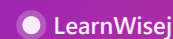


Theming, Resources, Localization & UI Modernization

Modernize Wisej.NET applications using themes, CSSStyle, resources, icons, runtime theme switching, localization, and visual consistency rules.



Lesson goal

Modernize Wisej.NET applications using themes, CSSStyle, resources, icons, runtime theme switching, localization, and visual consistency rules — separating *look* (theme/CSS) from *behavior* (C#) so a redesign or a new language never means editing every screen.

Everything in this module rests on one discipline: **appearance and logic are different jobs and belong in different places**. Your C# decides *what the app does*; a theme decides *what the app looks like*; resources hold the *strings and images* the app shows. When those three stay separate, you can restyle the product, add a dark mode, or ship an Italian build without touching a single event handler. When they bleed together — a hex color typed into a control here, a caption hard-coded there — every change becomes a hunt across the whole codebase. This lesson is about keeping them apart on purpose.

1. Separating appearance from behavior

In Wisej.NET, a screen's *look* is data, not code. A **theme** is a JSON document that defines colors, fonts, spacing, borders and the appearance of every control type — buttons, grids, panels, tabs. Your C# never mentions `#0d6efd` or `font-size: 14px`; it just creates a `Button`, and the active theme decides how that button is painted. Change the theme and the whole application re-skins itself, because the controls were never carrying their own colors in the first place.

Layer	What it owns	Lives in
Theme (JSON)	Global look — the color palette, fonts, spacing scale, and the default appearance of every control type.	A <code>.theme</code> / theme JSON file, or Theme Builder.
CSSStyle	A targeted visual override applied to one control or a small family of controls.	The control's <code>CssStyle</code> property (C# or Designer).
Behavior (C#)	What the app <i>does</i> — validation, workflows, data, navigation.	Your event handlers and services.
Resources	The strings and images the app shows — captions, messages, icons, logos.	Resource files / the Resources folder.

The payoff of centralizing colors and spacing in a theme instead of styling each control is leverage. Suppose your brand color changes from blue to teal. With a theme, you edit one value and every button, link and highlight updates together. With per-control styling, that same change is a search-and-replace

across dozens of files — and you *will* miss one. Centralization also enforces consistency you didn't have to think about: because every control reads its spacing from the same scale, screens line up without anyone eyeballing pixels.



Plain-English rule

If a visual value describes your *whole app* (the brand blue, the standard gutter), it belongs in the theme. If it describes *one unusual control*, it belongs in that control's `CSSStyle`. It should almost never live as a literal in `C#`.

2. Themes and Theme Builder

Wisej.NET ships with a set of built-in themes and provides **Theme Builder**, a visual editor for creating or tweaking them. You start from a base theme — for example a light default or a dark variant like **BootstrapDark-4** — and adjust the palette, typography and control appearances, then save the result as your application's theme. Because Theme Builder edits the same JSON the runtime reads, there is no separate "export" step to get wrong: what you design is what the app loads.

A few concepts worth naming. A theme defines **named colors** (a palette) so appearances reference `"highlight"` or `"error"` rather than raw hex, meaning one palette edit ripples everywhere.

Appearances are the per-control-type style definitions. And **mixins** let you layer a small set of overrides on top of a base theme instead of forking the whole thing — the right tool when you want "the standard theme, but with our brand color and logo."

3. Switching theme at runtime

Because the theme is just the active look, you can change it while the app is running — for a per-user dark-mode toggle, a high-contrast accessibility mode, or a tenant-specific skin. You set the application's active theme, and Wisej.NET re-renders the open screens against the new appearances. You do not rebuild controls or reload data; only the paint changes.

A dark-mode toggle that swaps the active theme at runtime

```
private void btnDarkMode_Click(object sender, EventArgs e)
{
    // Pick the theme by name; the runtime re-skins open screens.
    // Behavior does not change – only the active look does.
    var next = _darkMode ? "Default-4" : "BootstrapDark-4";
    Application.Theme = next;           // switch the whole app's look
    _darkMode = !_darkMode;

    // Persist the choice so it survives the next login.
    _userPrefs.Save("theme", next);
}
```

Choosing the theme at startup, before the first screen paints

```
protected override void OnSessionStart(EventArgs e)
{
    // Read the saved preference and set the look once, up front,
    // so the user never sees a flash of the wrong theme.
    var saved = _userPrefs.Load("theme") ?? "Default-4";
    Application.Theme = saved;
    base.OnSessionStart(e);
}
```

Notice what the handler does *not* do: no color literals, no per-control restyling, no reloading of data. The look is named data; switching it is a one-line change. (The exact property/API for setting the active theme may differ by Wisej.NET version — the mechanism is the same: assign the application's active theme by name and the runtime re-skins the UI.)

