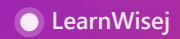


Production Localization Architecture, Testing and Capstone

Lesson Guide · Lab / Exam Guide



Lesson Guide

Localization that survives production: a thin `LocalizationService` over `ResourceManager`, domain code that returns values instead of sentences, an observable missing-key policy, a localization test matrix, pseudo-localization that exposes hard-coded strings, satellite-assembly deployment, and the `GlobalDesk` capstone. What this means for a developer who knows basic C# and Wisej.NET controls and now has to ship the application in more than one language — and how to read this module.



Production focus

Localization that survives production: a thin `LocalizationService` over `ResourceManager`, domain code that returns values instead of sentences, an observable missing-key policy, a localization test matrix, pseudo-localization that exposes hard-coded strings, satellite-assembly deployment, and the `GlobalDesk` capstone.

1. The production problem

After six modules `GlobalDesk` speaks two languages, but the machinery is scattered: designer resources on the customer editor, a helper over the shared `.resx` files, a language picker, a `CultureChanged` handler and an RTL demonstration. Nothing yet states what happens when a key has no translation, and nobody has proved the German build still fits a narrow window. A production localization system is predictable: it knows where each string comes from, which culture is active in this session, what a missing translation looks like, and what a language change does to open screens. This module turns those four answers into one small service, one written test matrix and the finished capstone.

2. Key concepts

Concept	Why it matters
<code>LocalizationService</code>	A small class that centralizes your application's choices around .NET localization instead of replacing it: one resource base name, one missing-key policy, one set of formatting methods.
Missing-key policy	The chosen, tested behavior when <code>GetString</code> returns nothing: a visible <code>[Key]</code> marker while you develop, a logged key plus neutral fallback text in production.
Product, system and diagnostic text	Three families with three owners: your own <code>.resx</code> strings, Wisej.NET's own labels through <code>Resources-[LANG].resx</code> , and log or exception text that stays in one developer language.
Culture-free domain	The model exposes a value such as <code>TicketStatus.Closed</code> and the UI maps it to a localized caption, so business logic and its tests never depend on a language.
Localization test matrix	A written list of runs — neutral, translated, region-falling-back, missing key, formatted values, runtime switch, long strings, RTL, system labels — so coverage is a checklist rather than a memory.
Pseudo-localization	A test resource set whose values are visibly altered and lengthened, such as <code>[!!! Save customer record !!!]</code> , built to expose clipping and hard-coded strings rather than to translate anything.
Satellite resources	.NET compiles culture-specific resources into satellite assemblies, so the culture folders produced by the build have to travel with the application when you publish or package it.

3. Lesson narrative

One service that owns every lookup

So far every screen has reached for the resource helper directly. That works, but it scatters decisions: which resource base name, what a missing key does, which culture formats a currency. A thin `LocalizationService` collects those decisions into one class and changes nothing else — it still reads through `Wisej.Resources.ResourceManager`, still relies on ordinary .NET culture fallback, and still formats with `Wisej.Web.Application.CurrentCulture`. What it buys you is a single place to change your mind, and a single place a reviewer has to read.

```

public sealed class LocalizationService
{
    private readonly Wisej.Resources.ResourceManager _resources =
        new("GlobalDesk.Resources.Strings", typeof(LocalizationService).Assembly);

    public string Text(string key)
        ⇒ _resources.GetString(key) ?? $"[{key}]";

    public string Currency(decimal value)
        ⇒ value.ToString("C", Wisej.Web.Application.CurrentCulture);

    public string Date(DateTime value)
        ⇒ value.ToString("d", Wisej.Web.Application.CurrentCulture);
}

```

Two details matter more than the code. First, `Text` never returns blank: an unknown key comes back as `[CustomerEditor.Save]`, which is impossible to miss on screen and trivial to grep for in a screenshot. Second, every formatting method reads `Application.CurrentCulture` at the moment of the call. Culture is per-session state in Wisej.NET, so a `CultureInfo` cached in a field — worse, in a static field — would quietly hand one user another user's language. Make the service the only place the UI touches `ResourceManager`, and a search of the solution for that type should return one hit.

