness readme at `external/Avalonia/tests/Avalonia.IntegrationTests.Appium/readme.md`).
3. Register your Avalonia app bundle so Appium can launch it by bundle ID. Avalonia's script `samples/IntegrationTestApp/bundle.sh` builds and publishes the bundle.
4. Start Appium. For Appium 2 use a base path to maintain compatibility with existing clients: `appium --base-path=/wd/hub`.

The harness toggles between Appium 1 and 2 using the `IsRunningAppium2` property (`external/Avalonia/tests/Avalonia.I` Set the property to `true` in `Directory.Build.props` or via `dotnet test -p:IsRunningAppium2=true` when running against Appium 2.

2. Package and register the test app

Appium launches desktop apps by path (Windows) or bundle identifier (macOS). The Avalonia sample uses `IntegrationTestApp` and rebuilds it before each run:

- macOS pipeline script (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/macos-clean-build-test` cleans the repo, compiles native dependencies, bundles the app, and opens it once to register Launch Services.
- Windows pipeline (`external/Avalonia/azure-pipelines-integrationtests.yml:42`) builds `IntegrationTestApp` and the test project before running `dotnet test`.

When testing your own app:

1. Provide a CLI or script (PowerShell/bbash) that packs the app and exposes the absolute path or bundle ID through environment variables (`TEST_APP_PATH`, `TEST_APP_BUNDLE`).
2. Inherit from `DefaultAppFixture` and override `ConfigureWin32Options` / `ConfigureMacOptions` to use those values. Example:

```
protected override void ConfigureWin32Options(AppiumOptions options, string? app = null)
{
    base.ConfigureWin32Options(options, Environment.GetEnvironmentVariable("TEST_APP_PATH"));
}

protected override void ConfigureMacOptions(AppiumOptions options, string? app = null)
{
    base.ConfigureMacOptions(options, Environment.GetEnvironmentVariable("TEST_APP_BUNDLE"));
}
```

3. For variants (e.g., overlay popups), add command-line arguments via capabilities. `OverlayPopupsAppFixture` adds `--overlayPopups` on both platforms (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/Overla`

3. Start/stop lifecycle scripts

Automation servers must be running when tests start and shut down afterward. Avalonia's pipelines demonstrate the sequence:

- **macOS job** kills stray processes (`pkill node`, `pkill IntegrationTestApp`), starts Appium in the background, bundles the app, launches it, runs `dotnet test`, then terminates Appium and the app again (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/macos-clean-build-test.sh:6`).
- **Windows job** uses Azure DevOps tasks to start/stop WinAppDriver (`external/Avalonia/azure-pipelines-integra`) When scripting locally, run `Start-Process "WinAppDriver.exe"` before tests and `Stop-Process -Name WinAppDriver` afterward.

General guidelines:

- Always clean up (`pkill`, `Stop-Process`) on both success and failure to keep subsequent runs deterministic.
- Redirect server logs to files (`appium > appium.out &`). Publish them when the job fails for easier triage (see pipeline's `publish appium.out` step).

4. Device cloud configuration

Device clouds (BrowserStack App Automate, Sauce Labs, Azure-hosted desktops) require the same capabilities plus authentication tokens:

```
options.AddAdditionalCapability("browserstack.user", Environment.GetEnvironmentVariable("BS_USER"));
options.AddAdditionalCapability("browserstack.key", Environment.GetEnvironmentVariable("BS_KEY"));
options.AddAdditionalCapability("appium:options", new Dictionary<string, object>
{
    ["osVersion"] = "11",
    ["deviceName"] = "Windows 11",
    ["appium:app"] = "bs://<uploaded-app-id>"
});
```

Upload your Avalonia app (packaged exe zipped, or macOS `.app` bundle) via the vendor's CLI before tests run. On hosted Windows machines, ensure the automation provider exposes UI Automation trees—some locked-down images disable it.

When targeting clouds, keep these adjustments in fixtures:

```
protected override void ConfigureWin32Options(AppiumOptions options, string? app = null)
{
    if (UseCloud)
    {
        options.AddAdditionalCapability("app", CloudAppId);
        options.AddAdditionalCapability("bstack:options", new { osVersion = "11", sessionName = TestCon
```

```
    }
    else
    {
        base.ConfigureWin32Options(options, app);
    }
}
```

Guard cloud-specific behavior using environment variables so local runs stay unchanged.

