

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

Contents

Localization Guidance for WPF	2
Localization Considerations	2
Unicode Support	3
Culture Mapping	3
Localizing Resources	4
Localizing WPF = Choices	5
Resources and Culture Formatting	6
Setting Culture and UICulture	6
Resources and Resource Fallback	7
Preferred Culture and ResourceFallback	9
Working with Resx Resources	10
Accessing Resources with the ResourceManager	14
Using XAML Resources and LocBaml to Localize Content	16
What can you localize with LocBaml?	19
Using Resource Dictionaries for Runtime Access to XAML Resources	20
Localizing with LocBaml	23
Enabling Localization in your Visual Studio Project	25
Run <i>MsBuild</i> to generate unique Uid's for each UI element in your XAML.....	29
Using LocBaml to Export Resources into a CSV file	30
Localizing Resources in the CSV File	32
Embedding Localized Resources back into Satellite Assemblies	33
BAML and Resx Resources Combined.....	34
Using an MSBuild Task	36
LocBaml – Not for the faint of heart.....	38
Localizing Resx Resources	39
Binding to Strongly-Typed Resx Resources.....	40
Organizing Static Resx Resources.....	42
Things to Consider with x:Static Resource Bindings	44

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

Custom Markup Extensions for Resx Resources	46
How the Custom Markup Extension Works.....	50
Attached Property Binding.....	55
Other Topics of Interest	60
Use Autosizing for Elements	60
Right To Left Display	61
Switching Languages on the Fly	62
Assigning a Resx Image Resource to an Image Control	64
Summary	65
Acknowledgements.....	66
Resources.....	66

Localization Guidance for WPF

Application localization is not a trivial task for any type of application scenario. The process is based on a few core principles that apply to WPF as they do to any other type of client application with a user interface. It is important for developers to understand the basic concepts of regional data display, locale-specific user interface customization and how to serve localized resources in both static and dynamic fashion. These concepts are very similar for most client applications but the actual process of localizing the static user interface components tends to vary between environments and WPF introduces yet another approach to resource localization for XAML resources.

This whitepaper will start with a quick review of general localization considerations for completeness, discuss how the .NET Framework handles resources for all applications, and then focus specifically on localization scenarios for WPF explaining some of the trade-offs within each approach.

Localization Considerations

Localization is the process of preparing an application to run in multiple locations. The .NET Framework provides very thorough support for localizing all types of client applications. That doesn't mean it's easy – localization is a very tedious and complex task and this whitepaper makes no attempt at covering the entire topic – it'd be enough to fill a book. In case you are new to localization concepts, this section and the next will serve as a short introduction to key localization concepts for .NET Framework applications.

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

Unicode Support

Unicode support is crucial for representing the wide variety of characters of the world's many languages seamlessly. This may seem obvious now, but it wasn't long before the .NET Framework was released that ANSI character sets and codepage translations were the norm. Unicode transcends the issues of these antiquated character set formats and their incompatibilities and provides an easy way to represent characters from multiple languages easily in the same string without having to worry about encoding issues.

The .NET Framework supports Unicode end-to-end – from the user interface all the way down to the database layer. This means that today, applications needn't do anything special to receive input, process or save data from different languages. In WPF most of the user interface content is encoded in static XAML documents which are just XML documents encoded in UTF-8 or UTF-16. Resx resources, the core resource format for .NET Framework applications, also are encoded in XML. These XML formats are Unicode compliant so the full range of characters supported by the world's languages can be directly expressed in XAML markup as well as in Resx resources that hold localized data.

As a point of interest, UTF-8 and UTF-16 are both capable of representing single-byte and double-byte character sets. UTF-8 does so in a more efficient way from a storage perspective, but is interpreted less efficiently for double-byte character sets. UTF-16 stores all characters using two bytes thus is a less efficient storage format but can yield more efficient interpretation of double-byte character sets. So, although UTF-8 will suffice for most scenarios, the latter is usually used specifically for double-byte scenarios.

Culture Mapping

Locations are defined – at least in the context of .NET localization lingo – as a language and country/region pair. For example *en-US* is for English in the United States or *fr-CA* for French in Canada. The country/region can also be omitted so *en* and *fr* are valid, non-specific culture identifiers which represent English and French without the regional code. Non-specific cultures are useful for storing resources that work for all regions, but are problematic for formatting and parsing since different regions often have different formatting rules for things like currency or date and time. For this reason the .NET Framework separates the concept of *UICulture* (for resources presented to the user which includes text, graphics and any other display elements) and *Culture* (for formatting currency, numbers, date and time, lists, sorting and so forth). It is quite possible to use a different culture setting for *UICulture* and *Culture* – the former used to determine which resources will be used to populate a localized user interface.

Culture information and manipulation in the .NET Framework is exposed through an instance of the *CultureInfo* class, which can be used to read the culture settings for a given locale, as well as setting a specific locale in an application. Every thread maintains a set of culture information stored as *CultureInfo* settings in *CurrentCulture* and *CurrentUICulture* properties. *CurrentCulture* determines behavior for

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

formatting and parsing (as previously discussed, for things like numbers and dates) and *CurrentUICulture* indicates the appropriate resources to load for the thread's locale. From any thread you can access *CurrentCulture* and *CurrentUICulture* from the following static members:

Thread.CurrentThread.CurrentCulture, *Thread.CurrentThread.CurrentUICulture* or *CultureInfo.CurrentCulture*, and *CultureInfo.CurrentUICulture*.

The main WPF thread will always be assigned a default setting for *CurrentCulture* and *CurrentUICulture*. You can also explicitly set these values based on user preferences (to be discussed). An important point to note, however, is that if a WPF application spins up new threads, those threads will not inherit the main UI thread's culture settings, in fact they will not be set at all! So, be sure to explicitly assign new threads with the appropriate values using the techniques discussed in this article.

Localizing Resources

The most prominent and tedious process of localization and the focus of this whitepaper is resource localization which refers to the process of translating the static pieces – primarily text – of an application. The idea is that the static pieces of an application – strings in labels, tooltips, menu items, headers, static messages and any other user interface content – are stored in such a way that they can be translated separately from the immediate user interface. The generally accepted approach for this is to provide a mechanism to extract relevant component properties as resources so that they can be stored and localized separate from the user interface with a copy for each supported culture. Resources are then applied to their respective component properties at runtime based on the current UI culture. An important benefit to this approach is that a single code-based is used for the application, and that future localizations to support new cultures should not impact the compiled code.

In WPF development there are two main approaches to resource localization: XAML-based and Resx-based.

XAML-based localization involves grouping localized content per XAML document rather than specifically mapping individual resources to specific XAML element properties. The XAML documents in the application serve as the base resource storage mechanism for all static content that is created in XAML. The idea is that for the neutral language you just create your XAML without any special markup or bindings for localization. A tool called LocBaml can then be used to export all the localizable, static content from compiled applications in a CSV text format. The exported resources can then be localized for each specific culture. Finally LocBaml can be used to merge and compile the localized resources into culture specific satellite assemblies (or binary resources). This approach is purely static resulting in compiled XAML (BAML) resources for each supported culture, where each culture-specific BAML resource contains the entire set of localized resource values. The approach is efficient but completely static and there is no way to interact with the way that resources are individually loaded. LocBaml is a

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

command line tool provided by Microsoft as a utility sample application and the process of using it is rather rigid and potentially error prone.

Resx-based resources are the traditional approach for .NET Framework application localization. They are XML-based files that compile into binary resources and can be accessed by the application through individual resource keys. Resx support is deeply integrated in the .NET Framework and Visual Studio, although there's no specific WPF experience in Visual Studio or in other XAML designers such as Blend. Binding Resx resources to XAML elements involves using bindings and manually mapping resources to individual properties with WPF binding syntax. The Resx designer and strongly-typed resources are still available in WPF projects so the resource editing process is straight forward. There's no official story for Resx resources in XAML so it's no surprise there are a lot of different approaches available for using Resx resources with XAML in the developer community. Two common approaches are using `x:Static` bindings to strongly-typed resources or using custom markup extensions. Resx resources are more granular than BAML resources as they are individually mapped to properties, but it also takes a little more effort to map resources to element properties at design time.

Localizing WPF = Choices

As you can see there are a couple of different approaches available for resource localization with WPF. The one you choose depends on the way you like to work and whether your localization process happens once the application is complete or incrementally while you are building the application.

The LocBaml approach is a very static process and best done when the application does not change frequently. Its big benefit is that you don't have to do much during development to get your application ready for localization as you can simply create XAML content in your default language and defer localization until later. The process of using LocBaml is rigid and fairly complicated using a command line tool that exports CSV files for the actual task of localization. Each culture has to be individually exported, localized and explicitly re-generated. In addition LocBaml has to be integrated into the build process explicitly to build localized output. It's not for the faint of heart.

Many Resx based solutions have sprung up in the developer community to simplify things. Resx is well supported in the Visual Studio environment so it's easy to enter and edit Resx resources. Resx localization is an open and extensible architecture that is more applicable in incremental localization processes and provides more flexibility in binding resource values. But it does require more forethought and interaction *while* the application is built as you have to map resource keys and bindings at development time explicitly in your XAML markup.

As you can probably tell by these descriptions localization support in WPF and Visual Studio lacks the sophistication that you might be used to in Windows Forms. With Windows Forms there was a clear path to localization with the Windows Forms Designer integrated solution. In WPF there's no such clear path – the choice is left up to you and either way requires building or using custom built components or command line tools. In this whitepaper the goal is to demonstrate the various approaches available and

highlight their pros and cons so you can make an educated choice about which solution works best for you.

Resources and Culture Formatting

Before jumping into specific solutions there are a few important general purpose concepts that are relevant to .NET Framework culture formatting, localization and WPF. For the latter, although the LocBaml and Resx-based approaches use resources differently both technologies use the resource location and loading features of the .NET Framework. The following section provides some background on these concepts.

Setting Culture and UICulture

As was previously discussed the current thread's Culture and UICulture setting is very important to achieve culture-specific formatting and content presentation for end-users. Both the `CurrentThread` and `CultureInfo` type exposes static properties to access the current Culture and UICulture as follows:

```
CultureInfo ci = Thread.CurrentThread.CurrentCulture;  
CultureInfo ci = Thread.CurrentThread.CurrentUICulture;  
CultureInfo ci = CultureInfo.CurrentCulture;  
CultureInfo ci = CultureInfo.CurrentUICulture;
```

You will typically use the `CurrentThread` type since it allows assignment of a new *CultureInfo* instance to change the current thread's settings. For example, the following code illustrates how to set the current Culture and UICulture to a hard-coded value:

```
CultureInfo ci = new CultureInfo("en-US");  
Thread.CurrentThread.CurrentCulture = ci;  
Thread.CurrentThread.CurrentUICulture = ci;
```

By default the UI thread will be initialized with a Culture and UICulture matching the regional settings of the machine that the application is running on. In other words, when the application starts it inherits the Culture and UICulture from operating system. Formatting will simply use the Culture settings specified. The application will attempt to use resources localized for the locale of the UICulture setting – assuming they exist. If they don't exist, the runtime uses a resource fallback process to find a suitable resource to present to the user – and this will be discussed in the next section.

While using the default Culture and UICulture of the operating system is useful in some cases – it is not realistic to assume that all users are running on a version of the operating system that matches their culture preferences. It is always a good idea for applications to provide a way for users to configure their culture preferences – either during installation or through some form of application configuration options. Ideally you want to use a configuration setting in the application's configuration file or a database to allow configurable selection of the locale used.

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

Assuming that the user's culture preference is known, where should Culture and UICulture settings be initialized in the lifecycle of a WPF application? For WPF applications the best place to explicitly set the application's Culture and UICulture is during the application's startup, in the main constructor. This ensures that all resources including system error messages and the initial XAML document to be loaded see the correct culture settings on start up.

The following code initializes culture settings from a configuration value in the application constructor found in App.xaml.cs:

```
public App()
{
    // Set application startup culture based on config settings
    string culture = ConfigurationManager.AppSettings["Culture"];

    if (!string.IsNullOrEmpty(culture))
    {
        CultureInfo ci = new CultureInfo(culture);

        // Force application to work with $ regardless of culture
        // Demonstrates customization of culture for app (optional)
        ci.NumberFormat.CurrencySymbol = "$";

        Thread.CurrentThread.CurrentCulture = ci;
        Thread.CurrentThread.CurrentUICulture = ci;
    }
}
```

This configuration assumes that Culture and UICulture are one and the same – but it is possible to use a specific culture for Culture and a neutral culture for UICulture. In fact Culture must be set to a specific region such as “en-US” or “fr-CA” while UICulture can be set to “en” or “fr” which would indicate the neutral language localization is acceptable rather than that of a specific region.

Setting the Culture and UICulture at application startup need only be done once unless you provide the user with a way to change their culture preferences while the application is running. In this case, you should of course save any changes to your application configuration and then restart the application so that the application can be reloaded for the correct culture. A later section of this whitepaper will also discuss dynamically changing cultures in a running application.

Resources and Resource Fallback

Resources are stored in and loaded from assemblies – one per each specific localized culture. Once a particular resource assembly is loaded resources are cached in a *ResourceSet* in the application domain, where each *ResourceSet* represents a single set of resources for a specific culture, equivalent to a single Resx file's content.

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

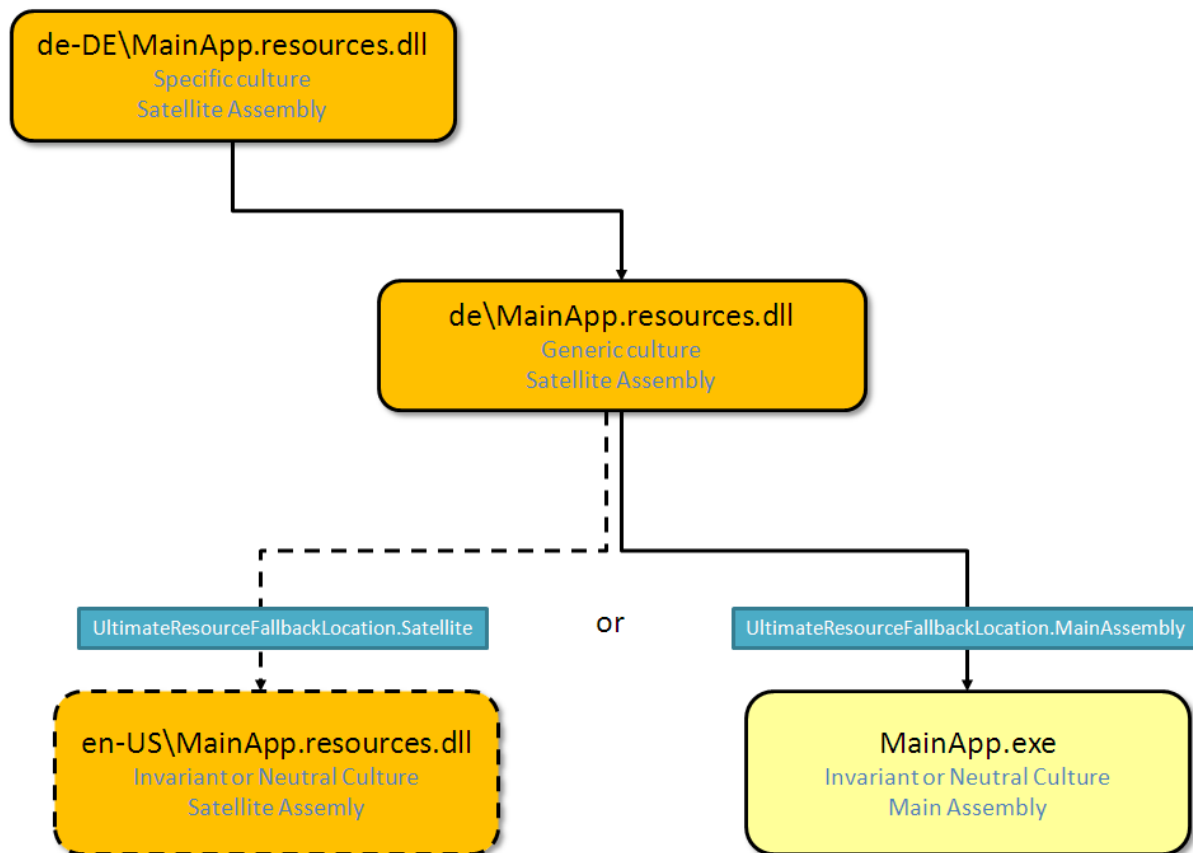
The *ResourceManager* type is used to retrieve resources for a specific culture. If a matching resource for the current culture doesn't exist – whether it's because there are no resources at all defined for this culture or whether a resource key is missing - the application falls back to the nearest resource match using a vital concept called *Resource Fallback*. Resource Fallback searches for most relevant resources down the culture hierarchy which means that cultures are searched from most specific to least specific.

For example, if an application is compiled with resources in US English (en-US) and the application is executed with a German UI Culture (de-DE) the fallback hierarchy looks like this:

1. Specific Culture (de-DE)
2. Neutral Culture (de)
3. Default Culture or Neutral Culture (en-US)

Figure 1 illustrates this fallback process.

Figure 1: Resource Fallback in an application works from most specific culture to the default culture with resources retrieved from culture specific assemblies



WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

Localized resource assemblies are called satellite assemblies – and they are located in culture-specific folders beneath the main application folder. In Figure 1 satellite assemblies are located under \de-DE and \de folders.

The main application assembly typically contains a set of resources called default resources or neutral resources – often based on the English culture. By default, these are the resources that are loaded if the runtime cannot find a matching resource key in the satellite assemblies in the fallback hierarchy. However, the main assembly can include a `NeutralResourceLanguageAttribute` setting which indicates which resources are stored inside the main assembly, or where to find the ultimately fallback satellite assembly. The following setting indicates that resource fallback should use the main assembly, and that any requests for en-US should immediately load from the main assembly rather than searching the satellite assembly folders:

```
[assembly: NeutralResourceLanguage("en-US",  
    UltimateResourceFallbackLocation.MainAssembly)]
```

This setting should be used with projects that use only localized Resx resources. You can also indicate a specific satellite assembly as the fallback assembly as follows:

```
[assembly: NeutralResourceLanguage("en-US",  
    UltimateResourceFallbackLocation.Satellite)]
```

The latter is **required** if you use localized BAML resources as localized BAML resources are always read from a satellite assembly, never from the main assembly.

Preferred Culture and ResourceFallback

Depending on how serious you take your localization task you need to plan for how you want to work with the culture hierarchy and which languages you want to localize for. Generally speaking it's more efficient to localize for non-specific cultures. So rather than localizing for *de-DE* (German in Germany) it's more useful to start by localizing for *de* (generic German), so that resource fallback can kick in for users of *de-AT* (Austrian German), *de-CH* (Swiss German) and so on. In many localization scenarios localizing for the non-specific language is sufficient. You can then do localization for the specific cultures as necessary to satisfy customers.

