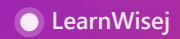


Mobile-Ready Capstone, Accessibility, Performance, and Production Governance

Lesson Guide · Lab / Exam Guide



Lesson Guide

Finishing the Adaptive Operations Console as a production-ready desktop, tablet and phone screen set, reviewing focus, contrast, invalid states and touch targets, keeping profile handlers cheap and idempotent, and governing the theme, CSS and ClientProfiles.json like code. What this means for a Wisej.NET developer — and how to read this module.



Production focus

Finishing the Adaptive Operations Console as a production-ready desktop, tablet and phone screen set, reviewing focus, contrast, invalid states and touch targets, keeping profile handlers cheap and idempotent, and governing the theme, CSS and ClientProfiles.json like code.

1. The production problem

Six modules of work now live in one solution: a theme derived from a base theme, scoped CSS, a nested container shell, Flow, Table and Flex regions, a ClientProfiles.json and a profile handler. What turns that into a product is not another feature. It is proof that the console is usable on every profile, that a keyboard or touch user can operate it, that a resize does not trigger expensive rebuilds, and that the next developer, or the next Wisej.NET upgrade, cannot silently break the visual system. This module is that proof, plus the grounding pack that lets an AI assistant answer questions about the decisions you made.

2. Key concepts

Concept	Why it matters
capstone screen set	The finished Adaptive Operations Console: custom theme, semantic appearances, scoped CSS, a desktop shell, an adaptive tablet shell, a phone shell, a grid, an editor, validation feedback, navigation and a profile indicator visible in test mode.
visual accessibility review	A pass over focus frames, disabled and invalid states, contrast, touch spacing, tooltips on icon-only buttons and keyboard tab order, done by operating the screen rather than reading the theme.
idempotent profile handler	A <code>ResponsiveProfileChanged</code> handler that assigns final values for the current profile, so running it twice, or after a refresh, produces exactly the same UI.
layout stability	<code>MinimumSize</code> and <code>MaximumSize</code> keep regions from collapsing, and <code>AutoSize</code> stays off large container hierarchies, so resizes are predictable and cheap.
theme governance	The theme JSON, the scoped stylesheet and ClientProfiles.json are source-controlled, reviewed as diffs, documented as tokens and never edited inside a built-in theme.
upgrade-safe customization	Documented appearances, Theme Builder workflows and server-side properties survive upgrades; CSS selectors that depend on a widget's internal DOM may not, so they are scoped, documented or avoided.
responsive QA matrix and grounding pack	The matrix records widths, orientations, expected profile and checks per scenario; the grounding pack records the decision tree, answer rules and source map an assistant answers from.

3. Lesson narrative

One console, three shells, one visual system

The capstone is judged on consistency, not novelty. Standard controls take their identity from the AdaptiveOps theme, which derives from a base theme and defines named colour tokens for brand, surfaces, text, danger, warning, success and the focus frame. Business variants are semantic appearances assigned through `AppearanceKey` : `action-button` , `destructive-button` , `kpi-card` , `nav-item` , `compact-editor` or equivalents, at least four of them. Application-specific content such as metric cards uses scoped classes through `CssClass` , and `CssStyle` appears only where a value is computed at runtime. If a screen reaches for `BackColor` , `ForeColor` or `Font` directly, the review asks which of those layers should have owned it.

The shell keeps the structure you built in Modules 4 and 5: docked toolbar, navigation rail, workspace, details region and status footer, with a `FlowLayoutPanel` for filters and cards, a `TableLayoutPanel` for the aligned ticket editor and a `FlexLayoutPanel` for the proportional workspace. Desktop, tablet and phone differences come from ClientProfiles.json plus designer-assigned responsive properties; the one `ResponsiveProfileChanged` handler exists for behaviour a property cannot express, such as opening the details editor as a modal form on phone. The profile indicator stays in the status region, visible in test mode, so every screenshot in the QA matrix names the profile that produced it.

Accessibility you can verify by using the screen

A theme must preserve visible focus, clear disabled states, readable text, sufficient contrast, invalid-state feedback and touch-friendly spacing. The most common regression is deleting the focus frame because it looks old; the rule is to replace it with a better one drawn from the `focusFrame` token, never to remove it. Invalid and warning states must not rely on colour alone: pair the danger border with an icon or a message the user can read, so the state survives colour-blindness and a projector. Contrast is checked against the actual token pairs the theme uses, for example `textMuted` on `surfaceAlt`, not against the brand colour in isolation.

Icon-only toolbar buttons, which the tablet profile switches to through the responsive `Display` value, need `ToolTipText` or an accessible label to stay discoverable. Tab order is verified by walking the screen with the keyboard and confirming `TabStop` and `TabIndex` put the primary command where a user expects it. Finally, hidden content needs a story: when the navigation rail and details panel are hidden on phone, the functions they carried must still be reachable, through the mobile menu button and the modal editor, and the architecture note records that alternative for every hidden control.