5. Managing driver compatibility

The harness conditionally compiles for Appium 1 vs. 2 via `APPIUM1`/`APPIUM2` constants (`AppiumDriverEx.cs`). Checklist:

- Run `dotnet test -p:IsRunningAppium2=true` when hitting Appium 2 endpoints. This updates `DefineConstants` and switches to the newer `Appium.WebDriver 5.x` client.
- Ensure the Appium server version matches the driver: Appium 2 + mac2 driver expect W3C protocol only.
- WinAppDriver currently supports only Appium 1, so keep a separate pipeline lane for Windows if you standardize on Appium 2 for macOS.

If you see protocol errors, print the server log (`appium.out`) and compare capability names. Appium 2 requires `appium:` prefixes for vendor-specific entries (already shown in `DefaultAppFixture.ConfigureMacOptions`).

6. Permissions and security prompts

Desktop automation breaks when the app lacks accessibility permissions:

- macOS: add the Appium binary, the terminal/agent, and Xcode helper to `System Settings → Privacy & Security → Accessibility`. The readme covers the exact steps.
- Windows: disable UAC prompts or run the agent as administrator. If UAC prompts appear, automation cannot interact with the foreground until dismissed.
- Device clouds: follow provider docs to grant persistent accessibility or run under pre-approved automation accounts.

Automate these steps where possible—on macOS you can pre-provision a profile or run a script to enable permissions via `tccutil`. For Windows, prefer an image with WinAppDriver pre-installed.

7. Logging and diagnostics

Augment your harness to collect evidence:

- Use `appium --log-level info --log appium.log` to write structured JSON logs.
- Forward driver logs to test output: `Session.Manage().Logs.GetLog("driver");` after a failure.
- For WinAppDriver, enable verbose logs via registry (`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WinAppDriver\Conso = 1`).
- Record video on Windows using the supplied `record-video.runsettings` file when executing through VSTest (Chapter 42).

8. Troubleshooting

- **`SessionNotCreatedException`** – Check that the app path/bundle exists and the process isn't already running. On macOS, run `osascript` cleanup like the sample script to delete stale bundles.
- **`Could not find app`** – Re-run your packaging script; the bundle path changes when switching architectures (`osx-arm64` vs. `osx-x64`).
- **Authentication failures on clouds** – Ensure credentials are injected securely via pipeline secrets; log obfuscated values for debugging but never commit them to source.

- **Driver mismatch** – Align `IsRunningAppium2` with the server version. Appium 2 rejects legacy capability names like `bundleId` without the `appium:` prefix.
- **Resource leaks** – Always dispose fixtures, even in skipped tests. Wrap `Session` accesses in `try/finally` or use `IAsyncLifetime` to guarantee cleanup after each class.

Practice lab

1. **Bootstrap script** – Create cross-platform scripts (`scripts/run-appium-tests.ps1` and `.sh`) that build your app, start/stop automation servers, and invoke `dotnet test`. Validate they leave no background processes.
2. **Configurable fixture** – Extend `DefaultAppFixture` to read capabilities from JSON (local vs. cloud). Add tests that assert the chosen configuration by inspecting `Session.Capabilities`.
3. **Permission audit** – Write a checklist or automated probe that verifies accessibility permissions before starting tests (e.g., attempt to focus a dummy window and fail fast with instructions).
4. **Driver matrix** – Run the same smoke suite against Appium 1 (WinAppDriver) and Appium 2 (mac2) by toggling `IsRunningAppium2`. Capture and compare server logs to understand protocol differences.
5. **CI integration** – Add jobs to your pipeline that call your bootstrap script on Windows and macOS runners. Upload Appium logs and test TRX files as artifacts, confirming cleanup occurs even when tests fail.

What's next - Next: Chapter45

45. Element discovery, selectors, and PageObjects

Goal - Locate Avalonia controls reliably through Appium's accessibility surface, even when templates or virtualization hide elements. - Encapsulate selectors and interactions in reusable PageObjects so suites stay maintainable as the UI grows. - Combine waits, retries, and platform-aware helpers to avoid brittle tests across Windows, macOS, and remote hosts.

Why this matters - Avalonia templates can reshape automation trees; hard-coded XPath falls apart when themes change. - Virtualized lists only materialize visible items—selectors must cope with dynamic children. - Cross-platform automation surfaces expose different attributes; centralizing logic keeps suites portable.

