Localizing a WPF app running on .NET Core 3 in 2020

I have recently updated my hobby desktop app, Embrace, from .NET 4.6 to .NET Core 3.1. The process was mostly painless as I was late to the game and most Nugets were already updated for .NET Core 3 / Netstandard support. I lost no functionality - except the ability to localize the UI.

The app runs in English, Czech, German, and Spanish, and the localization was powered by LocBaml, a legacy tool from Microsoft that was never fully supported and the development of which ended sometime 2005 or so.

Good-bye, LocBaml

Ancient as it is, LocBaml had one distinct advantage over competing approaches: I could leave literal strings in the XAML files, mark the elements containing them with `Uid`s, and the tool would extract them for me to translate. During build, the translations are then put into satellite assemblies. When the app starts, it decides which locale to use (based on user preferences) and loads the corresponding assembly.

The advantage of strings literals in XAML is hard to underrate. XAML looks and feels like HTML (for us web devs), and having human-readable strings makes it so much easier to edit, in the text mode as well as in the XAML Designer.

You see, the alternative route is to use RESX files, put your dictionaries there (originals as well as translations) and use keys instead of string literals in your XAML.

Compare this:

```
<Button x:Uid="ClickMeButton">Click me!</Button>
```

to this:

```
<Button Content="{ns:Loc ClickMeButtonText}" />
```

One disadvantage LocBaml has is, you cannot (easily) switch locales at runtime. Speculatively speaking, you might be able to re-load all of the UI but I haven't tried and the effort to make it worth would likely be disproportionate to the value delivered.

Nevertheless, LocBaml does not work with .NET Core 3 assemblies. What's a man to do? Go back to even more historical RESX files, of course!

Learning to love RESX again

My first steps were to re-learn what localization approaches

exist for WPF. To my dismay, there hasn't been any ground-breaking new technique developed since I first check. I wanted to keep my string literals but that wasn't to be.

Chance led me to this article about a simple approach of localizing a WPF app. It uses RESX files but with Bindings, and that means I could potentially switch locales at runtime, could I?

Yes, yes, that's indeed what this means.

A link at the bottom of the article then led me to this Github repo, that adds support for multiple dictionaries.

Embrace has multiple assemblies and even multiple frontends, and so this was directly applicable to my problem. I would not have to centralize all the texts but put them where they belong.

This solution is comparable to the WPF Localization Extension, which seems to be very popular. It's far lighter than that, however, just one C# file. I decided I could live without the extra complexity and features. The lighter my dependencies the better I think.

My learning steps

I have not used RESX files before, and so there was some learning curve to climb.

I created a file Strings.resx and one for each translation, i.e., Strings.cs.resx, Strings.de.resx, etc. For the dictionaries to be usable outside their assembly, I had to set the "Custom Tool" property to `PublicResXFileCodeGenerator`, otherwise they would only have the `internal` visibility.

To have a nice tree-view of related RESX files, I used the `DependentUpon` property of the `EmbeddedResource` declaration in the project file like this:

```
<ItemGroup>
    <EmbeddedResource Update="i18n\Strings.resx">
      <Generator>PublicResXFileCodeGenerator</Generator>
      <LastGenOutput>Strings.Designer.cs</LastGenOutput>
    </EmbeddedResource>
    <EmbeddedResource Update="i18n\Strings.en-US.resx">
      <Generator>PublicResXFileCodeGenerator</Generator>
      <DependentUpon>Strings.resx</DependentUpon>
    </EmbeddedResource>
    <EmbeddedResource Update="i18n\Strings.cs.resx">
      <Generator>PublicResXFileCodeGenerator</Generator>
      <DependentUpon>Strings.resx</DependentUpon>
    </EmbeddedResource>
    <EmbeddedResource Update="i18n\Strings.de.resx">
      <Generator>PublicResXFileCodeGenerator</Generator>
      <DependentUpon>Strings.resx</DependentUpon>
    </EmbeddedResource>
    <EmbeddedResource Update="i18n\Strings.es.resx">
      <Generator>PublicResXFileCodeGenerator</Generator>
      <DependentUpon>Strings.resx</DependentUpon>
    </EmbeddedResource>
  </ItemGroup>
```

Visual Studio support is there but won't overwhelm you with features. If you'd like to see all translations side-by-side, there's an extension ResXManager for this. I have a Jetbrains subscription, and ended up using its Localization Manager that does a comparable job.