4. CSSStyle — the targeted exception

Sometimes one control genuinely needs to look different from its siblings in a way the theme shouldn't encode globally — a warning banner with a distinct background, a card with a soft shadow. That is what the control's **CSSStyle** property is for: it attaches CSS to that specific widget without polluting the theme or reaching into the browser DOM yourself.

A CSSStyle override on a single status panel

```
pnlBanner.CSSStyle =
    "background: var(--wisej-warning, #fff3cd);" + // prefer a theme var over a raw hex
    "border-radius: 8px;" +
    "padding: 12px 16px;";

lblBanner.Text = "Your session expires in 5 minutes.";
```

Use CSSStyle sparingly and deliberately. Every CSSStyle string is a small piece of appearance that lives *outside* the theme, so it won't follow a redesign automatically. The habit to build: reach for the theme first, and only drop to CSSStyle when a control is a true one-off. If you find yourself pasting the same CSSStyle onto many controls, that is a signal it belongs in the theme instead.

Theme vs. CSSStyle vs. inline property — when to use each

Tool	Scope	Use it when...
Theme (JSON / Theme Builder)	The whole application, all controls of a type.	The value is part of your product's identity — palette, fonts, spacing scale, default control look. First choice for anything shared.
CSSStyle property	One control, or a small deliberate family.	A genuine visual exception the theme shouldn't carry — a special banner, a highlighted card.
Inline property (BackColor, Font...)	One control, one property.	Rarely — a truly local, data-driven tweak (e.g. a row turns red because a value is overdue). Never for brand-wide colors.

5. Resources — images, icons and strings

The other half of "separate look from logic" is treating the *content* the app shows — its captions, messages, logos and icons — as **resources** rather than literals scattered through code. Images and icons go in the Resources folder and are referenced by name, so a designer can swap the logo without a recompile of your handlers. User-visible strings go into resource files keyed by name, which is exactly what makes localization possible: the same key resolves to different text per culture.

Loading a localized message and an icon from resources instead of hard-coding them

```
// String comes from a resource keyed by name – not a literal.
// At runtime it resolves to the entry for the current culture.
lblWelcome.Text = Resources.Strings.Welcome;           // "Welcome" / "Benvenuto" / "Willkommen"

// Image comes from a named resource, not an ad-hoc file path.
btnSave.ImageSource = "resource.wx/App.Resources.SaveIcon.png";

// A whole Form/Page can be marked Localizable so the Designer
// stores its captions per culture automatically.
```

The general shape is always the same: a stable **key** in code, resolved to real content at runtime based on the active culture. That indirection is the whole trick — code references keys, and the meaning of a key can change per language, per brand, or per season without the code knowing.

6. Localization — culture, strings and formatting

Localization is more than translation. Two things vary by **culture**: the *strings* you show, and the *formatting* of dates, numbers and currency. Wisej.NET localizes user-visible text by resolving resource keys for the current culture (Forms and Pages can be marked **Localizable** so the Designer manages their captions per language). Formatting follows the thread's culture, so a date renders **7/3/2026** for one user and **03/07/2026** for another — from the *same* C#, because you formatted with the culture rather than a fixed pattern.

Setting the user's culture, then letting formatting follow it

```
// Set the culture for this user's session (from a preference or a picker).
var culture = new CultureInfo(_userPrefs.Load("culture") ?? "en-US");
System.Threading.Thread.CurrentThread.CurrentUICulture = culture; // which strings
System.Threading.Thread.CurrentThread.CurrentCulture = culture; // how dates/numbers
format

// Now the SAME code formats correctly for every user:
lblTotal.Text = amount.ToString("C");           // $1,234.50 vs 1.234,50 €
lblDue.Text = dueDate.ToShortDateString(); // culture-aware, no hard-coded pattern
```

A practical warning: translated text changes length. German labels are often longer than English; some scripts are taller. A layout that fits perfectly in English can clip or wrap badly in another language, so **test localization in real screen layouts**, not just in a spreadsheet of strings. This is why the module treats localization as a UI concern, not a text concern.