Prerequisites - Chapter 43 for harness fundamentals. - Chapter 44 for environment setup and driver configuration. - Familiarity with Avalonia accessibility APIs (`AutomationProperties`).

1. Build selectors on accessibility IDs first

Avalonia maps `AutomationProperties.AutomationId` and control `Name` directly into Appium selectors. Tests such as `AutomationTests.AutomationId` rely on `FindElementByAccessibilityId` (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/AutomationTests.cs:12`). Adopt this priority order:

1. `FindElementByAccessibilityId` for IDs you own.
2. `FindElementByName` for localized labels (`ElementExtensions.GetName`) or menu items.
3. `FindElementByXPath` as a last resort for structure-dependent lookups (e.g., tray icons on Windows).

Annotate controls in XAML with both `x:Name` and `AutomationProperties.AutomationId` to keep selectors stable. For templated controls, expose IDs through template parts so they enter the automation tree.

2. Reuse PageObject-style wrappers

Avalonia's Appium harness centralizes navigation in `TestBase`. Each test class inherits and passes the page name, letting `TestBase` click through the pager with retries (`external/Avalonia/tests/Avalonia.IntegrationTests.Appi`
Mirror this structure:

```
public abstract class CatalogPage : TestBase
{
    protected CatalogPage(DefaultAppFixture fixture, string pageName)
        : base(fixture, pageName) { }

    protected AppiumWebElement Control(string automationId)
        => Session.FindElementByAccessibilityId(automationId);
}

public sealed class WindowPage : CatalogPage
{
    public WindowPage(DefaultAppFixture fixture) : base(fixture, "Window") { }

    public AppiumWebElement WindowState => Control("CurrentWindowState");
    public void SelectState(string id) => Control(id).SendClick();
}
```

Wrap gestures (click, double-click, modifier shortcuts) in extension methods rather than duplicating `Actions` blocks. Avalonia's `ElementExtensions.SendClick` simulates physical clicks to accommodate controls that resist `element.Click()` (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/ElementExtensions.cs:235`).

3. Handle virtualization and dynamic children

Virtualized lists only generate visible items. `ListBoxTests.Is_Virtualized` counts visual children returned by `GetChildren` to prove virtualization is active (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/ListBo`

Techniques: - Scroll or page through lists via keyboard (`Keys.PageDown`) or pointer wheel to materialize items lazily. - Query container children each time rather than caching stale `AppiumWebElement` references. - Use sentinel elements (e.g., "Loading…" items) to detect asynchronous population and wait before asserting.

```
public IReadOnlyList<AppiumWebElement> VisibleRows()
    => Session.FindElementByAccessibilityId("BasicListBox").GetChildren();
```

Combine with helper waits to poll until a desired item appears instead of assuming immediate materialization.

4. Account for platform differences in selectors

Avalonia ships cross-platform helpers that encapsulate OS-specific attribute quirks:

- `ElementExtensions.GetComboBoxValue` chooses `Text` on Windows and `value` on macOS (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/ElementExtensions.cs:34`).
- `GetCurrentSingleWindow` navigates macOS's duplicated window hierarchy by using a parent XPath (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/ElementExtensions.cs:60`).
- `TrayIconTests` opens nested sessions to access Windows taskbar automation IDs, while macOS uses generic status item XPath (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/TrayIconTests.cs:13`

Keep such logic in dedicated helpers; PageObjects should consume a single API regardless of platform. Provide capabilities (e.g., `UseOverlayPopups`) through fixtures so tests stay declarative (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/OverlayPopupsAppFixture.cs:4`).

5. Synchronize with the UI deliberately

Animations and popups require waits. The harness uses:

- Retries in `TestBase` navigation with `Thread.Sleep(1000)` between attempts to allow fullscreen transitions (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/TestBase.cs:12`).
- Looped polling in `ElementExtensions.OpenWindowWithClick` to detect new window handles or child windows, retrying up to ten times (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/ElementExtens`
- Explicit sleeps after context menu or tray interactions when platform APIs lag (`external/Avalonia/tests/Avalonia.I`

Upgrade these patterns using `WebDriverWait` to poll until predicates succeed:

```
public static AppiumWebElement WaitForElement(AppiumDriver session, By by, TimeSpan timeout)
{
    return new WebDriverWait(session, timeout).Until(driver =>
    {
        var element = driver.FindElement(by);
        return element.Displayed ? element : null;
    });
}
```

Centralize waits so adjustments (timeouts, polling intervals) propagate across the suite.

