



Resources in Microsoft .NET

Bill Hall

Taking localization techniques one step further; for advanced users only

Introduction

Sometime around the year 2000, Microsoft Corporation released its .NET (read as "dot net") integrated programming environment to consolidate and simplify the disparate set of tools (COM, database operations, Win32 programming, code pages versus Unicode, etc.) that had emerged over a period of 20+ years of Windows development¹. Indeed, .NET was most welcome to those of us with a strong interest in internationalization, localization, and a transparent approach to Unicode. Finally, Microsoft had provided a clean, class- and Unicode-based system with effective and easy-to-use locale and resource models.

In this article, I will illustrate a localization technique in .NET that appeared in its first point release, version 1.1. The methodology is classic, and the fundamental idea is well-known in the localization community. The basic concept revolves around a combination of an embedded default resource containing a fallback language surrounded by satellite resources to handle additional scripts.

In a follow-up article, you will learn about an interesting concept, which first appeared in .NET 2.0. In this case, the satellite resources vanish in favor of a single, strongly-typed resource. The result is robust and less prone to failure, but you may have to work a bit harder to create versions of your program for different languages.

Basics

In .NET, resources are XML based and the files have a .resx extension. A .resx file provides the basic machinery for storing key and object pairs. Usually, an object is itself a string but more complex structures are also possible, including icons, bitmaps and other graphic structures, if they can be converted to Base64. Tools are available in .NET for this purpose. In this article, I use only simple text in the examples.

A .resx file begins with a schema and is followed by key / data pairs containing resource information. Following is an abbreviated example: The key string is "tram" and the value string is a comment about the ironies of life. The quote is from Camillo Sbarbaro, translated into English.

¹ My first job in the computer world was to port Windows 1.0 to Olivetti and AT&T computers. A significant part of the code was 8086-based. Previously, I had worked as associate professor of mathematics at a large American university, teaching undergraduate and graduate students. It was a paradigm shift that occurred at the age of 50.



```

< ?xml version="1.0" encoding="utf-8" ? >
<root>
  <xsd:schema > ... </xsd:schema>
  ...
  <data name="tram">
    <value>life is like a tram, when you get to sit down it's the end of
the line</value>
  </data>
  ...
</root>

```

Figure 1: Basic .resx format

Of course, the native .resx format is not an overly friendly place to work unless you enjoy complexity. Hence, for practical reasons, the localizer or developer is provided a table where the desired entries can be inserted. During compilation, the table contents are converted to the .resx format. Any comments are also retained but cannot be accessed directly. You will have to use XML techniques to retrieve the value. If you add satellite resources to your project, you get a set of additional tables as well.

Name	Value	Comment
tram	Life is like a tram, when you get to sit down it's the end of the line	From the works of Camillo Sbarbaro
...

Figure 2: Example of resource file entry table

The Resource Model in .NET 1.1

The resource model in .NET 1.1 will be quite familiar to experienced localizers, although the details are quite different from those used to manage Windows resources. There is a main .resx file that is compiled into the executable assembly. The content can be in any language; you normally choose it as the final fallback. In the example presented below, the fallback is to English. The fallback concept is important since if all else fails, at least some information will be displayed, even if it is not readily understood by the user.

Satellite resources are added to take care of other languages. They have a special naming convention consisting of a base name, a Microsoft RFC 1766 identifier (language or language + region such as es, es-ES, fr, de-DE, etc.), and the extension .resx (<base name>.<rfc1766Id>.resx). It is quite straightforward to add such a resource file to the project. For each language, you have access to an additional entry table, which is compiled for you during the build. The resulting satellite assembly is placed in a directory having the same name as the RFC 1766 identifier and located just below the directory of the main assembly. Figure 3 shows some of the details for the release version in our example. Note the relationship between the French resources and the executable itself. Do keep in mind that the fallback resource resides in the original assembly and all other languages are stored in the satellite DLLs.