Cheap handlers, stable layouts, governed files

Responsive design becomes expensive when every resize causes complex layout, data reloads and client updates. The handler that reacts to a profile change must be idempotent and cheap: it assigns the final visibility, docking and size for the current profile instead of toggling, it does not rebuild the UI when a few property changes are enough, and it never reloads data just because the width changed. The lab's `ApplyProfile` method is called once from the constructor with `Application.ActiveProfile` and again from the handler with the event's current profile, and the subscription is removed when the page is disposed:

```
private void ApplyProfile(string name)
{
    if (name == _lastProfile) return;           // same profile: no work
    _lastProfile = name;
    bool phone = name != null && name.StartsWith("Phone");

    lblProfile.Text = name ?? "Default";         // test-mode indicator
    navRail.Visible = !phone;                   // final values, never toggles
    detailsPanel.Visible = !phone;
    btnOpenDetails.Visible = phone;
    ticketGrid.MinimumSize = new Size(phone ? 0 : 320, 240);
}
```

Layout stability comes from the same discipline: `MinimumSize` and `MaximumSize` on fill-weighted and docked regions so nothing collapses to zero, no `AutoSize` on large container hierarchies, and responsive data changes that are intentional rather than a side effect of a resize. Then treat the theme like code. `AdaptiveOps.theme`, `AdaptiveOps.css` and `ClientProfiles.json` live in source control with clear names; their diffs are reviewed; the design tokens are documented; built-in themes are never edited directly. The team writes down when a developer may use `BackColor`, `ForeColor` or `Font`, when `AppearanceKey`, `CssClass`, `CssStyle` or custom CSS is the right tool, and keeps a screenshot set of the major screens at desktop, tablet and phone sizes.

Upgrade safety follows from the same rules. Documented theme appearances, Theme Builder workflows and server-side properties are stable across Wisej.NET versions; a CSS selector that depends on a widget's internal DOM is not. When a third-party or custom widget genuinely needs direct CSS, scope the selector tightly and document why the theme system does not own that piece. After every upgrade, run the responsive screenshot matrix again and inspect each custom appearance key and widget-specific state.



Common pitfalls

Removing focus frames instead of restyling them. Signalling invalid state with colour only. A profile handler that toggles state or rebuilds the whole page on every change. Fill-weighted regions without a `MinimumSize`. Editing a built-in theme in place. Global CSS patched onto standard controls through internal DOM selectors. A screenshot set that exists once and is never rerun after an upgrade.

A grounding pack an assistant can answer from

The last deliverable is written for two readers: the next developer and a grounded AI assistant. The architecture note explains each theme, layout and profile decision and which mechanism owns it. The grounding pack adds the answer rules the assistant must follow: answer from the course first and from the official documentation in the source map second; warn against brittle internal selectors and against changing a shared theme for every session when one session was meant; never propose CSS media queries as the primary responsive mechanism when responsive properties or `ClientProfile` fit; and ask for the profile and screen size when a responsive problem is ambiguous. The source map lists the documentation pages behind every rule so the assistant can cite rather than guess.

At the centre of the pack is the course decision tree, applied in order until a rung fits:

- Theme first, for the identity of standard controls.
- `AppearanceKey` for semantic variants.
- `CssClass` for scoped application CSS and third-party widgets.
- `CssStyle` for one-off dynamic values only.
- Layout containers for adaptive positioning.
- Responsive properties for designer-set profile values.
- `ResponsiveProfileChanged` for runtime behaviour.
- `ClientProfiles.json` for the profile definitions themselves.
- Production governance for maintainable delivery.

The rubric weighs theme design and consistency at 25 percent, responsive layout correctness at 25, `ClientProfile` and responsive properties at 20, maintainability and governance at 15, and explanation and grounding alignment at 15. Those last thirty points are earned by the note and the pack: if the assistant, reading only what you wrote, can explain why the details region is a modal on phone and why the metric card is a `CssClass` rather than an appearance, the console is finished.



Visual-system habit

Use the Wisej.NET mechanism that owns the problem: the theme for the identity of standard controls, AppearanceKey for semantic variants, CssClass for scoped app-specific CSS, CssStyle only for one-off dynamic values, layout containers instead of resize code, responsive properties for designer-set profile values, and ResponsiveProfileChanged for behaviour that is not simply visual.



Learning objectives

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

- ✓ Deliver a branded desktop/tablet/phone UI that uses theme, layout containers, and ClientProfile consistently.
- ✓ Review visual accessibility: focus indicators, invalid states, contrast, touch targets, tab order, and hidden content behavior.
- ✓ Prepare a production checklist for theme files, CSS files, ClientProfiles.json, screenshots, acceptance tests, and upgrade-safe customizations.
- ✓ Package source-map and RAG materials so a grounded AI assistant can answer implementation questions from the course.

Lab / Exam Guide

What you'll build in the hands-on lab, the suggested approach, and the required deliverables.