6. Model complex selectors as queries

Large UIs often require multi-step discovery:

- Menus: `MenuTests` clicks through root, child, and grandchild items using accessibility IDs and names (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/MenuTests.cs:25`). Wrap this into helper methods like `OpenMenu("Root", "Child", "Grandchild")`.

- Tray icons: `GetTrayIconButton` first attempts to find the icon, then expands the overflow flyout if absent, handling whitespace quirks in names (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/TrayIc`
- Windows: `OpenWindowWithClick` tracks new handles or titles, accommodating macOS fullscreen behavior by ignoring untitled intermediate nodes (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/Elemer`

Treat these as queries, not static selectors. Accept parameters (icon name, menu path) and apply consistent error messaging when assertions fail.

7. Use test attributes to scope runs

Selectors often depend on platform capabilities. Decorate tests with `[PlatformFact]` / `[PlatformTheory]` to skip unsupported scenarios (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/PlatformFactAttribute`
This prevents PageObjects from needing conditionals inside every method and ensures pipelines stay green when features diverge.

Group tests requiring special fixtures (e.g., overlay popups) via xUnit collections (`external/Avalonia/tests/Avalonia.Inte`
PageObjects then request the appropriate fixture type through constructor injection.

8. Troubleshooting selectors

- **Elements disappear mid-test** – virtualization recycled them; retrieve fresh references after scrolling.
- **Click no-ops** – switch to `SendClick` actions; some controls ignore `element.Click()` on macOS.
- **Wrong element chosen** – qualify by automation ID before falling back to names. Names may change with localization.
- **Popups not found** – ensure you expanded parent menus or overflow trays first. Add logging describing the hierarchy you traversed for easier debugging.
- **Timeouts** – adopt structured waits instead of arbitrary sleeps; log the search strategy (selector type, fallback attempts) on failure.

Practice lab

1. **PageObject refactor** – Extract a PageObject for a complex page (e.g., ComboBox) that exposes strongly-typed actions and returns typed results (`GetSelectedValue`). Replace direct selector usage in tests.
2. **Selector fallback** – Implement a helper that tries `AutomationId`, then `Name`, then a custom XPath, logging each attempt. Use it to locate menu items with localized labels.
3. **Virtualized scrolling** – Write a test that scrolls through a long `ListBox`, verifying virtualization by checking `GetChildren().Count` stays below a threshold while confirming a distant item becomes Selected.
4. **Wait utility** – Replace `Thread.Sleep` in one test with a reusable `WaitFor` method leveraging `WebDriverWait`. Confirm the test still passes under slower animations by injecting artificial delays.
5. **Cross-platform assertions** – Add assertions that rely on windows or tray icons, guarding them with `[PlatformFact]`. Implement helper methods that throw informative exceptions when run on unsupported platforms.

What's next - Next: Chapter46

46. Cross-platform scenarios and advanced gestures

Goal - Exercise Avalonia apps under platform-specific shells—window chrome, tray icons, menus—without duplicating logic per OS. - Drive complex pointer and keyboard gestures (drag, multi-click, context tap) using Appium actions that map correctly on Windows and macOS. - Validate multi-monitor layouts, fullscreen transitions, and system integrations while keeping selectors and waits resilient.

Why this matters - Desktop affordances behave differently across Win32 and macOS accessibility stacks; tests must adapt or risk false negatives. - Advanced gestures rely on low-level pointer semantics that Appium exposes inconsistently across drivers. - Cross-platform consistency is a core Avalonia selling point—automated verification keeps regressions from sneaking in.

Prerequisites - Chapter 43 for the foundational Appium harness. - Chapter 45 for selector patterns and PageObject design. - Familiarity with Avalonia windowing APIs (Chapters 12 and 18).