Importantly, it can import / export dictionaries as CSV, which turned out to be super helpful.

One of the artifacts of the LocBaml build process is a bunch of CSV files with strings and in which usercontrol to find them. I wrote a Node.js script that extracted them from these CSVs and merged them into the format that Localization Manager could understood. It looks like this:

```
Path,Name,"en-US","cs","de","es"
Embrace.NetCore3/i18n/Strings,registerview.baml_YourNameLabel
```

Here's the gist of my translation helper.

I imported the generated CSV into Localization Manager, and only had to add the English dictionary by sifting through my XAML files, extracting the strings and replacing them with their dictionary keys.

I encountered a few gotchas along the way.

First and foremost, I thought that the default dictionary (**`Strings.resx`**) would be used if the app did not find one for the selected locale. It turned out I was wrong. The English originals had to be put in the **`*.en-US.resx`** resource, otherwise an exception was thrown, complaining about a missing satellite assembly.

Second, the XAML designer support is not 100% reliable. Some texts only ever show up as keys, others are translated in design-time, too. I am not a big enough Visual Studio internals geek to understand what's happening there.

Usage in XAML

This toolkit is intended for use in XAML markup, and it could not be easier to use.

The setup is straightforward. At the topmost level, you import the namespace and specify the default dictionary to use like this:

```
xmlns:resx="clr-namespace:Embrace.Common.i18n"
xmlns:loc="clr-namespace:WpfLocalizationWithMultipleResourc
loc:Translation.ResourceManager="{x:Static resx:Strings.Res
```

The first import points to the namespace where you have your resource files. The second imports the namespace of the localization helper. The third specifies the default dictionary.

The texts are then translated on-the-go like this:

```
<TextBlock Text="{loc:Loc trackeditor_baml_TrackTitleLabel}"
           x:Uid="TrackTitleLabel"/>
```

Because the resource lookup is based on a Binding, the user can select a different locale in Settings, the app informs the localization helper about the new locale, and all texts are refreshed automatically as a result of PropertyChanged event invokation. Sweet!

Usage in code

I have a few scenarios where I need to show a message to the user from code, e.g. a notification. In that case, I don't have any XAML.

Using the localization helper from code is not the primary use-case but it's possible.

In my App.xaml.cs, I add the default dictionary such that it's available downstream, and set the culture like this:

```
TranslationSource.Instance.AddResourceManager(Strings.Resou
TranslationSource.Instance.CurrentCulture = localCulture;
// localCulture is an instance of CultureInfo here
```

The localization helper uses the Singleton pattern (`TranslationSource.Instance`), of which I am not a huge fan but it feels appropriate here.

Later, when I need to access localized strings, I access the singleton and ask for them like this:

```
var translator = TranslationSource.Instance;
_statusBarService.StatusBarMessage = translator["Embrace.Comm
```

You'll notice the hard-coded path to the resource key, which happens to be the fully-qualified name of the resource file (incl. the assembly name). ~~I hope to fix that at some point but `nameof(...)` only returns the name without the assembly qualification.~~ I'll fix that with `typeof(Embrace.Common.i18n.Strings).FullName`.

Future steps

I have not given up on my string literals yet. In the future, I hope to come up with an approach that would let me use them again, perhaps adding a build step extracting them into RESX and replacing them with resource keys before compilation. I'll then update my notes with the solution.

What do *you* think? Sound off in the comments!

Your name or nick | Your email (optional)

Your website (optional)

Just write what you have in mind, don't worry about formatting.
Feeling fancy? You *can* deploy your Markdown-fu!

Post your comment

Feb 8, 2021 15:02 by Hossein

> Thanks for this, Doesn't LocBaml work with .net core assemblies? what about .net 5? I couldn't find MS saying its no longer supported.

Feb 8, 2021 16:02 by TomK

> You are welcome.
>
> The LocBaml tool was never production-ready and Microsoft even says so (https://docs.microsoft.com/en-us/dotnet/desktop/wpf/advanced/how-to-localize-an-application?view=netframeworkdesktop-4.8). In my estimation it was never supported.
>
> I have no clue whether you can get it to work with .Net 5, however, I haven't tried. Perhaps a good opportunity to revisit the topic! Have you tried?

Oct 8, 2021 00:10 by dude24816

> Checkout the RessourceDictionary-approach of this article: https://www.codeproject.com/Articles/37339/WPF-Localization might be a good way, too