Hands-on lab

Complete the Adaptive Operations Console capstone: a custom theme with named tokens and at least four semantic appearances, scoped `CssClass` usage, a desktop shell with Flow, Table and Flex regions, `ClientProfiles.json` with phone, tablet, small desktop and desktop behaviour, responsive properties plus an idempotent `ResponsiveProfileChanged` handler, and a profile indicator visible in test mode. Run the accessibility review for focus frames, invalid states, contrast, touch targets and tab order, fill in the responsive QA matrix with screenshots for every profile, and write the architecture note and grounding pack that explain each theme, layout and profile decision.

Starting point

Open the Adaptive Operations Console solution you carried through Modules 1 to 6: `AdaptiveOps.theme` and the scoped `AdaptiveOps.css` exist, the shell is nested containers with Flow, Table and Flex regions, `ClientProfiles.json` lists six profiles and the Module 6 handler already shows the active profile name in the status label. Nothing new is built from scratch in this lab; every step hardens, reviews or documents what is there.

Suggested approach

- ✓ Audit the visual layers first: confirm the theme derives from a base theme with named tokens, count the semantic appearances assigned through `AppearanceKey` (add a `destructive-button`, `kpi-card` or `nav-item` if you have fewer than four), and move any stray `BackColor`, `ForeColor` or `Font` override into the theme or a `CssClass`.
- ✓ Harden the layout: set `MinimumSize` on every fill-weighted or docked region and `MaximumSize` where a region must not grow, remove `AutoSize` from large container hierarchies, and resize through every width to confirm nothing collapses or overflows horizontally.
- ✓ Rewrite the profile logic as one `ApplyProfile(string name)` method that assigns final values and returns early when the profile has not changed; call it from the constructor with `Application.ActiveProfile` and from the `ResponsiveProfileChanged` handler, unsubscribe in `Dispose`, and show `lblProfile` only when a test-mode flag is set.
- ✓ Run the accessibility review by operating the screen: Tab through every region and fix `TabIndex` and `TabStop`, trigger validation and confirm the invalid state has an icon or message as well as colour, check contrast for each token pair, add `ToolTipText` to icon-only buttons, and measure touch targets at phone width.
- ✓ Fill the responsive QA matrix (desktop, small desktop, tablet portrait and landscape, phone portrait and landscape) with the expected profile, a screenshot and the checks for each row, then write the architecture note and the grounding pack: decision tree, answer rules, source map and one question the assistant must answer from your material.

Required deliverables

- ✓ Finished Adaptive Operations Console with custom theme, semantic appearances, scoped CSS and Flow, Table and Flex regions
- ✓ ClientProfiles.json, responsive properties and an idempotent ResponsiveProfileChanged handler with a test-mode profile indicator
- ✓ Accessibility review covering focus frames, invalid states, contrast, touch targets and tab order
- ✓ Responsive QA matrix with desktop, tablet and phone screenshots in both orientations
- ✓ Architecture note and AI grounding pack explaining every theme, layout and profile decision

Student review questions

- ✓ Your `ResponsiveProfileChanged` handler runs twice for the Tablet profile after a browser refresh. Which line of yours guarantees the second run changes nothing, and what would the user see if that line were missing?
- ✓ A reviewer asks you to remove the focus frame from `action-button` because it "looks dated". What do you change instead, which token drives it, and which two other states do you re-verify in the Theme Builder afterwards?
- ✓ The ticket grid header needs a tweak that only a selector on the widget's internal DOM can reach. What do you try first, what must be true before the selector is allowed, and what do you write down so the next upgrade does not surprise the team?

Capstone hand-in

This lab is also the capstone submission for the course. Submit the source project, AdaptiveOps.theme, AdaptiveOps.css, ClientProfiles.json, the completed responsive QA matrix with its screenshots, the accessibility review, the architecture note and the grounding pack. Be ready to walk the console through each profile and to answer, for any control on screen, which mechanism owns its look and why.

Rubric area	Weight	Excellent looks like
Theme design and consistency	25%	Base-derived theme, tokens reused everywhere, semantic appearances for every business variant, no ad-hoc colour or font overrides.
Responsive layout correctness	25%	Containers own the layout, regions have minimum sizes, every QA-matrix width is usable in both orientations.
ClientProfile and responsive properties	20%	First-match profile order, designer-assigned values first, one idempotent handler for real behaviour changes, test-mode indicator.
Maintainability and governance	15%	Files in source control, diffs reviewed, CSS exceptions documented, screenshot matrix and accessibility review complete and repeatable after upgrades.
Explanation and AI grounding alignment	15%	Architecture note and grounding pack from which an assistant can explain each decision and warn about brittle choices.



Visual-system habit

Use the Wisej.NET mechanism that owns the problem: the theme for the identity of standard controls, AppearanceKey for semantic variants, CssClass for scoped app-specific CSS, CssStyle only for one-off dynamic values, layout containers instead of resize code, responsive properties for designer-set profile values, and ResponsiveProfileChanged for behaviour that is not simply visual.