Resx Resources are easy to incrementally localize because individual resource keys follow resource fallback. If a resource is missing for the given specific culture it falls back to the neutral culture for the locale, and finally to the default culture (associated with the main assembly) to return a value. This means that you can localize the generic culture like *de*, *fr*, *es* and then add only resources that need to be customized for specific cultures like *de-DE*, *fr-CA*, *es-MX*. Typically only a few resources must be translated in specific culture. The neutral culture for each locale should contain all culture keys, but the default culture for fallback must contain all keys to avoid runtime exceptions.

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

BAML resources contain entire localized XAML documents as content so resource fallback applies to the document as a whole rather than for individual resource keys. Each BAML document has to localize all resources – effectively there's no resource fallback for individual resource values, only for the entire BAML document. Even if there's a difference of one single word between cultures a new copy of the entire BAML binary must be created. This means that the process of localizing BAML content is a lot less iterative and works best when the application is completed and will require few changes. The exact process for Resx and XAML localization is discussed in later sections of this paper.

Another thing to consider related to the user's preferred culture settings is the impact on culture formatting. When users select a preferred culture you may allow them to select a non-specific or specific culture. The associated code is what you use to set the Culture and UICulture setting in a WPF application at startup. Keep in mind, however, that the Culture must be set to a *specific* culture since formatting is always based on regional settings. You can assign the CurrentCulture property an instance of `CultureInfo("de-DE")`, but creating `CultureInfo("de")` would throw an exception. In the event the user selects a non-specific culture you should use the static method `CreateSpecificCulture()` to assign the CurrentCulture from a non-specific culture as follows:

```
Thread.CurrentThread.CurrentCulture =  
CultureInfo.CreateSpecificCulture("de");
```

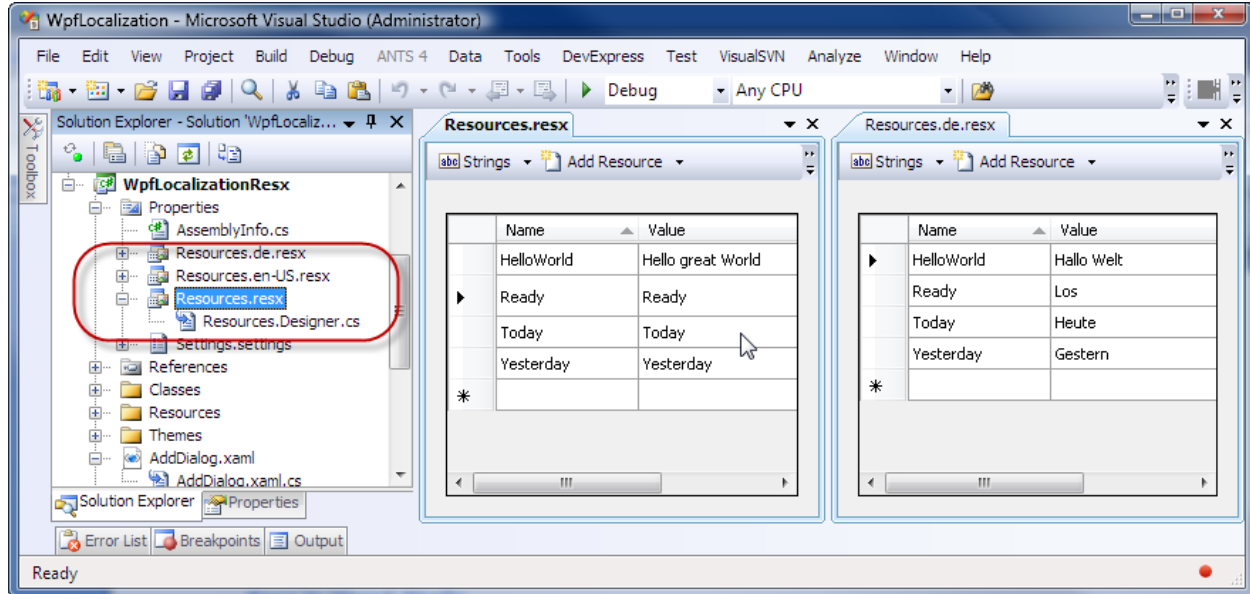
Working with Resx Resources

Resx resources are stored in .resx files and enjoy first class support in Visual Studio. You can create Resx resources simply by adding a resource file to the project and adding resource keys and values. To localize a resource file, simply copy the default resource file and rename it to match the locale. Localized resources include a locale identifier before the .resx extension. For example a file named Resources.resx in the main assembly would be named *Resources.de-DE.resx* or *Resources.de.resx* for the specific (de-DE) or neutral (de) German locale version. Figure 2 illustrates working with resources in Visual Studio.

Figure 2: Resx Resource support in Visual Studio is rich and makes it relatively easy to examine and modify resources interactively although there's no direct side by side editing/sync support

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

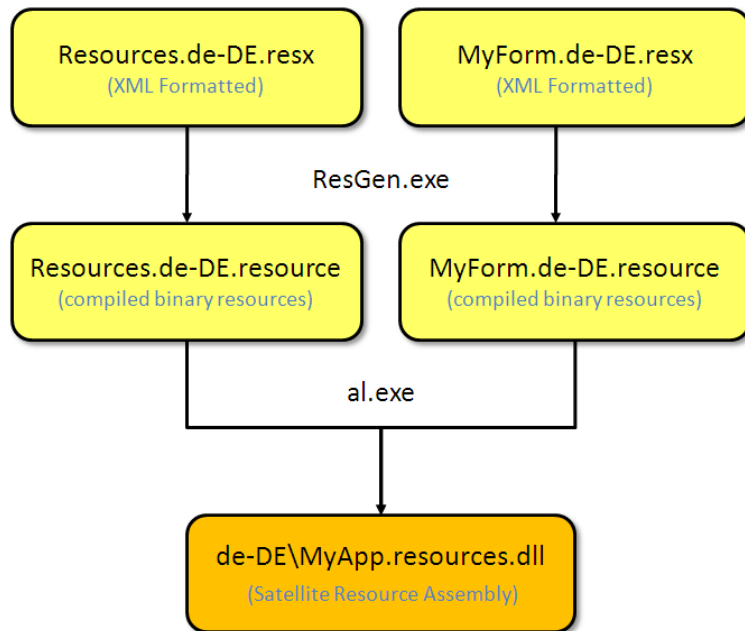


Resx resources are stored in XML based `.resx` files and automatically compiled into `.resource` files that represent a binary image of the resources. This binary image is then embedded in the appropriate assembly. The main assembly includes all default resources. When a project includes localized resources with the appropriate file naming convention, satellite assemblies are automatically created for each locale beneath culture-specific folders such as `\de-DE` and `\de`. These satellite assemblies include the appropriate localized resources. Figure 3 illustrates how satellite assemblies are generated from Resx resources.

Figure 3: Resx resources are XML-formatted and compile down to assembly resources embedded in the main assembly or satellite assemblies for specific cultures

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

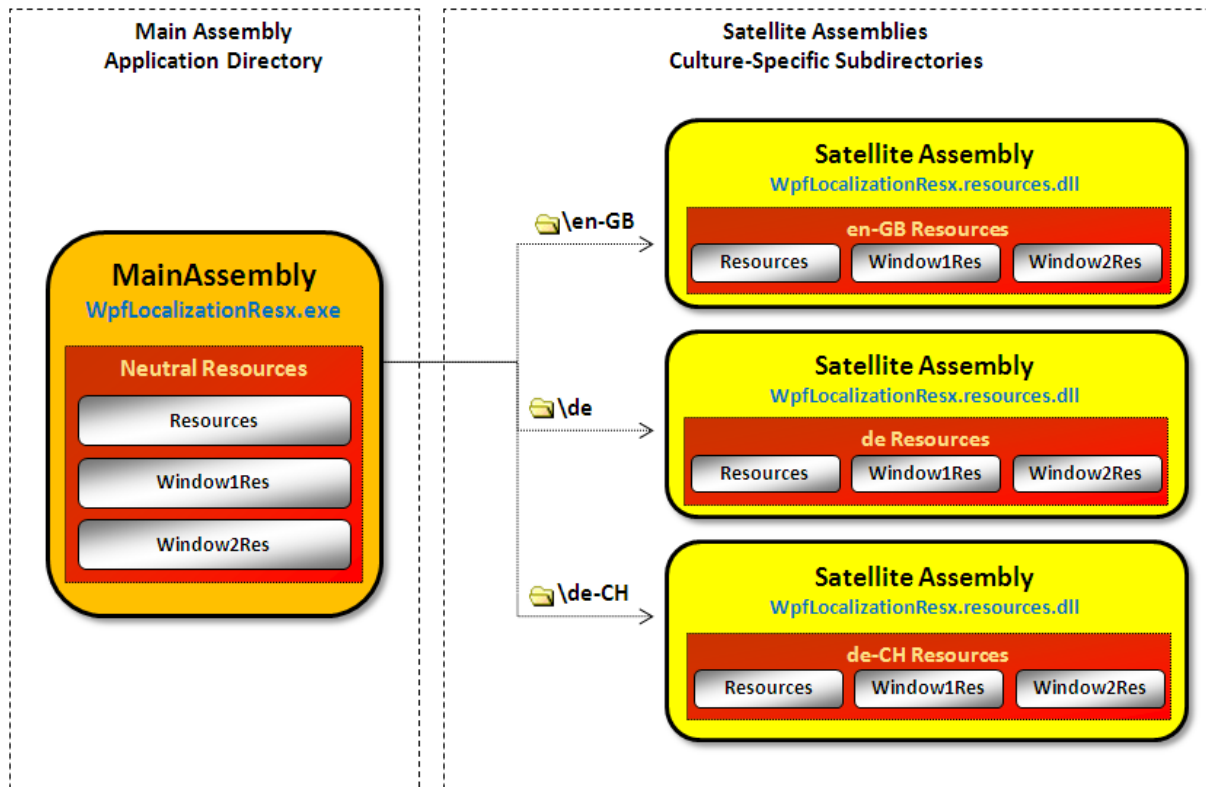


As mentioned, resources for each specific culture are stored in separate satellite assemblies which are placed in folders below the application's target output folder. Each culture gets a folder which in turn contains a satellite assembly with the same name as the main assembly (.exe or .dll) with a .resources.dll extension as shown in Figure 4.

Figure 4: Localized Resx resources are stored in satellite assemblies below the application folder and contain localized versions

WPF Localization Guidance

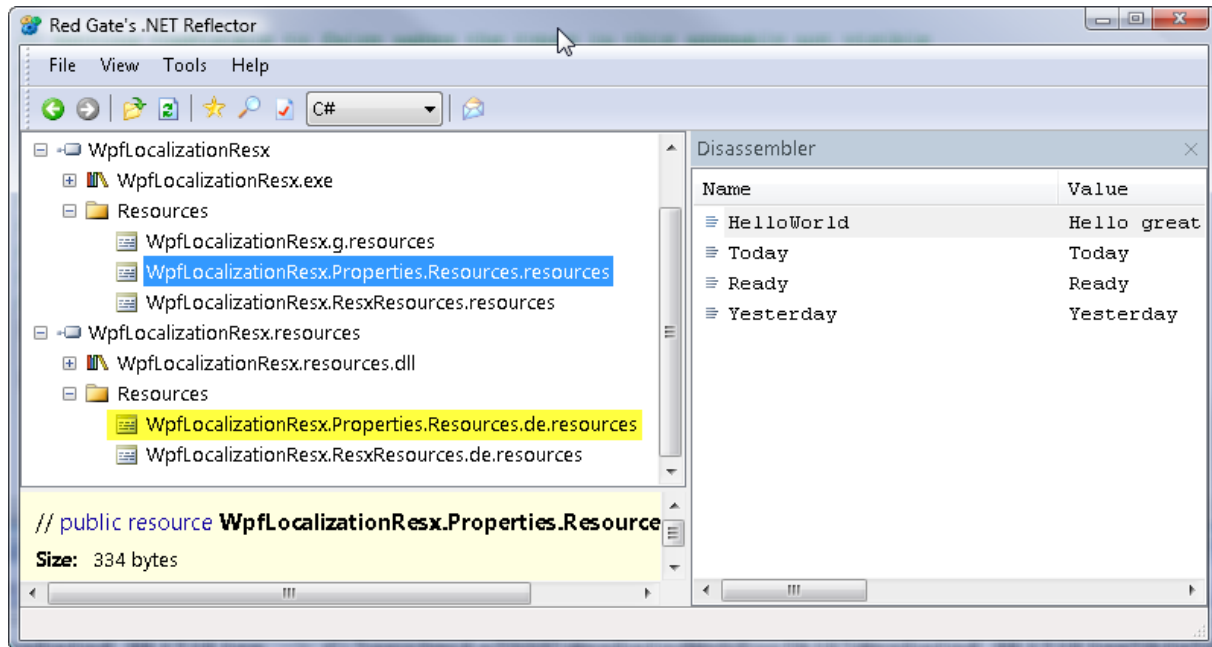
Rick Strahl & Michele Leroux Bustamante, June 2009



Resx resources are stored internally as resource sets which is a collection of resource keys and values. Each resource set corresponds to a Resx file and contains any number of individual resource keys that can be localized. In Figure 4 there are three resource sets: The default Resources.resx file, and two other optional resx files that are used by the application. When the application is compiled satellite assemblies are automatically generated for any localized resources. So if you have Resources.resx and you also create a Resource.de.resx the compiler automatically creates a satellite assembly in the \de folder with those resources compiled into it.

Default resources are a special case and depending on how your project is set up they can be stored either in an external satellite assembly or inside of the main assembly. As mentioned earlier, if you use Resx resources in combination with BAML localization you *must* store default resources in a satellite assembly. Figure 5 illustrates how the resources are laid out in a compiled assembly and satellite assembly using Reflector.

Figure 5: Resx resources are compiled into an assembly and stored in resource sets that represent each Resx file with key/value pairs for each resource entry



ResourceSet names are important as they determine how a *ResourceManager* finds and loads these resources. Each ResourceSet name is made up of the default project namespace (WpfLocalizationResx) plus the path to the resource in the project's hierarchy (Properties) plus the name of the file (Resources) all separated by a period: WpfLocalizationResx.Properties.Resources.

Accessing Resources with the ResourceManager

The *ResourceManager* type will by default load resources for the current thread's *UICulture* setting, but you can also explicitly create a *ResourceManager* for a particular culture. *ResourceManager* loads resources into memory and caches each *ResourceSet* instance so they are loaded only once from the resource store. Resources are cached until explicitly unloaded or until the application domain is unloaded.

To access resources you create a *ResourceManager* instance for a specific *ResourceSet* located in one of the application's main assemblies. Because resources are reused and *ResourceManager* internally caches *ResourceSet* instances the *ResourceManager* instance is typically declared static:

```
public static ResourceManager ResGlobal =  
    new ResourceManager("WpfLocalization.Properties.Resources",  
        typeof(App).Assembly);
```

You can then access resources like this:

```
string hello = ResGlobal.GetString("HelloWorld");
```

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

```
Bitmap bitmap = ResGlobal.GetObject("CountryFlag") as Bitmap;
```

The resource set is specified in the constructor as string-based resource path that corresponds to the full name of the resource, such as `WpfLocalization.Properties.Resources`. You must also specify the assembly from which the resources are loaded. Optionally you can also pass a specific culture – and if the parameter is omitted as it is in the example, the current `UICulture` is used.

Visual Studio 2005 and later also creates strongly-typed resource classes that map the resources in Resx files into a strongly-typed class so you don't have to access the resource manager directly – these classes manage an underlying static *ResourceManager* instance for you and expose static properties for each resource key. If you create a resource file called `MyResources.resx` Visual Studio automatically creates a strongly-typed class in `MyResources.designer.cs` and `MyResources` class to access those resources.

Using the strongly-typed resources is easier than raw *ResourceManager.GetString()* calls as they are discoverable with Intellisense and provide compile-time type checking to ensure you're referencing valid resource keys. The following illustrates using strongly-typed resources:

```
// using WpfLocalization.Properties;
string hello = Resources.HelloWorld;
Bitmap bitmap = Resources.CountryFlag;
Bitmap bitmap2 = Resources.ResourceManager.GetObject("CountryFlag")
    as Bitmap;
```

The resource manager in strongly-typed resources always uses the current `UICulture` to load resources so if the `UICulture` is changed at runtime this is reflected immediately. If you want to retrieve resources for a specific culture you can access the *ResourceManager* property on the class directly and explicitly use the *ResourceManager's* methods:

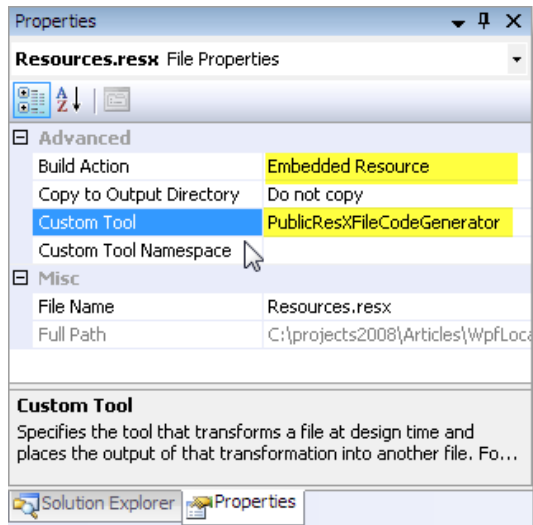
```
MessageBox.Show(WpfLocalization.Properties.
    Resources.ResourceManager.GetString("HelloWorld",
    new CultureInfo("de-DE")));
```

By default, when you add a Resx resource to a project using Visual Studio an internal class is created. For WPF usage it's important that the generated class is marked as *public* rather than *internal*, so that WPF can use XAML bindings with strongly-typed resources. Specifically ensure that you set the Custom Tool to generate these strongly-typed classes to use *PublicResxFileCodeGenerator* instead of the default *ResxFileCodeGenerator* as shown in Figure 6.

Figure 6: To enable WPF binding to strongly-typed resources make sure you use *PublicResxFileCodeGenerator* on resources to generate public rather than internal classes

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009



Individual properties must also be public if you want to use strongly-typed resources as XAML binding sources. If you don't use the custom tool to generate resources, you can modify the scope of each property in the Resx Resource Editor in Visual Studio – through the *Access Modifier* drop down in the toolbar.

Strongly-typed resources are definitely a key feature to WPF for binding to Resx resources. This is discussed further with Resx Binding implementations later in this whitepaper.