Three families of text, three different owners

Product text is yours: captions, messages, validation wording, the sentences a translator will ever see, all living in your `.resx` files under stable semantic keys. Wisej.NET system text is the framework's own: the buttons on a `MessageBox`, the labels the built-in dialogs supply, overridden per language with a `Resources-[LANG].resx` file. Diagnostic text is for developers: what you write to a log or attach to an exception. It stays in one language on purpose, because a stack trace translated into German is a stack trace nobody can search for.

The same separation applies inside the application. A ticket is not "Closed"; it has a status, and `TicketStatus.Closed` is the value the model exposes. The UI, through the localization service, maps that value onto a resource key and asks for the caption. Domain code that returns sentences drags a language into your business rules and into every test that asserts on them; domain code that returns values keeps both stable while the UI gains a fourth language.

A test matrix instead of a spot check

Clicking the language picker once and seeing German captions proves very little. A useful localization test matrix is written down and rerun: the neutral culture; at least one translated culture; a language-region culture that has to fall back to its language resource; a deliberately missing translation, so you see the fallback and not a blank; date, number, percentage and currency values; a runtime culture switch with screens already open; a long-string pass; an RTL culture with a mirrored layout; Wisej.NET system labels in a `MessageBox`; and the validation and error messages, which are the strings teams forget most often.

Pseudo-localization needs no special tooling. Build a test resource file whose values are visibly altered and lengthened, such as `[!!! Save customer record !!!]`, and run the application against it. Accuracy is irrelevant; the point is that clipped buttons and cramped labels become obvious, and anything that stays

plain English during that run is a hard-coded-string candidate to fix. One caution: some Wisej.NET premium extensions wrap third-party JavaScript libraries whose localization has its own vendor culture setting, and extension configuration supports culture and locale placeholders for component resources. Treat that as an integration layer on top of your culture, and test the wrapped component after a language change instead of assuming its labels follow your `.resx`.

What has to travel with the published build

.NET normally compiles culture-specific resources into satellite assemblies, which the build drops into culture-named folders beside the application. `ResourceManager` finds them through the standard .NET lookup; there is nothing to load by hand. The deployment lesson is therefore short: do not discard culture-specific output folders when you publish or package the app.

That is the kind of mistake a deployment script makes silently. A copy step that takes only the main assembly, or an image built from a curated file list, produces an application that looks perfect on your machine and falls back to English everywhere else. Add one line to the capstone checklist: after publishing, open the output folder and confirm the culture folders are there.

Finishing GlobalDesk

The capstone is not a large application; it is proof that you can localize one end to end rather than translate a label. GlobalDesk needs a neutral `.resx` with a meaningful set of semantic keys, at least one translated resource file, one designer-localized screen, the `LocalizationService`, a language selector that sets `Application.CurrentCulture`, culture-aware date, number and currency display, a Wisej.NET system-resource demonstration, an RTL demonstration, translation-workflow notes and the QA checklist.

Rehearse the demo in a fixed order, because the order is the argument: start on the automatic browser culture, switch to German, show translated captions and formatted values changing together, trigger a `MessageBox` so a system label appears, show a deliberately missing translation falling back, switch to the RTL culture and look at the mirrored layout, then show the workflow notes and the checklist. If every step works without you editing anything, the localization is finished.



Localization layers habit

Decide where each string lives before you type it. Designer resources belong to one designed screen and are applied when its controls are created; shared application resources in `.resx` carry the text that code generates or several screens reuse; `Resources-[LANG].resx` overrides Wisej.NET system labels. Never store a formatted date, number or currency: keep the value and format it for the active culture at display time. And remember that culture is per session, so one user's language must never change another user's screen.



Learning objectives

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

- ✓ Create a small localization service around `ResourceManager`.
- ✓ Separate product text, system text and diagnostic text.
- ✓ Test fallback, formatting, RTL and runtime culture switching.

- ✓ Understand satellite-resource deployment at a high level.
- ✓ Complete the GlobalDesk localization capstone.

Lab / Exam Guide

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

Hands-on lab

Finish the GlobalDesk capstone. Wrap every resource lookup in a `LocalizationService` with `Text`, `Date`, `Number` and `Currency` methods, give it a missing-key policy that shows a bracketed key in development and logs the key in production, and make that service the only place the UI touches `ResourceManager`. Keep the domain out of it: the ticket model exposes a `TicketStatus` value and the UI maps it to a localized caption. Then run the localization test matrix over the finished application, covering the neutral culture, German, a language-region culture that falls back, a deliberately missing key, formatted dates and currency, a runtime switch, a long-string pass and the RTL culture, using a pseudo-localized resource set to expose anything still hard-coded. Confirm the culture folders survive a publish, and hand in the QA checklist as the capstone deliverable.