As for the code, it is quite simple in this case. However, if you do not have experience working with .NET, it might take you a bit of time to learn exactly which parameters are required when using the Resource Manager (See the listing in Figure 4). I hope you do not mind having to read and understand a bit of code. The programming language is C#, and if it is any comfort, C# is one of the easiest to learn of today's modern programming languages.

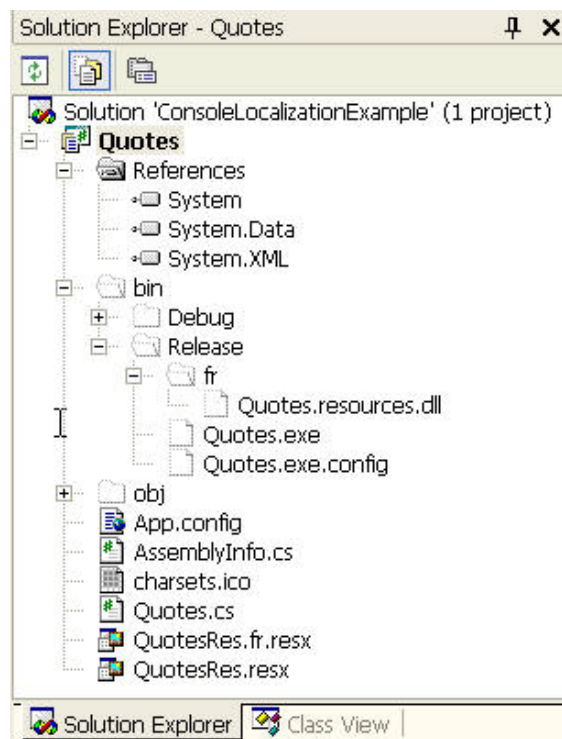


Figure 3: Project directory file locations

In this example, I use some .NET configuration tools to obtain an RFC 1766 identifier that specifies the *locale* (or in .NET terms, the *culture*) of the resources. Then a *ResourceManager* object is created referencing the program (*Assembly* in .NET parlance). The *ResourceManager* then requests the quote using the keystring "tram." Note that the result depends on the culture instantiated by the RFC 1766 identifier. Finally, the output is either displayed or the exception handler reports an error.



This is the result, when the identifier is "fr" (French) and the retrieval successful:

« la vie c'est comme un autobus: dès que tu réussis à t'asseoir, tu dois descendre »

```
using System;
using System.Globalization;
using System.Resources;
using System.Reflection;
using System.Configuration;

namespace ConsoleLocalizationExample {
    class Quotes {
        [STAThread]
        static void Main(string[] args) {
            // use app.config to set culture
            AppSettingsReader rdr = new AppSettingsReader();
            // read RFC-1766 identifier to determine the user interface language
            string rfcid = (string)rdr.GetValue("culture", typeof(System.String));
            // create a resource manager referencing the program
            ResourceManager rm = new ResourceManager("Quotes.QuotesRes",
                Assembly.GetExecutingAssembly());

            // things can go wrong, so use try / catch to get useful information
            try {
                // if successful, output will be the quote in the expected language
                Console.WriteLine(rm.GetString("tram", new CultureInfo(rfcid)));
            } catch (Exception ex) {
                // failed, show exception information
                Console.WriteLine(ex);
            }
        }
    }
}
```

Figure 4: Code for the sample project



For any other culture, expect only English (this is the fallback²; remember the original .resx file we created?) You can experiment by changing the 'value' setting in the app.config file, and if you feel that you understand the programming steps, you can try to add another satellite resource to test your skills. For example, try Czech ("cs" is the language tag) as the language: The following translation will appear (but be sure you set the output code page in the console to UTF-8; if you pass this test, you are doing well!).