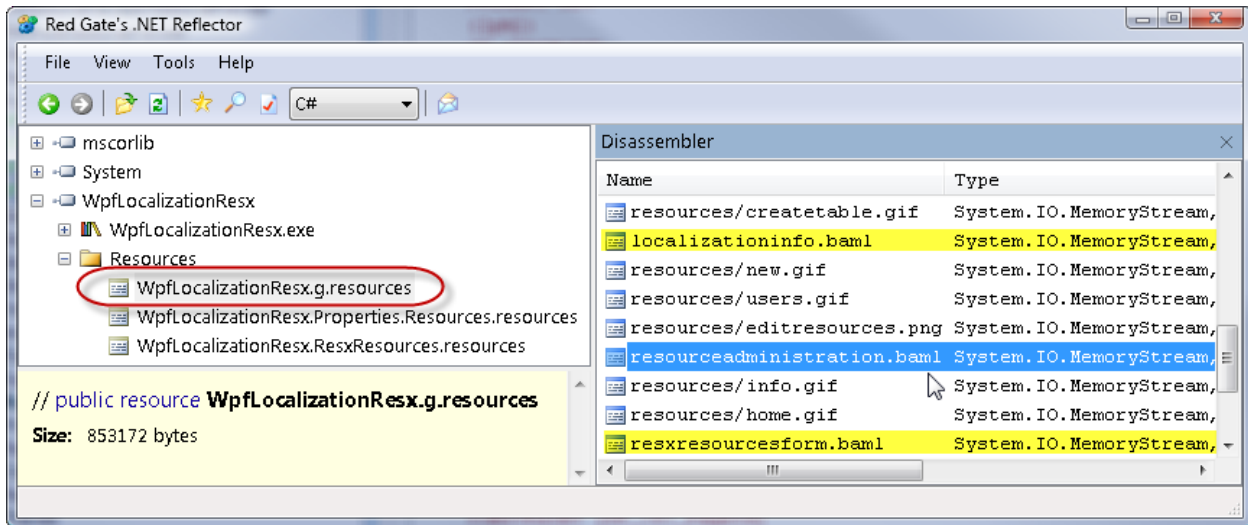
Using XAML Resources and LocBaml to Localize Content

WPF introduces a new resource localization approach based on XAML resources. A XAML document itself is a text based XML document, which is compiled into a Binary XAML (BAML) document. These BAML documents are stored as resources inside the compiled assembly of the application or component, with each BAML document becoming a resource entry. Figure 7 illustrates how each BAML document is stored as a resource in the output assembly.

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

Figure 7 - Compiled XAML (BAML) documents in WPF are always loaded as binary resources as shown here in [.NET Reflector](#)



Compiled BAML resources can be localized using a tool called *LocBaml* provided by Microsoft. LocBaml allows extraction of individual resource values from all BAML resources into a single CSV text file. Once the extracted values in the CSV file have been localized, the CSV file can be merged into localized BAML resources in culture specific satellite assemblies.

The LocBaml tool is *sample application* with source code provided by Microsoft. To use it, download the sample, build it and then copy the executable to your application's output directory and run it from there.

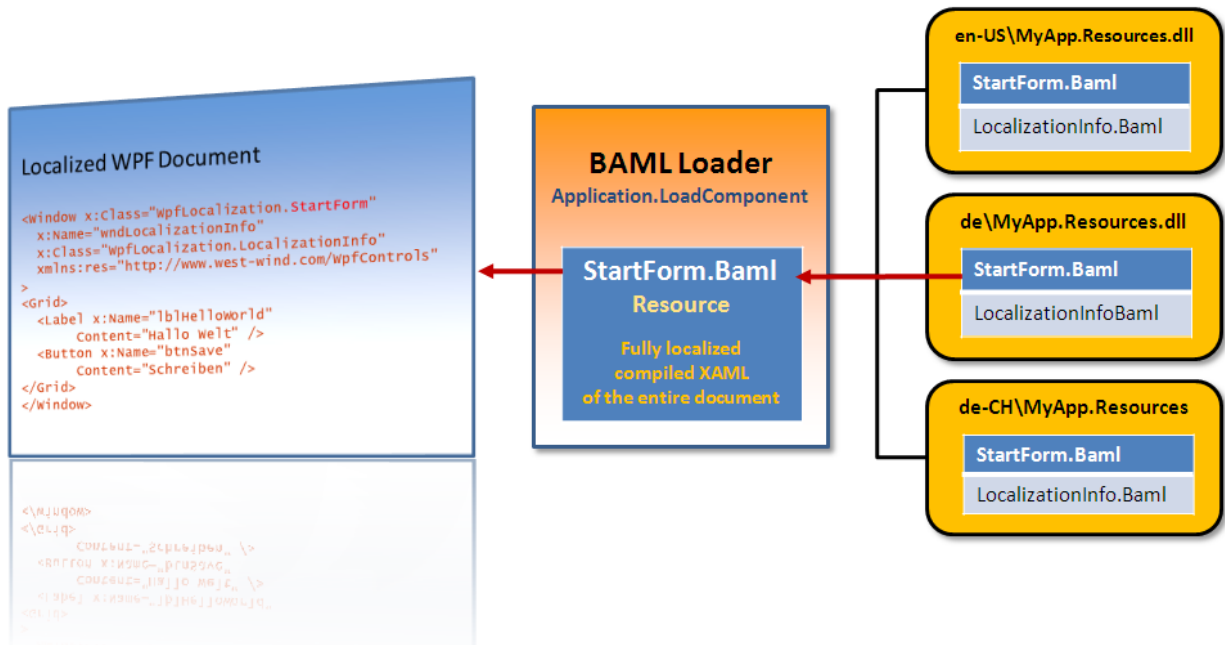
Download LocBaml from here: <http://msdn.microsoft.com/en-us/library/ms771568.aspx>

If a WPF application is localized and localized BAML resources have been generated – the application will load localized BAML resources at runtime based on the main thread's *UICulture* (see Figure 8). Essentially, when a XAML document is to be loaded the runtime loads the localized version of the XAML document. Compared to Resx resources which render localized resources one element or property at time, the BAML approach is very efficient.

Figure 8: BAML resources are served from satellite assemblies as entirely localized documents

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009



This is a key concept behind LocBaml localization. Rather than mapping individual resource keys to resource entries, the entire XAML document is treated as one big resource container. As a developer this means you can create your application using the neutral language without paying particular attention to localization of the content. You simply embed static text into the page for the neutral culture and let LocBaml worry about pulling the relevant content out for localization at a later point in time. When the time comes to localize you can export all localizable resources from the all XAML documents in the application into a single localizable CSV file.

The CSV file contains all static content from all of your XAML documents in the application including all text as well as quite a bit of format information like Width, Height, Margin, Padding and anything that relates to layout that was manually set in the XAML document.

At this point it's up to the localizer to localize the neutral culture CSV into each of the supported languages and cultures. Typically this means renaming the CSV file to a culture specific version and updating text in the CSV file in a text editor or a CSV editing tool.

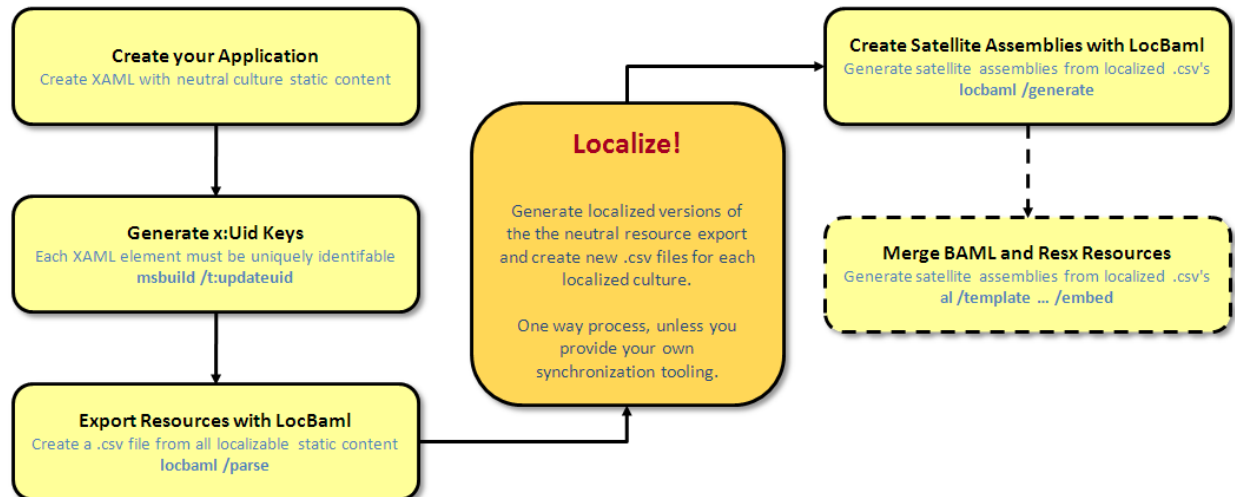
Once localization is complete the localized CSV files needs to be merged back into BAML resources. LocBaml provides the */generate* option to create either compiled resources (.resources files) or a complete satellite assembly for a given culture. If you are just localizing with LocBaml and have no Resx resources in your project compiling directly to satellite assemblies is easiest. If you are using Resx assemblies, you have to combine the Resx resources with the generated BAML resources as LocBaml always overwrites any existing assembly files. For the latter process it's necessary to invoke the assembly linker and pick up Resx resources out of the temporary build folder and combine them with

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

the .resources file created by LocBaml's /generate option. The flow of LocBaml localization is shown in Figure 9.

Figure 9: Localizing with LocBaml involves a fairly rigid step by step approach



The localization process with LocBaml is fairly rigid and follows a step by step approach. Using just the tools provided by Microsoft the process is one way – once you export your XAML resources to CSV and localize, there's no support for re-synching another LocBaml export back to the already localized versions of CSV files. The process of re-synching a second resource export to already localized resources is entirely left up to you.

For this reason, it's safe to say that unless you build custom synchronization tools for CSV files, the process of LocBaml localization is one way: You should complete your application release entirely first, then localize the content with LocBaml and finally compile the application with the updated resources. This involves setting up a build process that merges the BAML resources into the appropriate satellite assemblies each time you compile the application.

The LocBaml localization process is a bit rough as Microsoft has provided only minimal tool support via an unsupported sample application. LocBaml is based around the complex process of replacing binary BAML documents in resource assemblies. The tool itself also suffers from a few limitations: It has to run out of the target application's output folder in order to find any dependent assemblies and XAML can't represent binary content so any non-textual content has to be stored elsewhere – either in loose resources or in Resx files.

What can you localize with LocBaml?

LocBaml allows you to localize XAML content almost completely by exporting any localizable content from the original XAML document. Practically everything that you see in a XAML document is also available in the extracted CSV file. This includes text that is typed into labels, tooltips, headers, menu

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

captions and so on, assigned values to properties and Widths, Heights, Margins, Padding as well as resource dictionaries that can contain non-UI content. When looking at the CSV file you get the feeling that you're looking at a deconstructed XAML document – most of the same information is still there but it's now available in the row based, comma delimited CSV format that can be edited and localized.

Using LocBaml the original, default culture XAML document does not require any forethought for localization, because the LocBaml export simply spits out everything localizable from the original XAML document in the exported CSV file. You can create your document as you normally would by hard-coding text and other localizable content without any special markup or formatting. LocBaml pulls out any localizable content along with a unique identifier for each framework element and makes it available for editing. When using LocBaml all XAML content is automatically ready for localization.

Compiled BAML resources at runtime are loaded as one big resource stream per document and the compiled BAML code includes all localized content inside of it. A localized BAML document is a full copy of the neutral XAML document with static content replaced with the localized values created by LocBaml. With the exception of resource dictionaries there are no individual resource values that can be accessed via code in the BAML– the localized content is applied as static default values when the document loads.

Using Resource Dictionaries for Runtime Access to XAML Resources

Since you may also need resources that can be accessed via code or XAML binding you can also embed non user interface resources into XAML documents using resource dictionaries. Resource dictionaries are common in WPF for styling and declarative code assignments, but they can also hold arbitrary resource values that can be accessed via code or explicit bindings at runtime. Resource dictionaries are defined in XAML and localized right along with all other XAML content in a document, but unlike static property values which are simply assigned at load time, the resources defined in a resource dictionary are accessible at runtime.

This is useful for things like displaying localized messages in a dialog prompt or status bar for example. XAML resources are stored in a resource dictionary within a **<Resources>** section of a XAML element. As an example, here's a set of string resources embedded into a XAML Window element:

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
    x:Class="WpfLocalization.SimpleForm"
    x:Name="Window"
    Title="SimpleForm"
    Width="640" Height="480">
<Window.Resources>
    <!-- Embedding static resource strings into the page -->
    <sys:String x:Key="Ready">Ready</system:String>
```

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

```
<sys:String x:Key="Today">Today</system:String>
<sys:String x:Key="Yesterday">Yesterday</system:String>
</Window.Resources>
```

This resource dictionary contains three string values that are embedded into the default resource dictionary of the Window. The namespace for the CLR type must be visible to the XAML document. For example, the prefix “sys:” in the example stands for *System* in *mscorlib* which is defined in the <Window> element in this namespace declaration: `xmlns:sys="clr-namespace:System;assembly=mscorlib"`.

You can also store resources in an external file to avoid cluttering up your main XAML document. This makes sense if you have many static standalone localization strings in a document or application. In the following example two external resource dictionaries are merged into the current Window’s resource dictionary:

```
<Window.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="GlobalResources.xaml" />
            <ResourceDictionary Source="SimpleFormRes.xaml" />
        </ResourceDictionary.MergedDictionaries>
        ... more resources here
    </ResourceDictionary>
</Window.Resources>
```

In this scenario one of the resource dictionaries is merged from an external file *GlobalResources.xaml* that holds the resource strings mentioned earlier:

```
<ResourceDictionary
    x:Name="WpfLocalization.GlobalResources"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:system="clr-namespace:System;assembly=mscorlib">
    <system:String x:Key="Ready">Ready</system:String>
    <system:String x:Key="Today">Today</system:String>
    <system:String x:Key="Yesterday">Yesterday</system:String>
</ResourceDictionary>
```

Once resources are declared in a resource dictionary you can access these resources in code. Resources are searched for up the container chain from current document to the containing element all the way up to *App.xaml*. For example, to access a resource with the identifier "Ready" from the code behind a particular XAML document you can do the following:

```
this.StatusBarMainPanel.Content = this.Resources["Ready"] as string;
```

If you want to search the entire resource hierarchy *FindResource()* is more thorough:

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

```
this.StatusbarMainPanel.Content = this.FindResource("Ready") as string;
```

The resources in a resource dictionary are localized along with all the rest of the XAML resources when using LocBaml, so the entry will be loaded from the appropriate satellite assembly or main assembly based on the current thread's *UICulture* setting. This behavior is similar to using *ResourceManager* or strongly-typed *Resx* resources in code.

Resource dictionaries present you with another choice for localization. Assuming you use LocBaml for XAML document content, where should you store other non-static resources? You can keep all arbitrary runtime loaded resources in resource dictionaries or you can create *Resx* resources for these – which means using XAML and *Resx* resources side by side. If you go the LocBaml route, keeping resources in XAML is preferable because it keeps the localization process consistent. The alternative, localizing some resources using LocBaml and some using *Resx*, means you must localize the CSV and individual *Resx* files. Mixing XAML and *Resx* resources also complicates the compilation process since LocBaml doesn't merge XAML resources into existing resource files that already contain *Resx* resources. This issue will be further explored in a later section that talks about the LocBaml localization workflow.

Aside from using resource dictionary entries in code, you can also bind entries to XAML elements:

```
<TextBlock Text="{StaticResource Today}"></TextBlock>
<TextBlock Text="{StaticResource Ready}"></TextBlock>
```

If you are using the LocBaml approach, there is little need to bind elements to resource dictionary entries unless there is some common text reused in many places (for example every status bar has an initial "Ready" caption). Shared resources can reduce the localization effort and reduce errors.

Beware however, as there is a fairly serious gotcha when accessing resource dictionary entries stored in *App.xaml*. Consider this global resources dictionary defined in *App.xaml*:

```
<Application.Resources>
  <ResourceDictionary x:Uid="ResourceDictionary_1">
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="GlobalResources.xaml" />
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</ApplicationResources.Resources>
```

The following code uses *FindResources()* to access the *Ready* key:

```
MessageBox.Show(this.FindResource("Ready") as string);
```

Resources will load just fine, but unfortunately value returned is always from the default resources and not the appropriate localized resource – regardless of the *UICulture* setting. For resource fallback to work properly you must load resources directly into each XAML document's resources section:

```
<Window.Resources>
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary
            Source="ApplicationResources.xaml" />
    </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
</Window.Resources>
```

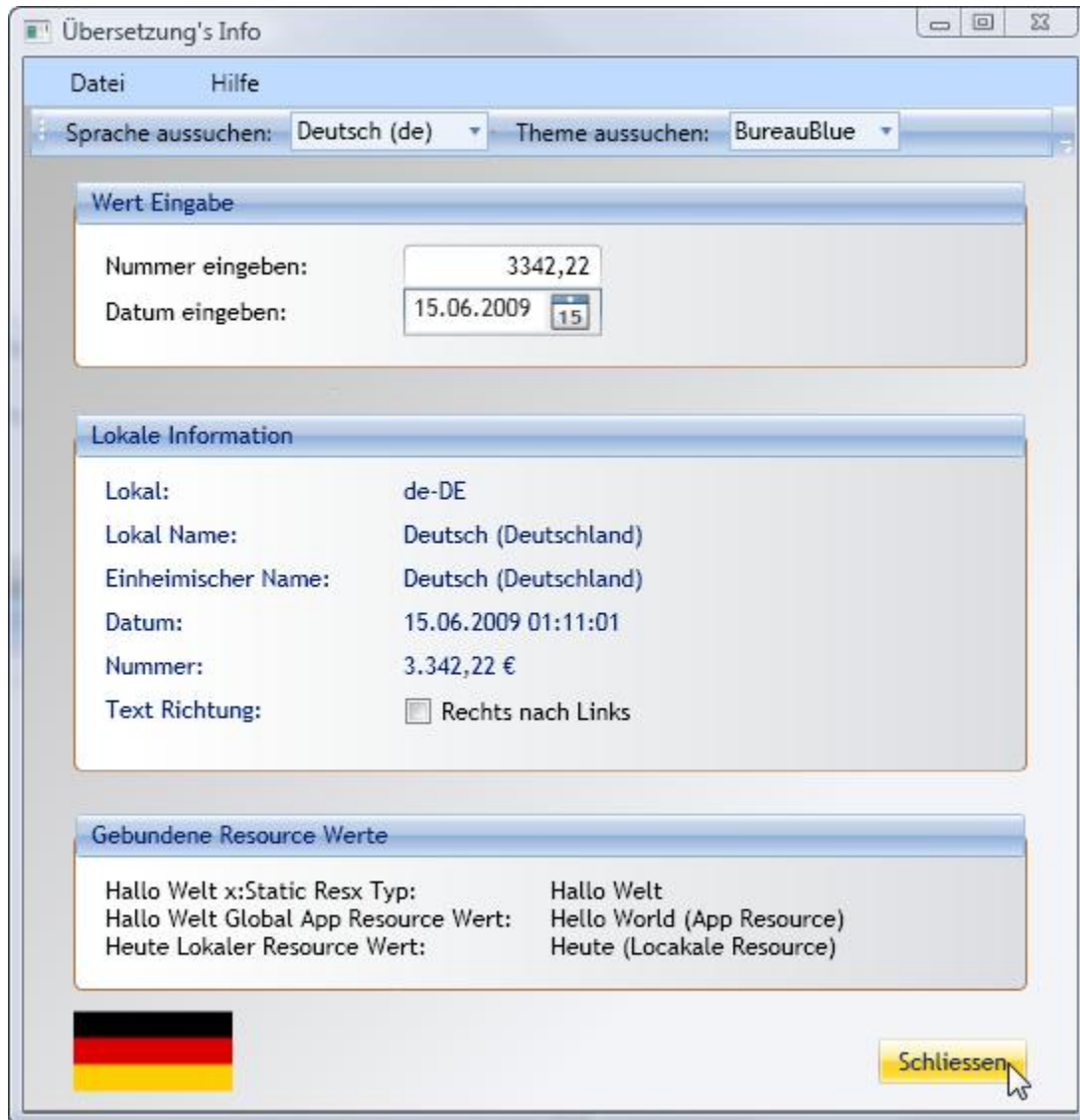
Localizing with LocBaml

Up to now you have learned that if you use the LocBaml approach you can design your XAML documents without thinking about localization, and with this approach you would probably create resource dictionaries (shared or embedded in each Window) for dynamic resources. Now that you have an idea of how LocBaml works, let's take a closer look at the process for localizing an application with it. Consider a very simple application that consists of a startup form and a main LocalizationInfo form that displays some localization related status info as shown in Figure 10.

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

Figure 10: A LocBaml sample application that includes a couple of forms including this Localization Information form that displays culture and resource settings



You can find this WpfLocalizationLocBaml application in the code samples accompanying this whitepaper.

The LocBaml localization process involves these steps (some still to be discussed):

- Layout your application using XAML without special concerns for localization
- As much as possible complete the application before localizing
- Enable localization in your Visual Studio project file by setting `<UICulture>`

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

- Setting the default `NeutralResourcesLanguage` in your project
- Running `msbuild /t:updateuid` to ensure unique x:uid attributes are created
- Run `LocBaml /parse` to export resources into a single large CSV file
- Create a localized copy of the CSV file for each culture
- Save each localized CSV file with a new culture-specific name
- Run `LocBaml /generate` against the localized CSV file to create a .resources or assembly file containing compiled BAML resources. If you're not using Resx resources generate directly into a satellite assembly. If you are using Resx resources generate to .resources files and run the assembly linker to compile the .resources file plus any other Resx resources (if any) into a single satellite resource assembly.

Enabling Localization in your Visual Studio Project

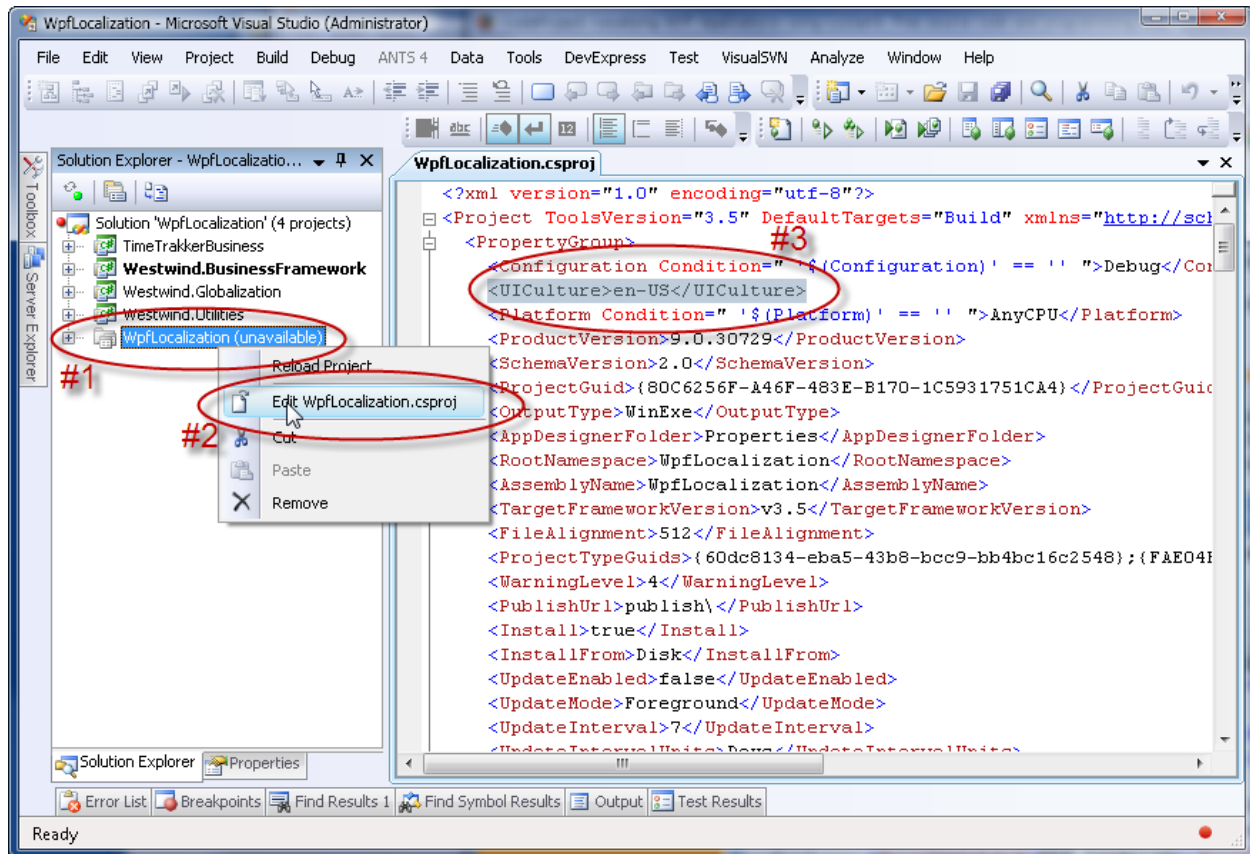
By default the BAML localization features in a WPF project are disabled in Visual Studio and the compiler does not export localized BAML resources into separate resource satellite assemblies. Enabling localization in your project is a two step process. First, you add a `UICulture` setting to the project file. Second, you set the assembly's neutral resources language.

The first step to make a WPF project culture aware is to set the neutral `<UICulture>` in your project file. This project setting enables the project compilation process to generate localized resources for each culture and cause the compiler to generate satellite assemblies. Looking at Figure 11, to do this from within Visual Studio first unload the project using the *Unload Project* option on project's context menu (#1) in the Solution Explorer. Right-click on the project node and select *Edit YourProject.csproj* from the context menu (#2) – which opens the project file as an XML document. Set the `<UICulture>` element in the project file to your default culture (#3).

Figure 11: BAML Localization requires that your VS Project has the `<UICulture>` key set to your neutral culture

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009



Once you've added this value, close the project file, then right-click on the project node again and select *Reload Project*.

The next step is to configure the default fallback culture and the location of the fallback assembly. To do this open up the *AssemblyInfo.cs* belonging to your WPF project and uncomment the following line:

```
[assembly: NeutralResourcesLanguage("en-US",  
    UltimateResourceFallbackLocation.Satellite)]
```

The culture specified should be the default culture in which resources are encoded in the project's XAML files such as en-US. The value here should match the value set in the project file's `<UICulture>` setting. These two flags in combination cause the compiler to emit resources as culture specific resource satellite assemblies.

Neutral Satellite Assemblies and Resx Resources

Resx resources stored in the main assembly are known as default or neutral resources and all other culture-specific resources are stored in satellite assemblies. Using `UltimateFallbackLocation.Satellite` changes the resource fallback behavior so that it requires that default resources are stored in a separate satellite assembly. This is an uncommon

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

setting when working with Resx resources and if you're not careful to provide the default satellite assembly it will cause problems with Resx resources not being found by your application at runtime.

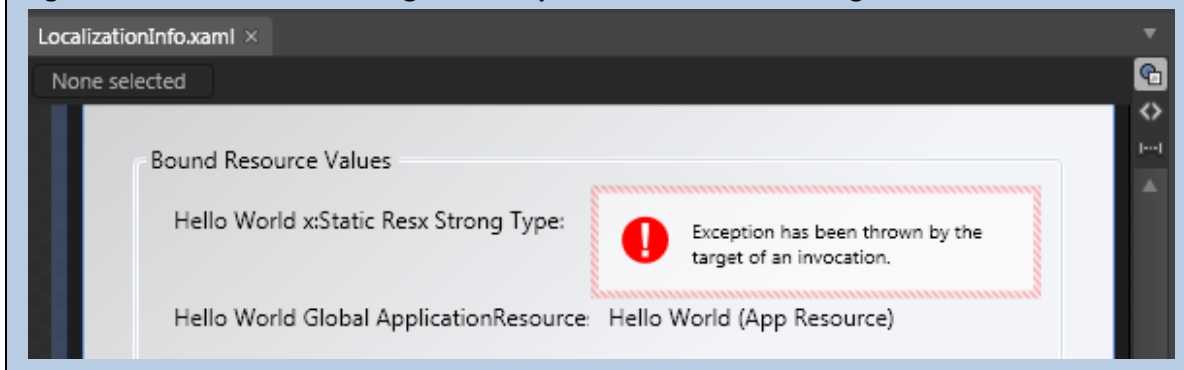
For Resx resources this means that you can no longer rely on non-culture-specific Resx files like `Resources.resx` to provide the necessary default resources. Instead you must supply resources for the specific culture `Resources.en-US.resx` representative of the default resource set. However, if you also want strongly-typed resources you must also provide the neutral `Resources.resx` file since Visual Studio generates strongly-typed resources from non-culture-specific Resx files. In the end this means you need **both** `Resources.resx` and `Resources.en-US.resx` even if both files hold the same resources. This applies to all resource files in your project and means that you need to copy these two files frequently to keep them in sync.

Note also that Expression Blend cannot bind to static resources in satellite assemblies, so if you're using static bindings to resources like this:

```
Content="{x:Static props:Resources.HelloWorld}"
```

Blend will always display an error as it can't access the satellite assembly that holds the neutral Resx resources (see Figure 12). The same layout does however work in the Visual Studio designer.

Figure 12: Resx resource bindings fail in Expression Blend when using Satellite assemblies

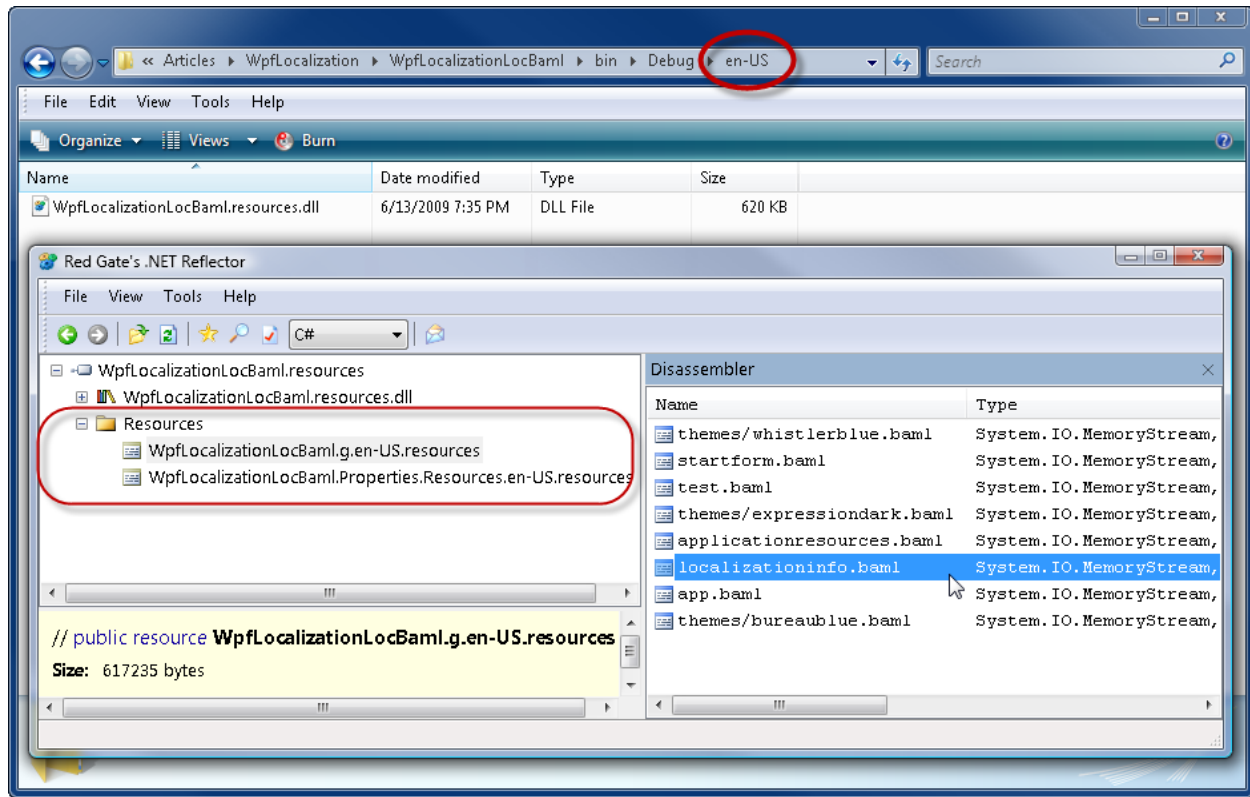


If the *UltimateFallbackLocation.Satellite* setting indicates "en-US" resources as the fallback, then a satellite assembly must exist beneath the `\en-US` folder below the project's output directory for fallback to work properly. Figure 13 shows the content of the satellite assembly for the default en-US culture in the project `WpfLocalizationLocBaml`.

Figure 13: Resources exported when compiling with Localization enabled in your project (this project contains both BAML and Resx resources)

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009



Note that both BAML and Resx resources are stored in the resource assembly. BAML resources all load into a single resource set (WpfLocalizationLocBaml.g.en-US.resources) in the resource assembly. The resource set contains all XAML documents in addition to any assets added to the WPF project that use the *Resource* build action (automatically applied to project assets). In Figure 13 there's also one Resx resource set included for WpfLocalizationLocBaml.Properties.Resources which contains all the individual resources keys defined in Resources.en-US.resx.

By default the compiler will only create a BAML resource satellite assembly like this for the configured neutral resource satellite assembly. Satellite assemblies for other cultures are only created for Resx resources present for other cultures. As already discussed, BAML resources must be manually created and added to satellite assemblies after localization is complete using `LocBaml /generate`. As you'll see in a minute this causes some additional complexity in the build process.

To revisit the point, if it's possible to avoid using Resx resources in combination with BAML resources do so. It'll make the localization and build process quite a bit cleaner for LocBaml scenarios. Although a valiant goal, this may not always be possible. While you can use loose binary resources (*Resource* Build Action in Visual Studio) to hold arbitrary resource content like images, these resources are not localizable. You must place a copy of each content item with a properly localized filename for it to be included in satellite assemblies. If you need localizable binary resources then you must use Resx in combination with BAML resources are your only option.

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

Run *MsBuild* to generate unique *Uid*'s for each UI element in your XAML

The next step ensures that all XAML UI Elements in your project have *x:Uid* attributes associated with them. These *x:Uids* are used to identify each element to be localized in the XAML document uniquely. You might wonder why you need *Uids* when you already have *x:Name* attributes in your documents. *x:Name* attributes are typically not assigned to every element and these attributes generate properties in the code behind class. If names were used for every single XAML element your class would become unmanageably large very quickly. Instead *x:Uid* is used separately to uniquely identify each element on the page along with a tool to generate *Uids* for every element in all the XAML documents in your project.

LocBaml requires these unique *Ids* in order to extract resources to localize and then sync them back up after localization when the localized content is merged with the original assembly. The merge process uses the neutral resource assembly to match *Uids* to those in the localized CSV content and merges the two to write new localized BAML resources. Any missing values are pulled from the base assembly you use for the LocBaml merge process.

Only elements that have *x:Uid* associated with them are exported with LocBaml's parse function and while you can manually add these attributes to your XAML, there's an easier way: Microsoft provides an *MsBuild* task to generate *Ids* for you on every localizable element.

Open up a Visual Studio Command Prompt, and change path to your project's base folder. Then type the following to generate the *x:Uid* attributes for all XAML elements:

```
msbuild /t:updateuid WpfLocalizationLocBaml.csproj
```

If you now open up any of your XAML documents you should find that each and every element has an *x:Uid* attribute associated with it:

```
<MenuItem x:Uid="mnuFile" Header="_File" x:Name="mnuFile">
    <MenuItem x:Uid="mnuSave" Header="_Save" x:Name="mnuSave"/>
    <MenuItem x:Uid="mnuExit" Header="_Exit" x:Name="mnuExit" \
</MenuItem>
```

If you have an *x:Name* attribute on an element, the id generation uses that name for the *Uid*. If no *x:Name* is specified a *x:Uid* based on the element type is generated with incremental values for each element name generated:

```
<MenuItem x:Uid="MenuItem_1" Header="Help">
    <MenuItem x:Uid="MenuItem_2" Header="Help Contents" />
    <Separator x:Uid="Separator_1" />
    <MenuItem x:Uid="MenuItem_3" Header="About" />
</MenuItem>
```

Clearly the first example is the better choice as it will result in resource names that are identifiable in the

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

exported resource files. It is generally a good practice to give every localizable element a meaningful name (and consequently a *x:Uid*) so the localizer has a clue as to what property she is dealing with.

The WpfLocalizationLocBaml project folder contains MakeUids.bat that can run this task for you. Keep in mind that you should to re-rerun this task anytime you add new components to a document to ensure that elements end up localizable so creating a batch file that you can quickly access and run.

Using LocBaml to Export Resources into a CSV file

The next step is to export all XAML resources available in a given assembly to a CSV file. To do this you will need to use LocBaml, which the Microsoft provided sample application discussed earlier. Recall that you must download the zip file of the project, extract it into a folder and build the project to produce LocBaml.exe.

A compiled version of LocBaml.exe is provided with the WpfLocalizationLocBaml sample project under the *\tools* folder.

Once you've compiled LocBaml.exe you need to copy it into your target folders (bin/Debug or bin/Release) and run it from Command Window or Powershell. LocBaml has to run in the output folder of your application's main executable in order to find and load any private assembly dependencies.

If en-US is specified as the default culture and satellite assembly output for resources, the assembly bin\Debug\en-US\WpfLocalizationLocBaml.resources.dll would be input to LocBaml.exe to produce the CSV file. You can send CSV file output to a Res folder beneath the project root. Here is an example of the command line to run LocBaml from the \bin\Debug folder (all one line):