Production habit

Theme first, exceptions second. Keep every user-visible string and image as a resource, set the culture once per session, and format with the culture rather than a fixed pattern — then verify the result in the actual screen for each language you ship.



Learning objectives

By the end of this module you'll be able to:

- ✓ Apply built-in themes and switch themes at runtime or startup without changing behavior.
- ✓ Use Theme Builder and CSSStyle without scattering one-off styling everywhere.
- ✓ Centralize colors, spacing, strings and images so a redesign edits one place, not every screen.
- ✓ Localize Forms, Pages, system resources, and data formatting where appropriate.
- ✓ Create a UI modernization checklist for enterprise screens.

7. A modern UI, briefly

Modernizing is not only theming. A modern enterprise screen uses a consistent **spacing scale**, a clear **typographic hierarchy**, accessible **contrast**, recognizable **icons**, and predictable **status and error patterns** so users learn one visual language and reuse it everywhere. The reason all of this belongs in the theme is the same as before: consistency you get for free is consistency you can't accidentally break, and a redesign becomes a data edit instead of a project.

Common mistakes

Mistake	Why it hurts	Do instead
Hard-coded colors and spacing scattered across screens.	A brand change becomes a search-and-replace across dozens of files — and you'll miss one.	Define the palette and spacing in the theme; reference named values, not hex.
Pasting the same CSSStyle onto many controls.	The style lives outside the theme, so it won't follow a redesign and drifts out of sync.	Promote the repeated style into the theme's appearance for that control type.
User-visible strings written as literals in C#.	They can't be translated and a copy edit means recompiling code.	Keep strings in resources keyed by name; resolve per culture.
Images embedded ad-hoc from file paths.	Assets scatter, break on deploy, and can't be swapped without a rebuild.	Add them to Resources and reference by name.
Ignoring culture — fixed date/number patterns.	Dates, decimals and currency render wrong for other regions.	Set the session culture and format with it (<code>Tostring("C")</code> , short-date).
Testing translations only as a string list.	Longer text clips or wraps; layouts break in the real screen.	Test each language in the actual layout before shipping.

Key vocabulary

- **Theme** — the JSON document defining the whole app's look: palette, fonts, spacing and per-control appearances.
- **Theme Builder** — the visual editor for creating and tweaking themes.
- **Appearance** — a theme's style definition for one control type.
- **Mixin** — a small overlay of overrides layered on top of a base theme instead of forking it.
- **CSSStyle** — a control's property for a targeted CSS override on that specific widget.
- **Resource** — a named string or image the app references instead of a literal, enabling swap and translation.
- **Localizable property** — a Form/Page (or control) marked so the Designer stores its captions per culture.
- **Culture** — the setting that drives both which strings show and how dates, numbers and currency format.
- **Runtime theme switch** — changing the active theme while the app runs, re-skinning open screens without rebuilding them.

Common pitfall to avoid



Common pitfall

Scattering hard-coded colors, one-off CSS and literal strings across many screens. It works in a demo and rots in production — a restyle or a new language then means editing every file instead of one theme and one resource set. This is exactly the smell you will remove in this module's lab.

Designer & code review checklist

- ✓ The screen remains usable in the Visual Studio / Wisej.NET Designer.
- ✓ Controls are named clearly enough for a teammate to follow the event code.
- ✓ Colors and spacing come from the theme, not literals in C# or repeated CSSStyle.
- ✓ User-visible strings and images are resources, not hard-coded values.
- ✓ Culture is set per session and formatting follows it.
- ✓ Business logic is not trapped in visual event handlers.
- ✓ Failure paths are visible, logged, and explained without leaking internals.
- ✓ The deliverable can be reviewed without running the whole course.



Self-check before the lab

Which styles are theme-level and which are genuine screen-level exceptions? Can you switch the whole app's look in one line without touching a handler? Is every user-visible string and image a resource? What changes in German or Italian text, and does the layout survive it? Can a future redesign avoid editing every screen? If yes to all, you are ready.

Companion code

`ThemingLocalizationPatterns.cs` collects the patterns from this module in one compilable file. Download it and read it alongside the lab.

[↓ Download ThemingLocalizationPatterns.cs](#)