1. Split platform-specific coverage with fixtures and attributes

`PlatformFactAttribute` and `PlatformTheoryAttribute` skip tests on unsupported OSes (`external/Avalonia/tests/Aval`
Use them to branch behavior cleanly:

```
[PlatformFact(TestPlatforms.MacOS)]
public void ThickTitleBar_Drag_Reports_Moves() { ... }
```

Group tests into collections bound to fixtures that configure capabilities. For example, `DefaultAppFixture` launches the stock ControlCatalog, while `OverlayPopupsAppFixture` adds `--overlayPopups` arguments to highlight overlay behavior (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/OverlayPopupsAppFixture.`

2. Window management across platforms

Windows

`WindowTests` (see `external/Avalonia/tests/Avalonia.IntegrationTests.Appium/WindowTests.cs`) verifies state transitions (Normal, Maximized, FullScreen), docked windows, and mode toggles. It uses `SendClick` on combo entries because native `Click()` is unreliable on certain automation peers (`ElementExtensions.SendClick`, `external/Avalonia/tests/Avalonia.IntegrationTests.Appium/ElementExtensions.`

macOS

`WindowTests_MacOS` covers thick title bars, system chrome toggles, and fullscreen animations (`external/Avalonia/tests/Ava`
Tests depend on applying window decoration parameters via checkboxes exposed in the demo app.

Tips - Normalize state by calling the same helper at test end; `PointerTests_MacOS.Dispose` resets window parameters before exiting (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/PointerTests_MacOS.cs:11`
- When switching states that trigger animations, add intentional waits or `WebDriverWait` polling before grabbing the next snapshot.

3. Multi-window flows and dialogs

Use `ElementExtensions.OpenWindowWithClick` to encapsulate the logic of detecting new windows. It differentiates between top-level handles (Windows) and child windows (macOS) and returns an `IDisposable` that closes the window on teardown (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/ElementExtensions.cs`

```
using (Control("OpenModal").OpenWindowWithClick())
{
    // Assert modal state
}
```

`PointerTests.Pointer_Capture_Is_Released_When_Showing_Dialog` relies on this helper to ensure capture is cleared when a dialog opens (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/PointerTests.cs:1`

4. Tray icons and system menus

System integration differs dramatically:

- **Windows**: `TrayIconTests` locates the shell tray window, handles overflow flyouts, and accounts for whitespace-prefixed icon names (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/TrayIconTests.c`). It also opens a secondary "Root" session that targets the desktop to access the taskbar.
- **macOS**: tray icons appear as `XCUIElementTypeStatusItem` elements and menus are retrieved via `//XCUIElementTypeStatusItem/XCUIElementTypeMenu`.

Wrap this logic in helper methods and hide it behind PageObjects so tests merely call `TrayIcon().ShowMenu()` and assert resulting automation flags.

5. Advanced pointer gestures

Gesture taxonomy

`GestureTests` demonstrates how to script taps, double-taps, drags, and right-clicks using Actions API (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/GestureTests.cs:16`). Examples:

- `new Actions(Session).DoubleClick(element).Perform();`
- Multi-step pointer sequences using `PointerInputDevice` for macOS-specific right-tap semantics (`GestureTests.RightTapped_Is_Raised_2`, line 139).

Title bar drags on macOS

`PointerTests_MacOS.OSXThickTitleBar_Pointer_Events_Continue_Outside_Window_During_Drag` verifies pointer capture beyond window bounds while dragging the title bar (`external/Avalonia/tests/Avalonia.Integration`). It uses `DragAndDropToOffset` and reads automation counters from the secondary window.

Practice - Always move the pointer onto the target before pressing: `new Actions(Session).MoveToElement(titleAreaCont` - After custom pointer sequences, release buttons even when assertions fail to leave the driver in a consistent state (`GestureTests.DoubleTapped_Is_Raised_2`, line 70).

6. Keyboard modifiers and selection semantics

`ListBoxTests` executes Shift-range selection and marks Ctrl-click tests as skipped due to driver limitations (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/ListBoxTests.cs:36`). Document such constraints in your suite and apply `[Fact(Skip=...)]` with explanations for future debugging.

`ComboBoxTests` rely on keyboard shortcuts (`Keys.LeftAlt + Keys.ArrowDown`) and ensure wrapping behavior toggles via checkboxes before assertion (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/ComboBoxTes`). Keep these interactions in PageObjects so tests remain expressive (`ComboBoxPage.OpenDropdown()` vs. inline key sequences).

7. Multi-monitor and screen awareness

`ScreenTests` pulls current monitor data and asserts invariants around bounds, work area, and scaling (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/ScreenTests.cs:12`). Use similar verifications when you need to assert window placement on multi-monitor setups.

For drag-to-monitor flows, record starting and ending positions via text fields surfaced in the app, then compare after applying pointer moves. Ensure tests reset state (move window back) when done to avoid cascading failures.

8. Troubleshooting cross-platform gestures

- **Stuck pointer buttons** – Ensure `PointerInputDevice` sequences end with `PointerUp`. If a test fails mid-action, add `try/finally` to release buttons.