Starting point

GlobalDesk as you left it after Module 6: a dashboard and a designer-localized customer editor, `Resources/Strings.resx` with its German and Italian companions, the helper that reads them, a language picker that sets `Application.CurrentCulture`, a `CultureChanged` handler that reapplies text and recreates the editor, RTL support and your workflow notes. Every layer works, but resource lookups still happen in several places, a few runtime messages are still English literals in code, and no one has written down how the application is tested.

Suggested approach

- ✓ Write `LocalizationService` first: a single `Wisej.Resources.ResourceManager` over your `Strings` base name and the assembly, a `Text(key)` method that returns a bracketed `[Key]` marker when the lookup yields nothing, and `Date`, `Number` and `Currency` methods that format with `Application.CurrentCulture` read at call time. Give the missing-key path one switch: marker while developing, logged key plus neutral text in production.
- ✓ Make it the only door. Replace every remaining direct resource call and every English literal the user can see with a service call, then search the solution for `ResourceManager` and confirm the service is the only hit. Leave your log and exception text alone and add a comment saying it is diagnostic on purpose.
- ✓ Push culture out of the domain: have the ticket model return a `TicketStatus` value, add a small mapping in the UI or the service from that value to a resource key such as `TicketStatus.Closed`, and check that nothing in the model or its tests contains an English caption any more.
- ✓ Build the pseudo-localized resource set — the same keys with values lengthened and bracketed, for example `[!!! Save customer record !!!]` — run the application against it, and list every caption that stayed plain English and every control whose text was clipped. Fix both lists.
- ✓ Write the QA checklist as a table and run it: neutral culture, German, a language-region culture falling back to its language resource, a deliberately missing key, a date, a number and a currency

amount, a runtime switch with the editor open, the long-string pass, the RTL culture and a `MessageBox` system label. Then publish the application and confirm the culture folders and their resource assemblies are in the output.

The capstone is reviewed as one working application, not ten separate exercises: a neutral resource file with a meaningful set of semantic keys, at least one translated resource file, one designer-localized screen, the `LocalizationService`, a language selector that changes `Application.CurrentCulture`, culture-aware date, number and currency display, a Wisej.NET system-resource demonstration, an RTL demonstration, translation-workflow notes and the QA checklist. Rehearse the demo in a fixed order: start on the automatic browser culture, switch to German, show translated captions and formatted values change together, trigger a `MessageBox` so a system label appears, show a deliberately missing translation falling back, switch to the RTL culture, then show the workflow notes. Remember that culture is session state, so if anything you wrote caches a `CultureInfo` outside the session it will pass your own demo and fail the moment a colleague opens the app beside you.

Required deliverables

- ✓ `LocalizationService` with Text, Date, Number and Currency methods and a documented missing-key policy
- ✓ Domain code returning a `TicketStatus` value that the UI maps to a localized caption
- ✓ Pseudo-localized resource set used to find every remaining hard-coded string
- ✓ Completed localization test matrix covering fallback, formatting, runtime switching and RTL
- ✓ Published output checked for its culture folders, plus the GlobalDesk localization QA checklist

Student review questions

- ✓ A reviewer opens GlobalDesk in French, a culture you never translated, and sees English captions with correctly formatted French dates. Explain which mechanism produced each half of that screen, and say whether anything is broken.
- ✓ Your pseudo-localized run shows one label still reading "Save" in plain English. List the three places that text could be coming from, and describe how you would tell them apart without guessing.
- ✓ A colleague proposes caching the resolved `CultureInfo` in a static field on `LocalizationService` "because looking it up on every call is wasteful". Describe what a second signed-in user would experience, and what you would suggest instead.



Localization layers habit

Decide where each string lives before you type it. Designer resources belong to one designed screen and are applied when its controls are created; shared application resources in `.resx` carry the text that code generates or several screens reuse; `Resources-[LANG].resx` overrides Wisej.NET system labels. Never store a formatted date, number or currency: keep the value and format it for the active culture at display time. And remember that culture is per session, so one user's language must never change another user's screen.