```
locbaml /parse en-US/WpfLocalizationLocBaml.resources.dll
        /out:../../res/en-US.csv
```

This produces a file called en-US.csv in the Res folder. The folder has to exist or LocBaml will fail. The output contains comma delimited entries for all resources in all XAML documents of the application. If you have a sizable project this file will be very large. The raw output of a single line looks like this (each of these two items on a single line):

```
WpfLocalizationLocBaml.g.en-US.resources:simpleform.baml,
mnuFile:System.Windows.Controls.MenuItem.$Content,Menu,
True,True,,#mnuSave;#mnuExit;
```

```
WpfLocalizationLocBaml.g.en-US.resources:simpleform.baml,
mnuFile:System.Windows.Controls.HeaderedItemsControl.Header,Menu,
True,True,,_File
```

Content can be partially encoded to get around the obvious problems of commas and line breaks

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

embedded in text. Anything that contains commas or line breaks is embedded in double quotes ("this text, has a comma"), and quotes become ". Here's an example of a line with commas and quotes:

```
WpfLocalizationLocBaml.g.en-
US.resources:test.baml,WindowMessage:System.Windows.Controls.TextBlock.$Conte
nt,Text,True,True,, "In the following text blocks the display is always in the
OS culture, and not the current culture which is displayed as the first item.
Testing a some &quot;quoted text as well&quot;;. "
```

The text exported from LocBaml is safely encoded. However, it's up to you and whatever tool you use to localize the content to keep the same format intact as you edit the document. Accidentally adding a comma or forgetting to escape a string can easily break the CSV file format. Depending on what type of editor you use, the editor also may not properly encode strings with commas or quotes. Be careful and be sure to back up frequently.

The actual fields for each entry in the CSV file are shown in Figure 14.

Figure 14: Fields of the LocBaml exported CSV file

CSV Field	Description
1. Baml Document	The BAML document that contains the resources
2. Resource ID	The fully qualified window relative Resource ID
3. Category	Type of resource that
4. Readable	Visible to localizer
5. Modifiable	Modifiable by localizer
6. Comments	Any localization comments
7. Value	The value of the resource.

If you open up the file in a text editor each line's content is rather long and it's difficult to separate the actual content from ids and noise. The cells you're typically interested in are 1,2 and 7 and maybe 6 so it also helps to have the ability to reorder the columns of the CSV file.

Most CSV editors will happily import the UTF-8 formatted CSV file, but only save it back to disk in the current Windows ANSI code page (such as Windows 1252). It is possible to use a CSV editing tool as long as you remember to convert the documents to UTF-8 before you run LocBaml against them otherwise you get funny looking symbols for upper ASCII characters. Visual Studio's editor is actually quite good at converting documents between encoding formats. You can use the *Save As | Save with Encoding* option that converts a document to UTF-8. If you play with CSV editors chances are you will need this functionality.

If you look a little closer at the CSV content notice that there's a lot more information than just text shared in the exported resources. There's containership information – for example the menu item shown in the first CSV example actually holds references (#mnuFile, #mnuExit) to other elements that

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

make up the content. There are also many settings such as margins, height and width, radius, alignment and so on. Essentially any fixed values are exported so you can potentially modify layout problems in the localized resources should localized content not fit.

This allows you to modify the layout of a document through the localization data which is useful in some special cases, but tricky if you actually make changes to the underlying user interface. For example, imagine that you have form and change its width to accommodate a longer text field in the localized CSV file. Later you go in and change the actual neutral language XAML document and completely remove the hard coded width and let the text auto-flow. The neutral language now auto-flows text and adjusts in size while the localized version still has the hard-coded width in place and you effectively end up with two different user interface behaviors. Unless you explicitly update any changes made in the localized CSV file it's quite easy to get the base document and the localized content out of sync.

It's a good idea to minimize the use of non-text property values in localization to avoid differing behaviors for different cultures. It's usually better to fix the underlying default culture's document so that it works with all cultures than trying to 'fix' a specific culture.

Localizing Resources in the CSV File

Once you've exported resources using LocBaml you can copy the CSV file to localize to another culture. Typically you export to something like en-US.csv and then copy the file with a name of the culture of your choice like de.csv. At this point you can make changes to the resource content in the CSV file to localize for the neutral German culture.

Do not add or change Resource Ids in the Localization CSV File

Keep in mind that you are effectively working with an export of compiled resources in the CSV file. LocBaml exports all resources from the neutral assembly. When merging back LocBaml looks at your resources and matches them to the resources of the neutral assembly. If any resources are missing in your CSV file the neutral resources are used. However if resource keys exist in the CSV file that don't exist in the neutral resource's BAML an error occurs. So be very careful not to add or inadvertently change keys. Also make sure that if you change a x:Uid in your XAML pages that you update the localized CSV file(s) to match this updated x:Uid.

In short: Missing Uids are non-fatal, but new Uids break LocBaml's generate command.

It's best to ensure your application is complete and ready for localization before you start localization with LocBaml. Try to minimize addition of new resource keys in your layouts after you've started localization. Ultimately if you have to update your application and localized content some tooling is required in order to be able to sync without starting over. There is a partial solution to this problem as well as some tips on automating the build process covered in this CodeProject article: <http://www.codeproject.com/KB/WPF/LocBamlClickOnce.aspx>

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

The process of localization using the CSV format even with a decent CSV editor is very tedious. It seems that LocBaml was designed with additional tooling in mind, but this tooling has not arrived to date. Currently the LocBaml export is one-way so if you export resources a second time there's no way to sync up an already localized CSV file. A second export simply writes out a brand new file of the neutral resources and it's your task to sync the new export to your existing localized resources.

It should be clear by now that using LocBaml is best done once your application is complete because it's very difficult to update already localized content in a CSV file – especially if new keys need to be merged or resource ids have changed. As much as possible, make sure your app's UI is complete before you start if you use LocBaml! If you have a highly dynamic application that changes drastically after localization then LocBaml is probably not the right approach for you unless you are willing to build synchronization tools yourself.

Embedding Localized Resources back into Satellite Assemblies

Once you've localized your resources in the CSV file for one or more specific cultures the next step is to merge those resources into a satellite resource assembly. LocBaml has a */generate* switch that creates a compiled .resources file or a full resource satellite assembly from a localized CSV file. It looks at a root assembly (such as the en-US satellite assembly discussed earlier) and compares *x:Uid* values against those found in the CSV file. It writes out any keys found in the CSV or uses the values from the base assembly if keys are missing in the CSV.

The process of merging the CSV resources depends on whether you only have BAML resources or a mix of both BAML and Resx resources. If you only have BAML resources LocBaml can generate a final satellite assembly directly. If you're mixing BAML and Resx LocBaml can create a .resources file that has to be separately combined and linked with the Resx resources using the .NET Framework assembly linker. Needless to say the former is the easier process.

Start by compiling your application so that the neutral culture satellite assembly is created. Assuming you've created a CSV file with localized resources let's start with the BAML resources only scenario which uses **LocBaml /generate** to create a resource assembly directly. In the following examples you can assume that the default language resource is en-US and that we're localizing to the neutral German culture (de).

To generate the German resource assembly from your localized CSV you can use this command line:

```
LocBaml.exe /generate en-US/WpfLocalizationLocBaml.resources.dll  
/trans:..\..\Res\de.csv /out:de /culture:de
```

This generates a de\WpfLocalizationLocBaml.resources.dll assembly with the localized BAML resources.

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

Remember LocBaml should be run from the folder where main assembly exists which typically will be bin\Debug or bin\Release. You specify the default resource assembly and your localized CSV file which in this example lives in the Res\de.csv folder beneath the project root. You also specify an output folder matching the correct satellite assembly path (in this case bin\Debug\de) and the culture that is set for the assembly.

The default resource assembly always holds all resources available and acts as the resource baseline so that any missing resources in the localized CSV file are still embedded into the BAML document using the default values. The merge process creates the de\WpfLocalizationBaml.resources.dll satellite assembly that contains a single resource set with all BAML resources contained within it.

Astute readers probably noticed that this process sidesteps standard resource fallback as it goes directly to the neutral resources if an individual resource is not found. One way to get around this problem is to export and generate specific cultures (de-DE or fr-CA) off their neutral cultures instead of the default culture. So when you generate de-AT resources you use the de\WpfLocalizationLocBaml.resources.dll as your template assembly for LocBaml when exporting and generating. This way resource fallback picks up the *de* specific resources when resources are missing in de-AT. But this requires that the parent culture is already localized and that resource compilation is done in a specific order. It's another step that complicates the LocBaml process.

If you have multiple cultures you are localizing for, you need to repeat this process for each of them.

BAML and Resx Resources Combined

If you have both XAML and Resx resources the process is even more complex. When compiling directly to a satellite assembly LocBaml overwrites the assembly if one already exists. Unfortunately, if you have Resx resources in your project a *de\WpfLocalizationLocBaml.resources.dll* exists already and contains those compiled Resx resources. Running **LocBaml.exe /generate** and creating a satellite assembly overwrites the existing assembly and you lose the Resx resources in the process. Not cool.

In order to get Resx and XAML resources to co-exist you have manually link the compiled BAML and Resx resources using the assembly linker (al.exe). LocBaml creates .resources files which are raw binary resource data files and merges them with any .resource files that the project created in the intermediate \obj folder. The process to build a generic German (de) satellite assembly with both BAML and Resx resources looks something like this in a batch file (both commands on one line each):

```
LocBaml.exe /generate ..\..\obj\WpfLocalizationLocBaml.g.de.resources
           /trans:..\..\Res\de.csv
           /out:de /culture:de

al /template:WpfLocalizationLocBaml.exe
  /embed:de\WpfLocalizationLocBaml.g.de.resources
  /embed:..\obj\WpfLocalization.Properties.Resources.de.resources
  /culture:de /out:de\WpfLocalizationLocBaml.resources.dll
```

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

Here LocBaml generates a .resources file rather than a final assembly. You then run the Assembly linker (part of the Windows SDK in C:\Program Files\Microsoft SDKs\Windows\v6.0A\bin) to combine the generated BAML .resources file plus any Resx resources that your project might be using. Resx resources are generated into the intermediate *obj* folder of your project and you can pick them up by the appropriate filename and culture plus resource extension. Your project's Resources.de.resx file has the full resource path filename of WpfLocalizationLocBaml.Properties.Resources.de.resources where the path is the default project namespace, plus the folder path to the resource, plus the culture all separated by dots plus a .resources extension. If you're not sure of the exact name check the \obj folder and look for any .resources files in your culture. Unfortunately the /embed flag doesn't allow for wildcards so if you have more than one Resx resource file in your project you have to add it manually with additional /embed switches.

Are we having fun yet? Remember you have to repeat this exercise for each culture you're localizing for – what's shown here is for a single localized culture.

To automate this process a little bit you might want to use a batch file. Figure 15 shows a semi-generic batch file that you can call with culture and filename parameters from the command window or a build task.

Figure 15: A semi-generic batch file that demonstrates the steps to generate both Resx and BAML resources for a single culture

```
REM  Template for embedding BAML and RESX resources into localized assembly
REM  Pass:      a locale (de) and main file name (WpfLocalization)
REM  ASSUME:    additional resource files beyond Properties.Resources.Resx
REM            add them to al.exe command line
CLS
echo off

REM Parm1 is culture to translate to
REM Assumes: RES folder with de-DECSV file
REM Assumes: Global Resources for de-DE exist
set culture=de
if NOT "%1" == "" (set culture=%1)
echo Culture: %culture%

set filename=WpfLocalizationLocBaml
if NOT "%2" == "" (set filename=%2)
echo Filename: %filename%

REM object path for compiled raw .resource files
set objPath=..\..\obj\debug
set objrelPath=..\...\obj\release
```

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

```
REM Generate raw .resource file in de-DE path so we can manually merge it
with
```

```
LocBaml.exe /generate %objPath%\%filename%.g.en-US.resources
/trans:..\..\Res\%culture%.CSV /out:%culture% /culture:%culture%
```

```
REM Combine resource files
```

```
al /template:"%filename%.exe"
/embed:%culture%\%filename%.g.%culture%.resources
/embed:%objPath%\%filename%.Properties.Resources.%culture%.resources
/culture:%culture% /out:%culture%\%filename%.resources.dll
```

```
REM remove the intermediate resource file
```

```
REM del "%culture%\%filename%.g.%culture%.resources"
```

```
pause
```

You can add this as a post build task in your project for each culture you have localized:

```
$(TargetDir)LocBaml_CreateResourceAssembly_MixedBamlResx.bat de
$(TargetDir)LocBaml_CreateResourceAssembly_MixedBamlResx.bat fr
```

This batch file makes a number of assumptions:

- It must run from the target directory (\Debug or \Release)
- LocBaml.exe has to exist in the same folder
- Localized CSV files are named *culture.csv* (i.e., de.csv) and stored in a \Res folder
- It assumes the default language is en-US (change for your chosen default culture)
- It assumes the Assembly Linker (al.exe) is in your environment path
- You've added any Resx resources manually with /embed commands on al.exe

The batch file described above is a quick and dirty solution that can act as a template, but it does require some slight modifications for your particular environment. The file along with LocBaml.exe should be copied to both \Debug and \Release folders.

Using an MSBuild Task

The batch file discussed in the previous section makes it easier to see what's going on in the process of generating the localized BAML resources and merging the Resx resources. Another approach is to use an MSBuild task. It's a little more work to set up but it's more integrated and reliable as you can tap into the build process at the time that the output satellite assemblies are built.

The same CodeProject article mentioned earlier (<http://www.codeproject.com/KB/WPF/LocBamlClickOnce.aspx>) shows how to perform this task along with some other useful tasks like a tool that can sync up exported CSV results with

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

previously localized CSV content. Concepts in this article were used to create a similar build script.

Figure 16 illustrates a somewhat simpler build script that handles only LocBaml generation and satellite assembly merging.

Figure 16: Using an MsBuild Target to run LocBaml for generating satellite assemblies

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <!-- Adds the build action 'LocBamlCsv' -->
  <ItemGroup>
    <AvailableItemName Include="LocBamlCsv"/>
  </ItemGroup>

  <Target Name="AfterBuild"
    DependsOnTargets="$(CreateSatelliteAssembliesDependsOn)" >

    <!-- Locbaml needs the runtime assemblies in the output dir -->
    <Copy SourceFiles="$(ProjectDir)..\Tools\LocBaml.exe"
      DestinationFolder="$(OutputPath)" />

    <!-- generate a .resources file for .csv merged output -->
    <Exec Command="LocBaml /generate
      ..\..\$(IntermediateOutputPath)$(TargetName).g.$(UICulture).resources
      /trans:%(LocBamlCsv.FullPath)
      /out:..\..\$(IntermediateOutputPath)
      /cul:%(LocBamlCsv.Culture) "
      WorkingDirectory="$(OutputPath)"
      ContinueOnError="true"
    />

    <Exec Command="al /template:$(TargetName).exe
      /culture:%(LocBamlCsv.Culture)
      /out:..\..\$(OutputPath)%(LocBamlCsv.Culture)\$(TargetName).resources.dll
      /embed:$(TargetName).g.%(LocBamlCsv.Culture).resources
      /embed:$(TargetName).Properties.Resources.%(LocBamlCsv.Culture).resources"
      WorkingDirectory="$(IntermediateOutputPath)"
      ContinueOnError="true"
    />

  </Target>
</Project>
```

To use this build target the following setup is required:

- LocBaml.exe must exist in a `\Tools` folder at the same folder level as your project
- Localized CSV files must reside in a `\Res` folder below the project root. Each of the cultures .csv files should be named with culture identifier plus the .csv extension (i.e., de.csv).

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

- Add the build target into your Project file after existing targets:
`<Import Project="$(ProjectDir)LocBamlCsv.Target.xml" />`
- Add an Item entry for each culture that you want to export and merge:
`<LocBamlCsv Include="Res\de.csv">
 <Culture>de</Culture>
</LocBamlCsv>`
- You have to add any additional Resx files manually in al.exe command line
`/embed:$ (TargetName) .MyResources.%(LocBamlCsv.Culture) .resources`

You need to manually edit the project file and the LocBamlCsv.Target.xml file to do this. You can check out the WpfLocalizationLocBaml project to see how this is set up in a completed project.

The build task copies LocBaml.exe into the target output folder (bin\Debug or bin\Release) and runs LocBaml from there. It then creates the output resources into the intermediate (obj\Debug or Release) folder. The assembly linker is then invoked to build the final satellite resource assembly that contains both the localized Resx and BAML resources.

LocBaml – Not for the faint of heart

The LocBaml localization approach is not for the faint of heart. From the fact that you have to download a separate and unsupported tool from Microsoft, compile it, copy it to your output folders and then manually run several command line tools and integrate them into the build process is daunting especially if you're just starting out with localization. Localization of CSV files also is very finicky in that files can easily be corrupted and become unusable. Make sure you make frequent backups that you can roll back to while localizing.

Here's a quick summary of the strengths and weaknesses of using LocBaml.

The pros of LocBaml are:

- **No special requirements for adding localization information to pages**
For developers BAML Localization is great because there's no special markup or resource mapping syntax to embed into the document. Developers can simply layout pages in their default culture and not worry about localization issues for each page. This is easily the biggest plus of LocBaml localization as it almost completely separates the localization process from the development process.
- **Efficient at runtime**
Localized BAML loading is very efficient at runtime because WPF loads each BAML document as a whole entity without looking up and binding each individual value.
- **Fairly easy localization**
Whether you use a CSV editor or a plain old text editor to edit your markup text the process of localization can be relatively quick because you are changing text in a text document and you can quickly march down the list of localizable values and localize them.

The cons of LocBaml are:

- **Unsupported tool**

The primary tool for BAML localization is LocBaml and this tool is a *sample application* that has existed since WPF was released and hasn't been updated since. There's been no news on whether this tool will see improvements in the future or whether additional tools will be provided to facilitate resource editing and syncing.

- **CSV file format exports**

LocBaml exports CSV files, which is a difficult format to work with for direct editing. Localizing CSV files isn't really realistic so most developers/localizers likely are faced with some sort of conversion of the CSV files into some other format more suitable for editing. The CSV format is also volatile – one misplaced comma and your file is broken. The LocBaml output format is also in UTF-8 encoded which many CSV tools (including Excel) do not maintain when saving. Another step of opening in Visual Studio and applying custom encoding is required.