- **Unexpected double-taps** – As shown in `PointerTests_MacOS.OSXThickTitleBar_Single_Click_Does_Not_Genera` add counters to your app to observe actual events and assert on them instead of stateful UI side effects.
- **Tray icon discovery failures** – Expand overflow menus explicitly on Windows; on macOS, allow for menu creation delays by polling after clicking the status item.
- **Localization differences** – Names of system menu items vary; rely on automation IDs when possible or provide fallback selectors.
- **Driver limitations** – Document known issues (e.g., WinAppDriver ctrl-click) with skip reasons so team members know why coverage is missing.

Practice lab

1. **Window choreography** – Script a test that opens a secondary window, drags it to a new position, toggles fullscreen, and returns to normal. Assert pointer capture counts using automation counters exposed in the sample.
2. **Tray icon helper** – Build a PageObject with `ShowMenu()` and `ClickMenuItem(string text)` methods that handle Windows overflow and macOS status items automatically. Use it to verify a menu command toggles a checkbox in the main window.
3. **Gesture pipeline** – Implement a helper that performs a parameterized pointer gesture (`PointerSequence` builder). Use it to test tap, double-tap, drag, and right-tap on the same control, asserting the logged gesture text each time.
4. **Multi-monitor regression** – Extend the sample app to surface target screen IDs. Write a test that moves a window across monitors and verifies the reported screen changes, resetting to the primary display afterward.
5. **Platform matrix** – Create a theory that runs the same smoke scenario across Windows/Mac fixtures using `[PlatformTheory]`. Capture driver logs on failure and assert the test records which platform executed for easier triage.

What's next - Next: Chapter47

47. Stabilizing suites, reporting, and best practices

Goal - Keep Appium-based Avalonia suites reliable on developer machines and CI by isolating flakiness causes. - Capture meaningful diagnostics (logs, videos, artifacts) that accelerate investigation when tests fail. - Scale coverage with retry, quarantine, and reporting strategies that protect signal quality.

Why this matters - Cross-platform automation is sensitive to timing, focus, and OS updates—without discipline the suite becomes noisy. - Fast feedback requires structured artifacts; otherwise failures devolve into manual repro marathons. - Stakeholders need trend visibility: which areas flake, which platforms lag, and where to invest engineering effort.

Prerequisites - Chapter 43–46 for harness setup, selectors, and advanced scenarios. - Chapter 42 for CI pipeline integration basics.

1. Triage flakiness with classification

Begin every investigation by tagging failures: - **Timing** (animations, virtualization) – resolved with better waits (`WebDriverWait`, dispatcher polling). - **Environment** (permissions, display scaling) – addressed by setup scripts or platform skips. - **Driver quirks** (WinAppDriver Ctrl-click) – documented with `[Fact(Skip="...")]` like `ListBoxTests.Can_Select_Items_By_Ctrl_Clicking` (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/ListBoxTests.cs:36`). - **App bugs** – file issues with automation evidence attached.

Maintain a living flake log referencing test name, platform, root cause, and remediation. Automate updates by pushing annotations into test reporters (Azure Pipelines, GitHub Actions).

2. Quarantine and retries without hiding real bugs

Retries buy time but can mask regressions. Strategies:

- Implement targeted retries via xUnit ordering or `[RetryFact]` equivalents. Avalonia currently handles retries manually by skipping unstable tests with reason strings (e.g., `TrayIconTests.Should_Handle_Left_Click` is marked `[PlatformFact(..., Skip = "Flaky test")]`, `external/Avalonia/tests/Avalonia.IntegrationTests`
- Prefer **automatic quarantine**: tag flaky tests and run them in a separate lane, keeping main suites failure-free. Example: use xUnit traits or custom attributes to filter (`dotnet test --filter "TestCategory!=Quarantine"`).
- Combine retries with diagnostics: on the last retry failure, dump Appium logs and take screenshots before failing.

3. Capture rich diagnostics

For every critical failure, collect:

- **Appium server logs** (`appium.out` in the macOS script) and publish them via CI artifacts (`external/Avalonia/azure-pipelines-integrationtests.yml:27`).
- **Driver logs**: `Session.Manage().Logs.GetLog("driver")` after catch blocks to capture protocol exchanges.
- **Screenshots**: call `Session.GetScreenshot().SaveAsFile(...)` on failure; stash path in test output.
- **Videos**: on Windows, VSTest runsettings `record-video.runsettings` records screen output (`external/Avalonia/tests/Avalonia.IntegrationTests.Appium/record-video.runsettings`).
- **Headless imagery**: pair Appium runs with headless captures (Chapter 40) to highlight visual state at failure.