"v životě je to jako v tramvaji, když si konečně můžeš sednout, je konečná"

After looking at the code, you should be acutely aware that successful retrieval of the message depends on having a *ResourceManager* object that can reference the main and any of its satellite assemblies. Using the manager requires instantiation with two parameters, the name of the assembly, the .resx file, and the currently executing assembly. Retrieving the string itself may require dynamically changing the user interface culture, as we did in the sample code. This capability is uniquely available to .NET programs and is a godsend to those of us testing such programs.

Alternatively, if you are able to switch your operating system to another user interface, you can try changing it to test your results. This approach, however, does require you to have the Multilingual User Interface (MUI) installed on your computer. That, in turn, means that your computer must be the U. S. English version of Windows with the required MUI language installed. My advice is to perform the test using the appropriate app.config setting (Figure 5).

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key = "culture" value = "fr" />
  </appSettings>
</configuration>
```

Figure 5: App.config file for the .NET 1.1 project

² When running the program, you may want to check that the results appear correctly in the console output. The console in Windows is usually set to a code page that matches its user interface language. Therefore, you may want to set the console code page to the expected output code page or to UTF-8 before running the program at the command level (look up the console command *chcp*). In .NET 2.0, you can bypass the *chcp* setting and use a new API to set the recommended output encoding (UTF-8). The result in the console may still appear to be incorrect, but redirection to a notepad file will guarantee proper results. The best solution would be for Microsoft to fix Unicode's poor visual support in the console once and for all!



Summary

The basic approach to localization in Microsoft .NET is familiar. Admittedly, the details may be new to you but the methodology is not difficult to learn and implement. However, time has passed since .NET 1.1 was released, and activity is now centered in .NET 2.0 and .NET 3.0 (Vista).

Expect a follow up in due course on strongly typed resources using .NET 2.0 / 3.0.

Bill Hall has been a developer and consultant for Windows and Win32 platforms with experience dating back to Windows 1.0, which he ported at the systems level to AT&T/Olivetti computers. A system and applications programmer throughout his computing career, he turned to internationalization in the early 1990s, taking several projects into European and Far Eastern languages and contributing numerous articles to Microsoft Systems Journal and Multilingual Computing. Currently, he is writing a series of monographs on .NET internationalization and localization and teaching Windows and .NET globalization. In past lives, Bill was a military and civilian aviator, an associate professor of mathematics for nearly 20 years, and served for three years as associate editor at Mathematical Reviews.

<Glossary>

.NET Assembly

In the Microsoft .NET framework, assemblies are partially compiled code libraries for use in deployment, versioning and security. There are two types, process assemblies (EXE) and library assemblies (DLL).

C#

Object-oriented programming language developed by Microsoft as part of their .NET initiative, and later approved as a standard by ECMA and ISO. C# has a procedural, object-oriented syntax based on C++ that includes aspects of several other programming languages (most notably Delphi and Java) with a particular emphasis on simplification.

Fallback

System for supporting the CPU in the event of an emergency and for retrieving lost data.

MUI

The Multilingual User Interface (MUI) is a Microsoft feature that gives the end user the ability to change the language of the user interface (UI). One such example would be when an end user working with an English UI wants to change the language to Spanish. If your application is intended for multiple language environment users, you can add code to automatically change your UI strings to match the language set up in a certain computer (assuming that it has the correct resources installed.)

Satellite Resources

Satellite resources are specific purpose resources that can be used instead of the main resource, which is for general purposes. In localization, the main resource could be in English, for example, while there would be several related satellite resources, each specific to a different language, such as French, Portuguese, Russian, and so on.

Strongly-Typed Programming Language

In computer science and computer programming, the term strong typing is used to describe how programming languages handle datatypes.

Unicode

Industry standard designed to allow text and symbols from all of the writing systems of the world to be consistently represented and manipulated by computers.

UTF-8

8-bit UCS/Unicode Transformation Format is a variable-length character encoding for Unicode. It is able to represent any universal character in the Unicode standard. For these reasons, it is steadily becoming the preferred encoding for e-mail, web pages, and other places where characters are stored or streamed.