- **LocBaml is a one-way export**

LocBaml always creates new resource files when you export. So if you run LocBaml once and translate say English to German, then update your application and change or add new elements there's no easy way to sync these changes to the already translated CSV file you created. You have to manually keep track of all updates and map them into the translated CSV file or else start over.

- **CSV files contain lots of extraneous layout Information**

In addition to localizable strings the CSV export includes many layout properties from the XAML document in the localization document. While it offers some flexibility to make UI changes in the localized CSV document, it's also very difficult to sync these changes back up when changes are made in the default culture XAML application. Even if no values are overridden it's possible that a change made in the neutral culture in XAML won't show up in a localized version.

- **Satellite assembly requirements don't play well with Resx resources**

Localized BAML resources are required to run out of satellite assemblies when localization is enabled, even for the default culture. If you mix BAML and Resx resources this requirement forces that you have both a culture specific and invariant neutral resource Resx file (i.e., Resources.resx and Resources.en-US.resx with the same content) in order to get both strongly-typed resources and Resx resources that work with the application at runtime. Additionally the LocBaml export requires merging of Resx and BAML resources using a separate build step.

Localizing Resx Resources

LocBaml is a new and largely unproven approach to localization. Luckily WPF is not limited to this method of localization and you can still use traditional Resx resources in your application, albeit with less integration than you might know from other technologies like Windows Forms or ASP.NET.

As discussed earlier in this whitepaper, Resx resources are a core feature of the .NET Framework and are supported for all project types. Resx resources can be easily accessed in code and in XAML documents via bindings but the process of assigning and using them is quite different from the LocBaml approach involving explicit mapping of resources to properties.

Binding to Strongly-Typed Resx Resources

Strongly-typed resources are an integrated feature of the Visual Studio environment. As discussed earlier strongly-typed resources are created automatically when a new Resx file is added to a project automatically creating a class with the same name as the resource file. In code you can simply access strongly-typed resources as static properties on generated the generated resource class. For example:

```
MessageBox.Show(WpfLocalization.Properties.Resources.HelloWorld);
```

This is useful, but in WPF applications you probably want to declaratively bind the value to XAML elements instead of using code to assign it. Based on the same resources shown above, you can easily bind to the static Resources object using the *StaticExtension* markup extension. Binding with the StaticExtension involves two steps:

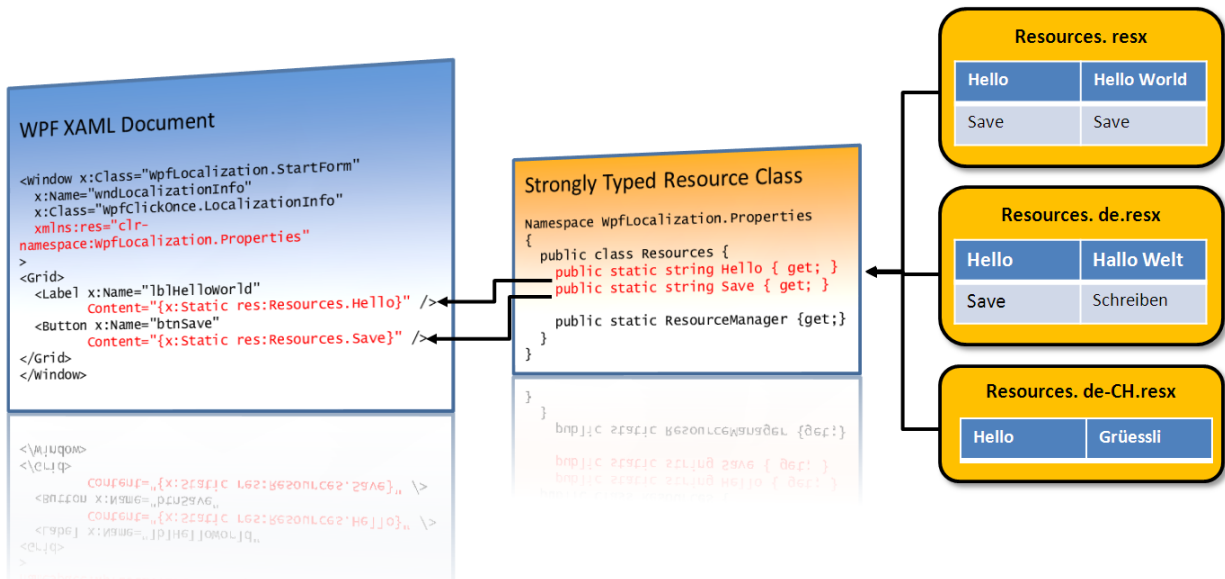
- Registering the namespace/assembly of your resources in the XAML document
`xmlns:props="clr-namespace:WpfLocalization.Properties"`
- Binding the resources to a property using the *StaticExtension*
`<Label x:Name="lblResxHelloWorldStrongValue"
Content="{x:Static props:Resources.HelloWorld}" />`

In this scenario you are binding to the static properties of a strongly-typed object like `WpfLocaliztion.Properties.Resources`, which matches the resource keys in the Resx resource file. In Figure 17 the current `UICulture` is matches the default culture (en-US in this case) and so resources are pulled from the default resource set. If the current `UICulture` were Swiss German (de-CH) the Hello key would be filled from the Swiss resources, while the Save key would be filled from the neutral German resources via resource fallback. If the German resources didn't exist resource fallback would fall back to the default resources and load the Save key from there.

Figure 17: Binding to strongly-typed Resx properties using `x:Static` bindings in XAML

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009



The `x:Static` binding points at the strongly-typed resource object and its static properties - in this case `WpfLocalization.Properties.Resource`. This is even type safe as the compiler catches any invalid type references in the markup extension during XAML compilation:

```
<Label x:Name="lblResxHelloWorldStrongValue"
  Content="{x:Static props:Resources.HelloWorld}" />
```

It's also possible to bind to non-string values – although it's a bit of a hassle to actually enter non-string resources into Resx files. There's no designer support for anything but strings and a few known image types like Bitmaps, Icons and generic streams. To create non-string resource values you have to do it manually in the Resx file rather than through the resource editor in Visual Studio. For example, to add a value for a width you can store a Double value in the Resx file as XML:

```
<data name="lblHelloWorld.Width" type="System.Double">
  <value>20</value>
</data>
```

This generates a property in the strongly-typed resource class of type Double:

```
public static double lblHelloWorld_Width {
  get {
    object obj = ResourceManager.GetObject("lblHelloWorld.Width",
      resourceCulture);
    return ((double) (obj));
  }
}
```

You can then bind the resource entry in WPF with:

```
<Label Content="{x:Static props:Resources.Today}"
        Width="{x:Static props:Resources.lblHelloWorld_Width}"
/>
```

The key is that the type of the value you bind **has to** match the type of the property you are binding to. The `StaticExtension` binds directly to the underlying property and there's no support for a type converter. In the previous example a *Double* value is stored in Resx and the strongly-typed property to bind to is the `Width` property on the `Label` element. This seems easy enough for simple types like *Double*, but becomes difficult for anything more complex like a *Thickness* object used for *Margin* and *Padding* for example. Standard string conversions that XAML uses don't work with XML serialization. In other words you can't assign a string value of 10 and expect the margin to be adjusted from resources. You get a compile time error for mismatched types.

Due to this complexity, accessing non-string values with `x:Static` and strongly-typed resources is not recommended, but you can work around this easily with the custom markup extension described in the next section.

Organizing Static Resx Resources

If you have anything but a trivial WPF application you are likely to end up with a large number of resources in your Resx files. You'll want to think carefully about organizing these resources in a way that lets you find resources to edit and access easily. If you've used localization in Windows Forms and ASP.NET you are probably familiar with the concept of local and global resources, where there's one set of global resources typically in `Resources.resx` and then one set of local resources per document in the application.

As mentioned earlier, WPF doesn't provide any built-in automatic resource mapping mechanism unlike either Windows Forms or ASP.NET and so it's up to you to decide how to store and map resources. It is a good idea to follow the global/local resources model even though it's not directly supported in the IDE as it provides a proven organization scheme to deal with large numbers of resources.

Using this approach you create a local Resx file for each XAML document that holds the resource keys that are specific to that document. This should include resources for immediate element property assignments as well as messages that are specific to the individual document.

To summarize, creating 'local' resources involves:

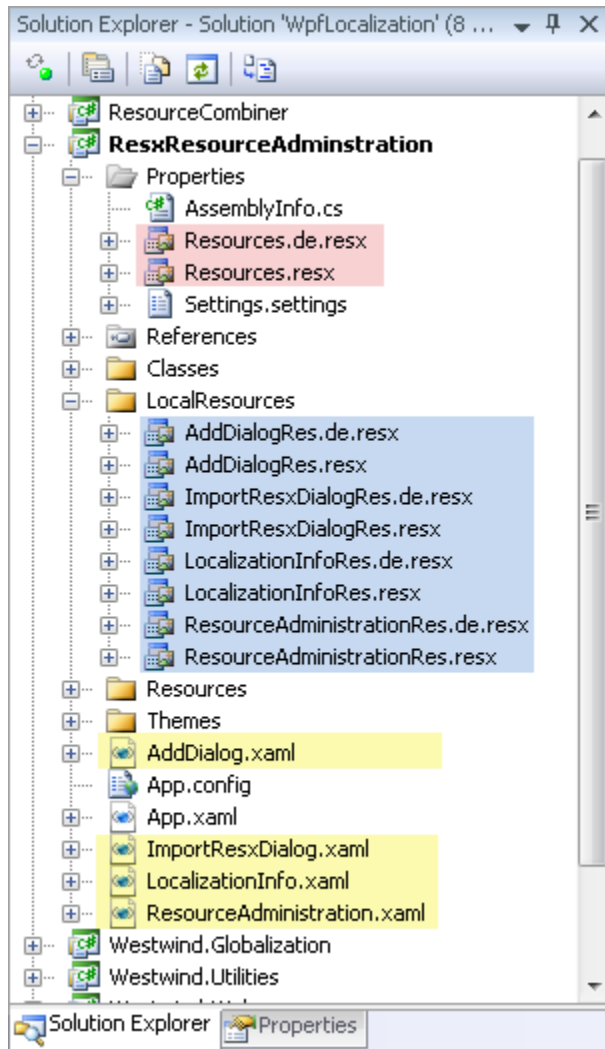
- Creating a Resx file for each XAML document
- Adding the namespace of the resources type to the XAML document
- Binding individual resources using this namespace

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

For example, the following example includes a document called `ResxResourceAdministration.xaml` and a resource file in `LocalResources\ResxResourceAdministrationRes.resx` plus a German version of the resources. Figure 18 illustrates the project layout when multiple windows are present:

Figure 18: Using 'global' resources (red) and 'local' resources (blue) related to Windows (yellow)



Separating out the resources per document keeps the number of resources in each resource set manageable – much more than one massive resource file that contains all resources. Using the *Res* postfix for each local resource filename, matching it with the name of the document they are related to, avoids name collisions so that the resources and document don't generate the same class name. This ensures that you can access the strongly-typed resources without specifying a namespace in code and – if you happen to create the resource file in the same folder as the document – that Visual Studio doesn't

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

generate exactly the same class name for document and resources. It also helps to place local resources into a separate folder to remove the clutter that multiple resource files introduce to the project tree.

When binding to resources in a document, here's what the markup looks like for a few labels bound to strongly-typed resources from 'local' and 'global' resource strings:

```
<Window x:Class="ResxResourceAdministration.LocalizationInfo"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:props="clr-namespace:ResxResourceAdministration.Properties"
    xmlns:res="clr-namespace:ResxResourceAdministration.LocalResources"
    Title="AddDialog" Height="300" Width="300">
    <Window.Resources>
    <StackPanel Margin="10">
        <Label Content="{x:Static res:ResxResources.lblHelloWorld.Text}"
            x:Name="lblHelloWorld" Margin="5" />
        <Label Content="{x:Static props:Resources.Today}" Margin="5"/>
    </StackPanel>
</Window>
```

All of this is optional though. The above are simply suggestions to give you some ideas for organizing your resources in your projects. Feel free to lay out your resource files any way that makes sense to you and your localization work flow.

Things to Consider with x:Static Resource Bindings

Using static resource bindings is easy to work with and it's completely native to WPF. It works well if you stick to the default Resx resource store and you are mostly localizing strings. String localization is the most common scenario and if that's all you need to do *x:Static* bindings to Resx resources is good enough.

Here's a summary of pros and cons – and don't let the shortness of the pro list deter you – part of the benefit of this mechanism is its simplicity.

The pros are:

- **x:Static is built-in and easy to use**
This is one of the most compelling reasons to use this approach. There are no extra tools or special markup syntax rules to use, there's nothing to configure, you can just add a namespace to your resource class and start assigning resource bindings and you're ready to go. The syntax is straight forward and it's efficient since the bindings are static.
- **x:Static bindings are checked by the compiler**
x:Static expressions are evaluated at compile time, so if you've specified an invalid value, the compiler catches the error and displays an error pointing into the XAML element that holds the expression.

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

- **Resx resources are easy to manage**

Resx resources are well supported in Visual Studio and easy to edit. There are also a host of third party tools out there that facilitate localization for applications using Resx resources and this infrastructure can be leveraged in WPF as well.

The cons are:

- **No type converter support**

x:Static bindings are direct bindings and so the value you retrieve is the value bound without any type conversion kicking in. This means your data stored in the Resx file has to match the specific type of the property you are binding to *exactly*. There's no easy binding to complex types like Thickness, Brushes or other UI structures. You can effectively only bind to strings easily. Binding to anything else is a hassle.

- **Manual mapping of localizable properties to resource keys**

Because there's no design-time support for resource synchronization, it's entirely up to you to map resources to element properties and apply the correct binding expressions. Unless you're diligent during the development process it is very easy to miss resources that need localization and the hookup process can be time consuming and tedious.

- **Bindings have to be added in XAML**

Bindings are a non-designer feature in the XAML designers and have to be entered in XAML markup. This means you have to switch to XAML view to hookup markup extension bindings while working in the designer.

- **Dependency properties bindings only**

XAML binding is supported only on dependency properties so if you need to set localized value on non-dependency properties any binding approach is not going to work. This shouldn't be a big problem since all of the key properties you might want to bind to are dependency properties. Potential exceptions are third-party controls or your own custom controls that don't implement dependency properties for UI elements.

- **Design-time problems if using satellite assemblies**

If you mix Resx and BAML resources you are forced to store localized resources in satellite assemblies including resources for the default culture. Microsoft Blend is unable to load resources from satellite assemblies and so the designer fails to render the element with the binding.

- **Bound to the strongly-typed Resx resources**

The bindings described here are bound to the strongly-typed classes generated by Visual Studio and these classes hard code the *ResourceManager* that is used. This means you are bound to using the stock Resx resource manager and you are tied to the strong-types it provides. If you want to access resources from a different resource store like a database or other custom storage you can't since there's no easy way to override the *ResourceManager* on the strongly-typed class.

None of the cons are deal breakers since the vast majority of localization occurs with string values. As to other types of layout properties WPF largely removes the need for these by providing a more fluent layout paradigm via auto-sizing that can provide pleasing results that work for all cultures rather than a custom fitted solution for each culture. It is customary to try to achieve a single code base and layout for all cultures where possible to keep localization manageable.

If you're interested only in strings, you can use strongly-typed resources and you don't need to switch cultures dynamically – *x:Static* binding to strongly-typed resources are the way to go.

Custom Markup Extensions for Resx Resources

If you want more control than the *x:Static* binding syntax provides you can take binding into your own hands and create a custom markup extension that provides additional functionality.

A few things that you can do with custom markup extensions not possible with the *StaticExtension*:

- Use *either* a custom *ResourceManager* or strongly-typed resources
- Handle type conversions so you can use string values with non-string properties
- Provide default values for binding failures
- Use format strings
- Dynamically detect culture changes and update the user interface at runtime

The *WpfControls* sample project included along with this whitepaper provides a *ResExtension* markup extension that implements the above features. This is a custom implementation to demonstrate the functionality of a custom markup extension for resource binding, but understand that this is only one example for how you might perform this task. If you search online you can find dozens of different markup extensions for WPF localization available so if the extension provided here doesn't strike your fancy there are other options available to you. Regardless, you will find that all markup extensions for *Resx* usage share a few common principles to be covered here with the *ResExtension* provided.

Because a custom markup extension is still a markup extension like the *x:Static* extension much of the process is the same: You still have to manually map resources to individual properties and assign the markup syntax to each element property in XAML.

To embed a localized value into the page using the *ResExtension* you first have to include a namespace and assembly:

- Add the *WpfControls* project or assembly to your project's references

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

- Add the `xmlns:res=http://www.west-wind.com/WpfControls` namespace to your document header (you can use any namespace prefix you like), here “res” is used

Once configured you can use any of the following markup syntax versions crammed into this single Label element:

```
<Label x:Name="lblResxHelloWorldMarkupExtension1Value"
      Content="{res:Res Id=HelloWorld,Default=Hello#}"
      Margin="{res:Res Id=HelloWorldMargin,Default=10}"
      Width="{res:Res Id=HelloWorldWidth,
                ResourceSet=WpfClickOnce.MyFormRes,
                Default=50}" />
```

This markup extension constructs a ResourceManager and uses the Id property as the resource key to look up. The extension also supports direct access to strongly-typed resource objects in much the same way as `x:Static` does but with post-processing provided by the extension so you can have a type conversion. Using strongly-typed resources the syntax uses the *Static* attribute instead of *Id* to identify the resource to bind to. Here is the same label using Static syntax:

```
<Window
...
  xmlns:res="http://www.west-wind.com/WpfControls"
  xmlns:props="clr-namespace:WpfClickOnce.Properties"
  xmlns:local="clr-namespace:WpfClickOnce"
...
>
<Label x:Name="lblResxHelloWorldMarkupExtension1Value"
      Content="{res:Res Static=props:Resources.HelloWorld,Default=Hello#}"
      Margin="{res:Res Static=props:Resources.HelloWorldMargin,Default=10}"
      Width="{res:Res Static=local:MyFormRes.HelloWorldWidth,
                Default=50}" />
```

If you’re using Resx resources and you have strongly-typed classes available the latter approach is preferable because it eliminates the need to include assemblies and resource sets as part of the binding. The former approach is useful if you don’t want to be tied to the default ResourceManager, for example to support elements presenting a culture other than the current `UICulture`.

The ResExtension supports a host of parameters shown in Figure 19 although only Static or Id are required.

Figure 19: The parameters for the ResExtension markup extension

Markup Parameter	
Id	The resource key that identifies the resource to load using ResourceManager.