Build helper methods so tests simply call `ArtifactCollector.Capture(context);`. Ensure cleanup occurs even when assertions throw (use `try/finally`).

4. Standardize waiting and polling policies

Enforce consistent defaults:

- Set a global implicit wait (short, e.g., 1s) and rely on explicit waits for complex states. Too-long implicit waits slow down failure discovery.
- Provide `WaitForElement` and `WaitForCondition` helpers with logging. Use them instead of ad-hoc `Thread.Sleep`.
- For dispatcher-driven state, expose instrumentation in the app (text fields reporting counters like `GetMoveCount` in `PointerTests_MacOS`, `external/Avalonia/tests/Avalonia.IntegrationTests.Appium/PointerT` Poll those values to assert behavior deterministically.

Document wait policies in CONTRIBUTING guidelines to onboard new contributors.

5. Structure reports for quick scanning

Azure Pipelines / GitHub Actions

- Publish TRX results with names that encode platform, driver, and suite (e.g., `Appium-macOS-Appium2.trx`).
- Upload log bundles (`logs/appium.log`, `screenshots/*.png`). Provide clickable links in summary markdown.
- Add summary steps that print failing test names grouped by category (flaky, new regression, quarantined).

Local development

- Provide a script (Chapter 44) that mirrors CI output directories so developers can inspect logs locally.
- Encourage use of `dotnet test --logger "trx;LogFileName=local.trx"` + `reportgenerator` for HTML summaries.

6. Enforce coding standards in tests

- **Selectors**: centralize in PageObjects. No raw XPath in tests.
- **Waits**: ban `Thread.Sleep` in code review; insist on helper usage.
- **Cleanup**: always dispose windows/sessions (`using` pattern with `OpenWindowWithClick`). Review tests that skip cleanup (they often cause downstream failures).
- **Platform gating**: pair every platform-specific assertion with `[PlatformFact]`/`[PlatformTheory]` to avoid accidental runs on unsupported OSes.

Add lint tooling (Roslyn analyzers or custom scripts) to scan for banned patterns (e.g., `Thread.Sleep()` in test projects.

7. Monitor and alert on trends

- Track success rate per platform, per suite. Configure dashboards (Azure Analytics, GitHub Insights) to display pass percentages over time.
- Emit custom metrics (e.g., number of retries) to a time-series store. If retries spike, alert engineers before builds start failing.
- Rotate flake triage duty; publish weekly summaries identifying top offenders and assigned owners.

8. Troubleshooting checklist

- **Frequent timeouts** – confirm Appium server stability, check CPU usage on agents, review wait durations.
- **Intermittent focus issues** – ensure tests foreground windows (`SetForegroundWindow` on Windows) or click background-free zones before interacting.

- **Driver crashes** – update Appium/WinAppDriver, capture crash dumps, and reference known issues (e.g., mac2 driver close-session crash handled in `DefaultAppFixture.Dispose`).
- **Artifacts missing** – verify CI scripts always run artifact upload steps with `condition: always()`.
- **Quarantine drift** – periodic reviews to reinstate fixed tests; failing to do so erodes coverage.

Practice lab

1. **Artifact collector** – Implement a helper that captures Appium logs, driver logs, screenshots, and optional videos when a test fails. Wire it into an xUnit `IAsyncLifetime` fixture so it runs automatically.
2. **Wait audit** – Write an analyzer or script that flags `Thread.Sleep` usages in the Appium test project. Replace them with explicit waits and document the change.
3. **Quarantine lane** – Configure your CI pipeline with two jobs: stable and quarantine (`dotnet test --filter "Category!=Quarantine"` vs. `Category=Quarantine`). Move a flaky test into the quarantine lane and verify reporting highlights it separately.
4. **Trend dashboard** – Export TRX results for the past week and build a simple dashboard (Power BI, Grafana) showing pass/fail counts per platform. Identify top flaky tests.
5. **Regression template** – Create an issue template that captures test name, platform, driver version, app commit, and links to artifacts. Use it when logging Appium regressions to standardize triage information.

What's next - Return to Index for appendices, publishing checklists, or future updates.