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

Static	Allows you to point at a strongly-typed resource using the same syntax as <code>StaticExtension</code> . Uses the strongly-typed resource to retrieve the resource value, but post-processing for string conversions (for example) are still applied. When using this syntax <code>ResourceSet</code> and <code>AssemblyName</code> are ignored. <i>Example: <code>Static=properties:Resources.Helloworld</code></i>
ResourceSet	The name of a resource set to bind to. This is the name of the compiled resource in the assembly which is the default namespace, plus path plus the name of the Resx file separated by periods (.). If omitted the default resource set you provided is used. <i>Example: <code>ResourceSet=WpfLocalization.Properties.Resources</code></i>
Assembly	The name of an assembly or id of a preloaded assembly to allow binding to resources in a specific assembly. Needed only if you have multiple projects that are using the markup extension. If not specified the startup assembly or the <code>LocalizationSettings.DefaultResourceAssembly</code> is used. <i>Example: <code>Assembly=MyControlLibrary</code></i>
Default	A default value displayed if the value cannot be found or a binding error occurs. This value is also used in the designer when the markup extension cannot find the resource or resource set. You should always provide this value to avoid design or runtime errors.
Format	A format string that the converted value is embedded into.
Converter	An explicit type converter to use to convert the value. If not specified the default type converter for the property is used.

`ResourceSet` and possibly `Assembly` are required only when using `Id` binding. `Id` binding effectively creates its own `ResourceManager` instances and caches them. In order to find the right `ResourceManager` to load a `ResourceSet` and `Assembly` is required. A `ResourceSet` needs to be specified if you need to bind to anything but the default resources using an `Id`. When using `Static` binding `ResourceSet` and `Assembly` are ignored since the strongly-typed resource class referenced contains a `ResourceManager` instance to access.

The `ResExtension` also deals with binding failures by displaying *Default* values rather than failing. It's not uncommon for applications to run into problems with resources at runtime which are often hard to catch at compile time or even during user interface testing. Rather than failing the extension captures errors and displays a default value instead. If you prefer you can also make it so that default values show with some sort of delimiter to indicate that the resource was not loaded correctly but at least your application won't break because of it. Default values are also crucial for the Visual Studio and Blend designers which may not always have access to resources and rather than failing display the default value. This is especially true for Blend which cannot display resources from satellite assemblies for example.

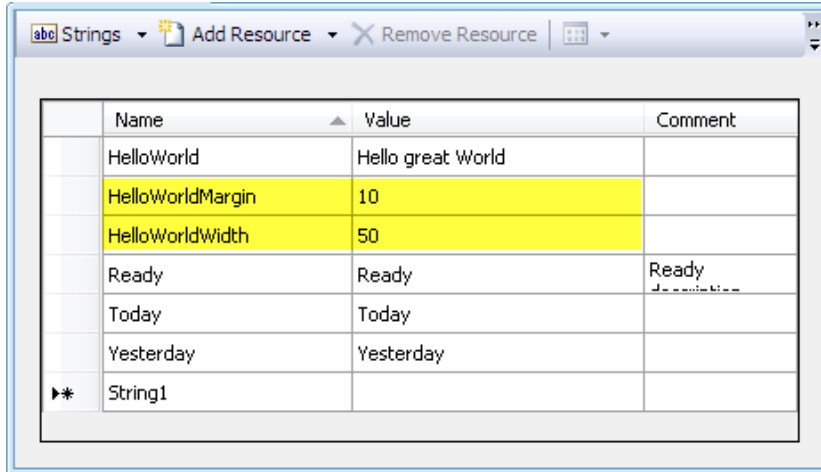
Another big benefit to this extension is that you can use resource string values for non-string properties in the same way you use them in XAML attributes. Because the markup extension applies a

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

ValueConverter the strings are converted into the corresponding XAML property types. In the example above Margin and Width are bound, and these are non-string properties (*Thickness* and *Double* respectively), but are stored in the resource file as simple strings (10 and 50 respectively as shown in Figure 20).

Figure 20: Custom extensions can store string values for non-string properties in the Resx file by using a ValueConverter to convert values automatically when binding



	Name	Value	Comment
	HelloWorld	Hello great World	
	HelloWorldMargin	10	
	HelloWorldWidth	50	
	Ready	Ready	Ready description
	Today	Today	
	Yesterday	Yesterday	
▶*	String1		

Watch out for Separator Characters in Property Values

You probably know that certain properties like Margin and Padding allow you to specify multiple values separated by a separator character. For example you specify each border's margin with `Margin="10,5,10,10"` where comma is the separator character. What you may not know is that these separators vary for different cultures. While the value shown in the last line works in en-US, it does not in de-DE because the comma is treated like a decimal point in the Double Margin values in German. If you apply 10,5,10,10 in the default culture and don't specifically localize the German version to 10;5;10;10 you'll get a runtime error.

Try to avoid separators as much as possible when using things like Margin and Padding in localized values. If you do use them make sure to check each culture carefully at runtime especially when resource fallback occurs.

The *Assembly* attribute is required only if you use the markup extension in more than one assembly in your application – for example a control library project *and* your main application. Because the markup extension is loaded without any specific context it doesn't know implicitly where to load resources from when using Id binding.

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

For this reason there's a static *LocalizationSettings* class that is used to provide the location of default resources and also a default *ResourceManager* if no *ResourceSet* is provided. The markup extension includes logic that finds the default resource assembly and *ResourceManager* automatically in most standalone WPF applications so no configuration is required, but you can also manually configure these settings at application startup:

```
public App()
{
    InitializeComponent();

    // Explicitly initialize the resource manager
    LocalizationSettings.Initialize(this.GetType().Assembly,
        WpfLocalization.Properties.Resources.ResourceManager);
    LocalizationSettings.AddAssembly("WpfControls",
        typeof(ResExtension).Assembly);
}
```

Note that the Visual Studio and Blend designers do not run this code, so if you use non-standard locations for resources the designer will not display these resources and fall back to default values or to displaying resource keys. This is where those default values come in handy – although your resource store has changed the designer can still display the default values.

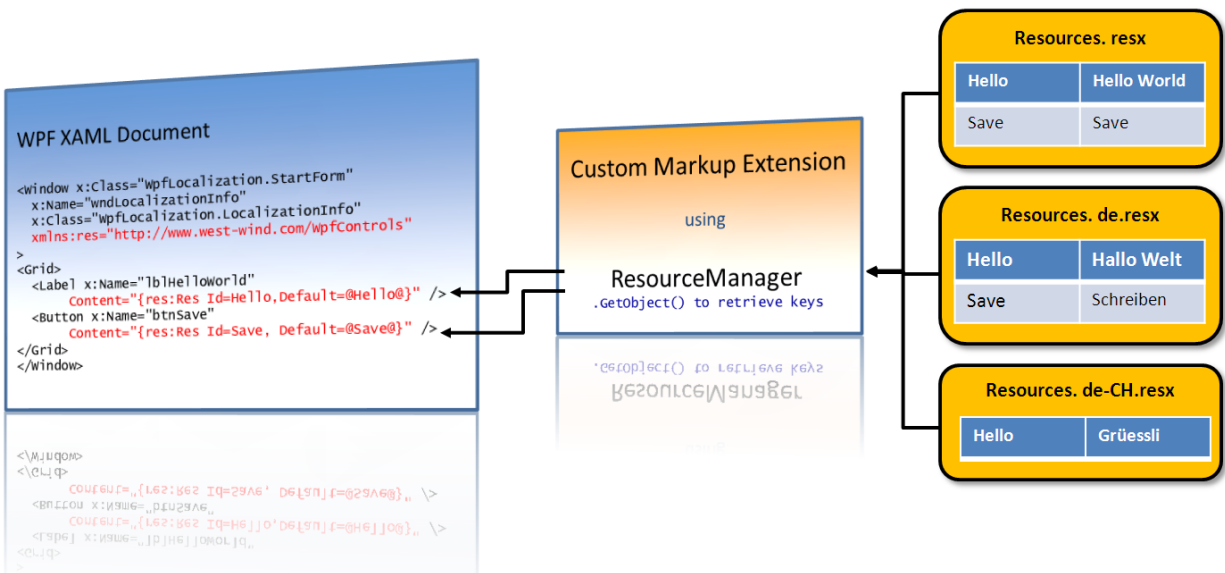
By default in the designer the *ResExtension* attempts to use the resources available in the main WPF application only. Internally it has to walk the loaded assemblies and try to find the Application instance. Designer support for resources with custom markup extensions is a thorny issue and you can examine the logic to retrieve the default resource assembly in the *ResExtension.FindDefaultResourceAssembly* method.

Because of the designer volatility especially in Blend the default value is once again quite crucial. You should always specify a default value so you're guaranteed to see an expected value displayed in the designer.

How the Custom Markup Extension Works

The implementation of the *ResExtension* is straight forward. Behind the scenes it uses one or more *ResourceManager* instances to retrieve resources directly and feed them back to the XAML document via bindings using the markup extension's syntax. Once a resource value has been retrieved type conversion is applied to match the string value retrieved to the type of the property the markup extension is binding to. The assumption with the markup extension is that all values are string values that are sent through the value converter of the property that is bound to if available. The entire process is wrapped into an error handling block that captures any binding errors and falls back to display the Default value if an error occurs. Errors are also written to the Trace output which you can monitor in Visual Studio using the Output Window when running in Debug mode.

Figure 21: The ResExtension custom extension uses a ResourceManager instance directly to retrieve resources and feeds them to the XAML document



It's beyond of this article to describe the entire markup extension here, but to give you an idea of what's involved in a markup extension and how ResExtension works in providing a value Figure 22 shows the `ProvideValueInternal` function (simplified) that does most of the work.

Figure 22: The key `ProvideInternal` method of the ResExtension markup extension is the core method that handles resource retrieval, formatting and if necessary error handling

```
private object ProvideValueInternal()
{
    object localized = null;

    if (Static == null)
    {
        // Get a new or cached resource manager for this resource set
        ResourceManager resMan = this.GetResourceManager(this.ResourceSet);

        // Get the localized value
        if (resMan != null)
            localized = resMan.GetObject(this.Id);
    }
    else
    {
        try
        {
            // Parse Static=properties:Resources.HelloWorld
            int index = this.Static.IndexOf('.');
            if (index == -1)
```

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

```
        throw new ArgumentException();

        // resolve properties:Resources
        string typeName = this.Static.Substring(0, index);
        IXamlTypeResolver service =
            _serviceProvider.GetService(typeof(IXamlTypeResolver))
            as IXamlTypeResolver;
        Type memberType = service.Resolve(typeName);

        string propName = this.Static.Substring(index + 1);
        localized = memberType.GetProperty(propName,
            BindingFlags.Public | BindingFlags.Static |
            BindingFlags.FlattenHierarchy)
            .GetValue(memberType, null);
    }
    catch
    { /* ignore retrieval errors */ }
}

// If the value is null, use the Default value if available
if (localized == null && this.Default != null)
    localized = this.Default;

// fail type conversions silently and write to trace output
try
{
    // Convert if a type converter is available
    if (localized != null &&
        this.Converter == null &&
        _typeConverter != null &&
        _typeConverter.CanConvertFrom(localized.GetType()))
        localized = _typeConverter.ConvertFrom(localized);

    // Apply a type converter if one was specified
    if (Converter != null)
        localized = this.Converter.Convert(localized,
            _targetProperty.PropertyType,
            null, CultureInfo.CurrentCulture);
}
catch (Exception ex)
{
    Trace.WriteLine(string.Format(
        Resources.ConversionErrorMessageFormatString,
        Id, ex.Message));

    localized = null;
}

// If no fallback value is available, return the key
if (localized == null)
{
    if (_targetProperty != null &&
        _targetProperty.PropertyType == typeof(string))
        localized = string.Concat("?", Id, "?");
    else

```

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

```
        return DependencyProperty.UnsetValue;
    }

    // Format if a format string was provided
    if (this.Format != null)
        localized = string.Format(CultureInfo.CurrentUICulture,
                                this.Format, localized);

    return localized;
}
```

Markup extensions receive information about the control and property that they are binding to, which provides a control object instance and the dependency property that is bound to. Based on the dependency property you can also retrieve a type converter which allows converting values from strings to the appropriate property type. These references can be used to set the value, based on the string that is retrieved from a ResourceManager.

The Markup Extension has the properties mentioned earlier (Id, ResourceSet, Default etc.) and based on these properties the code attempts to retrieve the appropriate ResourceManager and resource id. Once a value has been retrieved the type converter can be applied to it to format it properly. If an error occurs or the resource Id lookup fails it's then possible to fix up the returned result – typically by assigning a default value if specified.

The markup extension implementation is a simple, but practical example of what you can accomplish relatively easily. There's a lot of power in this mechanism and as you can see it's quite easy to create custom behaviors.

For an example of this markup extension in action take a look at the WpfLocalizationResx project and the LocalizationInfo.xaml document which uses both StaticExtension and the ResExtension for localization.

The benefits and shortcomings of custom markup extensions have some similarities with static bindings. You are still dealing with binding expressions and so you can only bind to dependency properties. Note also that you can also use static bindings and a markup extension in combination. If you're always binding to Resx string resources, then using *x:Static* is often the easiest and most efficient choice except when you need to bind to a non-string value in which case you can use the markup extension.

Either way binding to Resx resources provides you with maximum flexibility and control over the binding process even though the initial process of mapping resources to controls and properties can be more involved.

The pros of custom markup extensions compared to the StaticExtension are:

- **More flexible than static bindings**
They offer you more control over the binding process by allowing you to specify binding

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

parameters as part of the extension to customize the binding behavior exactly to your needs. TypeConverter and Default value support in particular are key to localization extensions.

- **Use either strongly-typed resources or raw ResourceManager**

You can choose between binding to static strongly-typed resources or using custom ResourceManager instances that your markup extension instantiates.

- **TypeConverter support allows for string representations of complex types**

By internally applying type converters custom markup extensions can ensure that you can bind to non-string properties easily using the same string values you already use in XAML markup attributes.

- **Allows for custom ResourceManager or resource storage**

Markup extensions give you complete freedom over how the resources are loaded so you can use a custom ResourceManager or even create a complete separate resource store that directly loads resources from XML or a database and bypasses ResourceManager entirely.

- **Support for resource load failures in the designer**

Resource load failures can break applications at runtime as well the design time experience. This is especially true in satellite assembly scenarios where resources often cannot be found by the Visual Studio and Blend designers. Default values are useful to ensure the designer always works even if resources are not directly accessible.

- **Changing cultures on the fly**

Markup extensions support the ability to force an update of the target they are bound to, so if you can detect a culture change it's possible to update the value that it's bound to without reloading the document. This topic is discussed later in this whitepaper.

The cons are:

- **Requires custom configuration**

The custom markup extension namespace has to be registered in every page and likely requires some global configuration at application startup similar to the `LocalizationSettings.Initialize()`.

- **Blend designer issues**

Due to the strict security environment and the way Blend resolves resource assemblies there are problems seeing live resources represented in the Blend designer. Default values can mitigate this issue and in Blend you can see default values if set.

- **Non-standard solution**

A custom extension by nature is not a built-in solution. This means localizers have to install an external assembly, and have to use custom syntax to use the markup extension.

Attached Property Binding

WPF allows for another powerful mechanism of hooking up resources to elements: *Attached Properties*. Attached properties allow for extension of existing elements and controls without creating a subclass. Rather attached properties can be attached to existing elements and allow for code to be executed when the attached property is changed. When a property is changed the value for the change is passed along with an instance of the UIElement that the change occurred on.

For localization this offers some interesting opportunities. If you're familiar with the way both Windows Forms and Web Forms use control groups for localization of control properties – for example binding `lblName.Content`, `lblName.ToolTip` to the corresponding properties on a control – then you will already have a good understanding of the attached property concept to be discussed here.

With markup extensions every single bound property we want to bind to – Content, Tooltip, Width, Margin, etc. – has to be explicitly mapped in XAML code with binding syntax. Using attached properties you can simplify the process by removing the explicit property binding and instead moving to a model where you specify a marker property on an element to indicate that you want to localize it. Based on an identifier the resources are then matched in the Resx file and all matching properties are bound.

Using a Translate property defined on Translate Extension looks like this:

```
<Window
  x:Name="wndLocalizationInfo"
  x:Class="WpfLocalizationResx.LocalizationInfo"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:res="http://www.west-wind.com/WpfControls"
  xmlns:local="clr-namespace:WpfLocalizationResx.LocalResources"
  Background="{DynamicResource WindowBackgroundBrush}"
  SizeToContent="WidthAndHeight"
  Title="LocalizationInfo"
  res:TranslationExtension.TranslateResourceSet=
    "WpfLocalizationResx.LocalResources.LocalizationInfoRes"
  res:TranslationExtension.TranslateResourceAssembly=
    "WpfLocalizationResx"
>
  <Label x:Uid="lblAttachedValue"
    x:Name="lblAttachedValue"
    res:TranslationExtension.Translate="True"
    Grid.Row="3" Grid.Column="1"
    >Hello World Attached Property Text (non res text)</Label>
</Window>
```

There are three attached properties involved:

- **TranslateResourceSet**
Document level property that specifies the resource set that is used for this document translation.

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

- **TranslateAssembly**

The name of the assembly that is to be used localization. If omitted (and usually you can) the document class's assembly is used.

- **Translate**

A simple flag that can be applied to any element that you have properties on that should be translated. You simply set `res:TranslationExtension.Translate="True"` on any element and then provide the appropriate Resx resources keys in the resource set to translate with where the syntax of the key is *elementName.propertyName*. Only string values can be assigned with this approach.

The Resx file then contains entries for each element like this for the element named `IblAttachedValue`:

LocalizationInfo.resx:

Resource Key	Value
<code>IblAttachedValue.Content</code>	Hello World (Res)
<code>IblAttachedValue.Tooltip</code>	Hello World Tooltip

LocalizationInfo.de.resx:

Resource Key	Value
<code>IblAttachedValue.Content</code>	Hallo Welt (Res)
<code>IblAttachedValue.Tooltip</code>	Hallo Welt Tooltip

When the page is loaded the `Translate` property is assigned to `True` which triggers the attached property's custom code to fire. A `ResourceManager` then finds the right resource set as provided by the top level document `TranslateExtension.ResourceSet` element and finds all properties with the element's name as a prefix. All matching resource keys assigned using Reflection to the appropriate elements. The slightly truncated code shown in Figure 22 illustrates how this can be accomplished (see the completed `WpfControls` sample for more).

Figure 23: Attached Property implementation that assigns all matching properties of an element

```
public class TranslationExtension : DependencyObject
{
    public static readonly DependencyProperty TranslateProperty =
        DependencyProperty.RegisterAttached("Translate",
            typeof(bool),
            typeof(TranslationExtension),
            new FrameworkPropertyMetadata(false,
                FrameworkPropertyMetadataOptions.AffectsRender,
                new PropertyChangedCallback(OnTranslateChanged)) );
}
```


WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

```
public static void SetTranslate(UIElement element, bool value)
{
    element.SetValue(TranslateProperty, value);
}
public static bool GetTranslate(UIElement element)
{
    return (bool)element.GetValue(TranslateProperty);
}
private static void OnTranslateChanged(DependencyObject d,
    DependencyPropertyChangedEventArgs e) {
    if ((bool) e.NewValue == true)
        TranslateKeys(d as FrameworkElement);
}

static void TranslateKeys(UIElement element)
{
    if (DesignerProperties.GetIsInDesignMode(element))
        return; // just display XAML doc values

    // walk the element tree to find the top level document
    FrameworkElement root = WpfUtils.GetRootVisual(element
        as FrameworkElement);

    if (root == null)
        return; // must be framework element to find root

    // Retrieve the ResourceSet and assembly from the top level element
    string resourceset = root.GetValue(TranslateResourceSetProperty)
        as string;
    string resourceAssembly =
        root.GetValue(TranslateResourceAssemblyProperty) as string;

    ResourceManager manager = null;
    manager = LocalizationSettings.GetResourceManager(resourceset,
        resourceAssembly);

    // Look at Neutral Culture to find all keys
    ResourceSet set = manager.GetResourceSet(
        CultureInfo.InvariantCulture,
        true, true);
    IDictionaryEnumerator enumerator = set.GetEnumerator();

    while (enumerator.MoveNext())
    {
        string key = enumerator.Key as string;
        if (key.StartsWith(element.Uid + "."))
        {
            string property = key.Split('.')[1] as string;
            object value = manager.GetObject(key); // enumerator.Value;

            // Bind the value AFTER control has initialized or else the
            // default will override what we bind here
            root.Initialized += delegate
            {
```

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

```
try
{
    PropertyInfo prop = element.GetType()
                               .GetProperty(property,
                               BindingFlags.Public |
                               BindingFlags.Instance |
                               BindingFlags.FlattenHierarchy |
                               BindingFlags.IgnoreCase);
    prop.SetValue(element, value, null);
}
catch (Exception ex)
{
    Trace.WriteLine(...);
}
};
}
}
}
```

The code consists of a class `TranslateExtension` that contains an attached property declaration for *TranslateProperty* as well as *TranslateResourcesSet* and *TranslateAssembly* which are not shown here. Attached properties, like dependency properties, are static declarations and consist of a set of methods that are called when values are set or retrieved. The static *RegisterAttached* method registers an attached property and makes it available to the XAML parser to apply against other elements. The syntax for the Translate property is simply `res:TranslateExtension.Translate="True"` which when set in markup fires the `OnTranslateChange` change event hooked up in the registration.

`OnTranslateChange` calls `TranslateKeys` which is responsible for finding all resource Ids that match the element's x:Uid value. You can assign the x:Uid value manually on elements or – as discussed earlier for LocBaml localization - by using `msbuild /t:updateuid` to automatically create x:Uid attributes for all XAML elements in the entire project.

`TranslateKeys` figures out which `ResourceSet` to use based on the `ResourceSet` and `Assembly` attached properties assigned in the the root element of the document. Doing so allows setting these values only once in a document rather than on each element, effectively providing page level context. Internally the `ResourceManager` retrieved is cached by the resource set name so this process is fairly efficient.

Once a `ResourceManager` is available it's used to retrieve all the default resources (using the `CultureInfo.InvariantCulture` as the culture) which should contain *all* resource keys available. The code loops through all of them to find any that start with the same name as the element we're working with based on the x:Uid property plus the separator dot (ie "lblName."). If a match is found the property name is extracted from the full resource key and Reflection is used to apply the resource value to the property.

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

Note that the Reflection value assignment occurs inside of an anonymous delegate hooked to the element's *Initialized* event. This ensures that the localized value is assigned properly after XAML has assigned its default values defined in the document.

The advantage of this attached property approach is that you don't have to map every individual property you want to bind with lengthy binding syntax. Using the Translate attached property you are setting only a single, cut and pasteable property on an element that indicates that it should be translated by finding any matching properties and applying them. The Translate property can also be set in the designer so you don't have to jump into XAML. Notice also that you can specify your default text as you normally would if you weren't localizing. The default values are applied as always and that's what you will see in the designer. At runtime the localized text overwrites any default static text with the localized values after initialization is complete.

The pros of attached property assignments are:

- **Easier element mapping**
Set a single generic property on each element and use naming conventions to assign element keys to map to properties. The property can be either entered and easily cut and pasted in XAML or you can set the value in the Visual Studio or Blend designers.
- **Unobtrusive design-time values**
You can continue to enter default property values and content as you normally would. There's no custom binding syntax. The standard property values/content show in the designer and the localized text only shows up at runtime.
- **Easy to add after development is complete**
Because you simply add a single attached property value to each element it's easy to add this functionality after development is complete by cutting and pasting into XAML. To make something localizable simply add the `res:TranslateExtension.Translate="True"` attribute to the element and add the appropriate Resx keys to the Resx file.

The cons of attached property assignments are:

- **No type converter support**
Because this mechanism infers element property names dynamically at runtime there's no access to the underlying dependency properties and their type converters. This means that you can only use string or simple numeric values work for localization. This behavior is same as discussed in the `x:Static` binding.
- **Somewhat less efficient**
Attached properties are called generically and without context. You get the value and the element, but no information about the dependency property bound to. This means every time the attached property is called the context needs to re-establish the root element of the page,

the resource set and assembly. Additionally for every access the entire ResourceSet has to be enumerated in order to find elements that match the element's x:Uid. Finally reflection is used to assign the values. For very complex documents with lots of localizable values there maybe a slight performance hit for all of this iteration.

- **Works only with contained elements**

Because this component relies on a root element to retrieve TranslateResourceSet and TranslateAssembly, this component has to be able to find the root element which is done by walking up the element containership hierarchy. However, some UI elements like ContextMenu are not part of the document's logical tree and so can't find the root element. For these components and their children you have to explicitly add the TranslateResourceSet and TranslateAssembly on the 'parentless' elements or use a markup extension.

Other Topics of Interest

Following are a few WPF localization related topics that have come up in discussions and research for this article.

Use Autosizing for Elements

One area WPF excels in regards to localization is how it handles layouts to accommodate changing text sizes for different cultures. When creating user interfaces it's often difficult to allocate sufficient space to fit static text in various languages into the allotted space. The key to flexible user interfaces is to have auto-sizing elements that can stretch or shrink to accommodate these different text sizes. WPF's flow based layout makes this process easier than previous rich client technologies by making it easy to create flowing user interface layouts that can adjust with resizing and different sizes and widths of text.

Here are a few tips for maximizing auto-sizing:

- **Use `SizeToContent.WidthAndHeight` on Windows**

This feature ensures that window sizes properly adjust when content widens inside of element content when initially loaded.

- **Avoid fixed Width and Height where possible**

Avoid using any fixed widths and heights to allow WPF to auto-flow content and size elements to their actual length. This means using auto sizing wherever possible. Avoid using *Canvas* with its absolute positions and sizes that don't allow for content to expand and contract and use any of the auto-sizing containers instead to compose your layout.

- **Enable TextWrapping in TextBlocks**

Turn on *TextWrapping* in *TextBlocks* to avoid text from clipping. Let text wrap and allow the container to adjust and flow with the document.

- **Use SharedSizeGroup in Grids to create uniform sizing across controls**

Grid *ColumnDefinitions* allow assignment of a *SharedSizeGroup* which makes all columns that

are assigned to the same *SharedSizeGroup* the same size. This is useful especially for button groups where each button should have the same size and still be long enough to fit the longest content.

- **Use MinWidth to avoid tiny elements**

During localization we're often worried about text getting too long to fit into allotted space but sometimes the opposite actually occurs: You end up with content that's too short and looks ugly. For example if you have an Autosizing button and Ok and Cancel text on the button these buttons will look funky as Ok is very short compared to the more normal Cancel button width. Using MinWidth especially on buttons, dropdowns and combo boxes can bring some uniformity to standard elements without any layout tricks, but still allows larger content to extend and create larger elements when necessary.

Right To Left Display

Certain cultures like Hebrew and Arabic flow text from right to left and localized applications should reflect the *UICulture*'s flow direction. Flow direction can be queried from *CultureInfo* with:

```
bool rtl = CultureInfo.CurrentCulture.TextInfo.IsRightToLeft;
```

Unlike resource localization which happens automatically, flow direction has to be explicitly assigned to a document. Luckily this can be done easily setting the *FlowDirection* property on a top level container and all child elements automatically inherit the setting and flow their content accordingly.

You can assign *FlowDirection* in code like this:

```
public LocalizationInfo()
{
    if (CultureInfo.CurrentCulture.TextInfo.IsRightToLeft)
        this.FlowDirection = FlowDirection.RightToLeft;

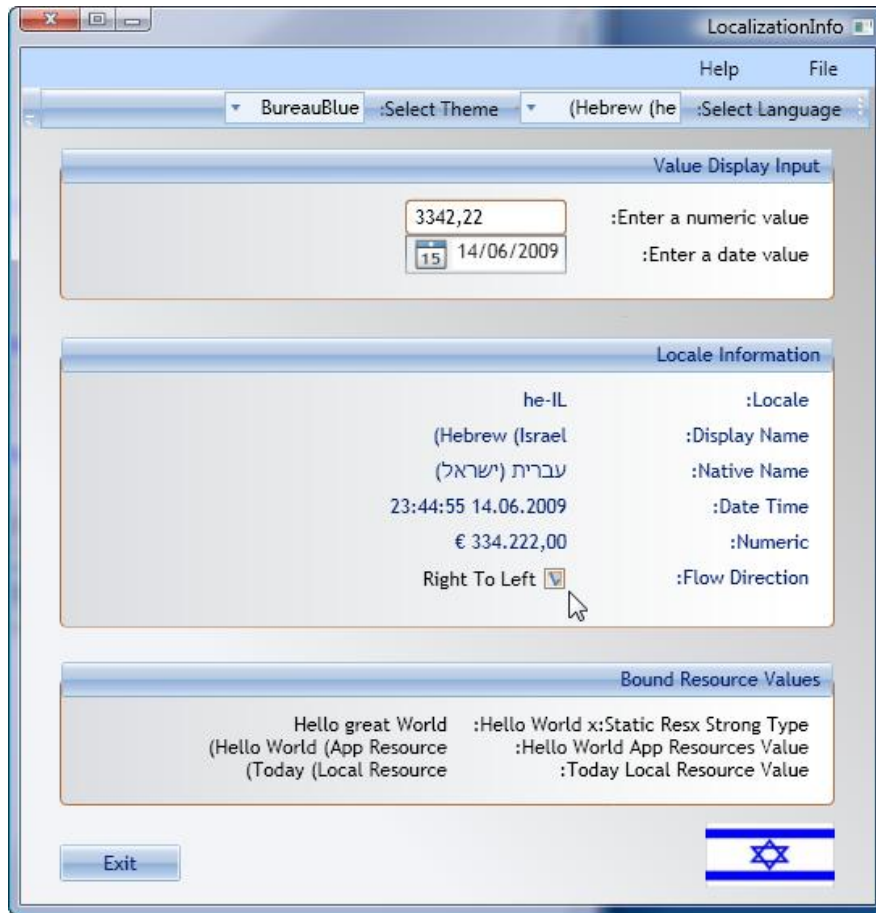
    InitializeComponent();
}
```

Figure 24 shows a (non-localized) form with *RightToLeft* set when switching into Hebrew which is a RTL language.

Figure 24: Right to left display is easy to accomplish in WPF by setting the *FlowDirection* on the document

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009



Note that certain user interface elements like context menus are not part of the document hierarchy so when you set FlowDirection on the document the context menu is not automatically updated. The ContextMenu's FlowDirection has to be updated explicitly.

Switching Languages on the Fly

If your application needs to run in multiple languages, one way or another you need to allow for switching languages and this process can be a very simple or fairly complex process. Ideally you want to set your application's language at startup according to the user's culture preference to ensure all forms and startup messages in the application receive the current culture.

Switching languages is easy enough in code – you can simply assign a new Culture and UICulture with a generic routine like the following static method placed on the App object:

```
public static void SetCulture(string culture)
{
    if (!string.IsNullOrEmpty(culture))
    {
        CultureInfo ci = new CultureInfo(culture);
        Thread.CurrentThread.CurrentCulture = ci;
    }
}
```

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

```
        Thread.CurrentThread.CurrentUICulture = ci;
    }
}
```

Calling this method from anywhere in your application will change the culture and result in UI resources served for the new specific language.

This makes good sense at the beginning of your application where you can possibly read the startup culture from a configuration file:

```
public App()
{
    SetCulture(ConfigurationManager.AppSettings["Culture"]);
    InitializeComponent();
}
```

But things aren't as straight forward when you change the culture after the application is executing. The new Culture and UICulture are applied immediately to formatting and resources loaded from this point forward so any new documents see the new culture immediately. The problem is that forms already loaded, including the main UI, won't automatically reflect the culture shock.

One approach is to close all windows of the application and reload the main window. Figure 25 demonstrates a generic *SetCulture* method with the ability to close all forms and re-open the main form.

Figure 25: A generic method to set the Culture and UICulture and providing an option to close all active windows and restart the main form

```
public static void SetCulture(string culture,
                              bool closeAllWindowsReloadMain)
{
    if (culture == null)
        return;

    bool cultureChanged = culture !=
        Thread.CurrentThread.CurrentUICulture.IetfLanguageTag;
    if (!cultureChanged)
        return;

    if (!string.IsNullOrEmpty(culture))
    {
        CultureInfo ci = new CultureInfo(culture);
        Thread.CurrentThread.CurrentCulture = ci;
        Thread.CurrentThread.CurrentUICulture = ci;
    }
}
```

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

```
if (closeAllWindowsReloadMain)
{
    Type mainWinType = App.Current.Windows[0].GetType();
    Window mainForm =
        Assembly.GetExecutingAssembly()
            .CreateInstance(mainWinType.FullName) as Window;
    mainForm.Show();

    // close all other windows
    foreach (Window win in App.Current.Windows)
    {
        if (mainForm == win)
            continue;

        win.Close();
    }
}
```

This works fairly well for single form or relatively small applications, but is probably too blunt for more sophisticated applications that might have active data that has to be saved before the application can shut down. Another option would be to warn the user that changing languages will require the application to restart, which is a single line of code. Yet another option might be to ask the user for the preferred culture the first time the application runs before the main form is loaded, and then restart the application silently after the information is written to the configuration file.

Assigning a Resx Image Resource to an Image Control

If you store images in Resx resources rather than as loose or compiled XAML resources stored in folders of your application, there's a bit of extra effort involved in loading the image and reading it explicitly into an Image control. Why would you want to store an image inside of a Resx ResourceSet when you can just store a XAML resource as an embedded file resource? File resources in WPF just like standard file resources in any other project are not localizable. They are embedded as resource streams but there's no localization information related to them. If you need to localize an image like a flag for example, you still need to use a Resx ResourceSet-based image resource.

Unfortunately WPF doesn't like to work directly with Bitmap images and instead requires translation of images into image sources. WPF prefers to load any image resource from a URI or with some manipulation from a stream which makes the process of loading a plain old Bitmap more complicated than it should be. For example, if you have a strongly-typed Bitmap stored in resources in `LocalizationInfoRes.CountryFlag` here's how you load this bitmap into an `Image.Source` property:

```
if (this.IsInitialized)
{
    MemoryStream ms = new MemoryStream();
```


WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

```
LocalizationInfoRes.CountryFlag.Save(ms, ImageFormat.Jpeg);  
ms.Position = 0;  
this.imgFlag.BeginInit();  
this.imgFlag.Source = BitmapFrame.Create(ms);  
this.imgFlag.EndInit();  
ms.Close();  
}
```

Notice that this code is executed only after the main document has been initialized. Failing to do so causes the image not to be loaded, without warning. The trick to loading a Bitmap instance is to use the BitmapFrame class which allows reading an image from a stream. The example shows using a Bitmap, writing it out to a MemoryStream and then assigning that stream as input to BitmapFrame.Create().

Summary

Localization in WPF is a complex topic and there are many options to handle the process for the developer. Unlike previous Windows desktop technologies that preceded it, WPF does not have one clear path to localization and no direct tool support in Visual Studio or Blend. This means the choices for localization are left up to the developer to explore and figure out.

The two major choices available are using LocBaml for localizing XAML resources or using the more traditional Resx approach. LocBaml focuses on using the XAML document itself as a resource store and allows for exporting of these XAML resources for localizations after development is complete. The rigid LocBaml approach lends itself to applications that are complete and won't change much but is not a good choice for applications that need to be incrementally localized. Resx provides better tool integration in Visual Studio with support for resource editing right in Visual Studio as well as the ability to create strongly-typed resource classes from Resx resources. For WPF accessing Resx resources involves the manual process of binding resources to XAML elements which is more work up front compared to LocBaml but allows for more flexibility when updating applications at a later point. The Resx approach also has number of options available. The simple *x:Static* binding markup extension makes for easy bindings of strongly-typed resources to properties using simple, built-in binding syntax. For more control custom markup extensions or attached properties can be used to provide custom binding to Resx resources.

The LocBaml approach is interesting in that it defers localization until the end of a project, but in its current form with the limited tool support in LocBaml itself and lack of integration the process of localization is very fragile – it's very easy to break the CSV localization files and have to start over or roll back to a previously saved version. Using LocBaml it's definitely worthwhile to back up *constantly*! Resx offers a more traditional approach and while it's definitely more work during development this is the more stable and consistent approach to localization at this time. There are lots of choices available for Resx localization as well beyond what has been discussed in this white paper, many of them interesting and innovative. Lots of innovation around Resx extensions, but nobody seems to be extending LocBaml.

WPF Localization Guidance

Rick Strahl & Michele Leroux Bustamante, June 2009

The bottom line is that as a WPF developer you need to spend a little time with each of these approaches to get a feel for what works for you and what works for your localization staff. Choices are good, but arriving at the right one unfortunately can take a little extra time. Hopefully this whitepaper has given you a good background on some of the tools available and a better understanding what to look for in any solution you use for WPF localization.

Acknowledgements

The following articles have been very helpful in establishing the state of WPF localization today and have provided valuable ideas and concepts incorporated in this whitepaper.

Csv Merging Article on CodeProject by André Heerwarde

<http://www.codeproject.com/KB/WPF/LocBamlClickOnce.aspx>

Localize a WPF Application by using a Markup Extension by Christian Moser

<http://www.wpfutorial.net/LocalizeMarkupExtension.html>

.NET Internationalization Book by Guy Smith Ferrier

<http://www.informit.com/store/product.aspx?isbn=0321341384>

Resources

LocBaml Tool Sample

<http://msdn.microsoft.com/en-us/library/ms771568.aspx>

.NET Internationalization Book by Guy Smith Ferrier

<http://www.informit.com/store/product.aspx?isbn=0321341384>

.NET Reflector

<http://www.red-gate.com/products/reflector/>