# ASP.NET 2.0 Localization Features: A Fresh Approach to Localizing Web Applications

Michèle Leroux Bustamante

IDesign Inc

October 2004

Updated April 2006

Applies to:

- Microsoft ASP.NET 2.0
- Microsoft Visual Studio .NET
- Microsoft Visual Studio 2005
- Localization

**Summary:** As companies reach out to other countries for business, creating global Web applications with Microsoft ASP.NET is becoming more and more important. ASP.NET 1.1 supported creating localized Web sites by means of the **ResourceManager** class. ASP.NET 2.0 makes it even easier to provide support for multiple cultures and locales through improved runtime and tool support. (23 printed pages)

Download the source code for this article.

**Contents**

Introduction

Localization with .NET 1.x

Introducing New ASP.NET 2.0 Localization Features

Generating Local Resources

Global Application Resources

Implicit Localization Expressions

Explicit Localization Expressions

Localizing HTML Elements and Static Text

Resource Localization and Deployment

Runtime Resource Providers

Preferred Culture Selection

Conclusion

Additional Resources

## Introduction

The business community has been reaching international markets since the Internet as we know it began in the mid-nineties. Not only do companies have a global Web presence through their corporate Web sites, but an increasing number are also hosting and/or licensing enterprise Web applications that may ultimately service customers all over the world. Preparing these Web sites and applications for localization is a necessity since most clients prefer to conduct business in their own native language and cultural environment. There is no question that this undertaking requires planning and effort, regardless of the deployment model (smart clients, Web sites and applications, Web services), specifically in terms of architecture and design. Effective development tools can influence that architecture for the better, but strangely, advancements in the area of tools for Web application localization are still lacking. As a result, many businesses resort to more traditional and cumbersome methods for localization that cost them significant time and money.

The release of Microsoft ASP.NET 2.0 promises to shift the way Web developers approach localization, for the better. The Microsoft Visual Studio .NET environment has expanded the developer toolbox, added new runtime capabilities, and providing a rich new programming API specifically targeting localization requirements. Now developers will be able to more quickly separate localizable content from their ASP.NET pages, reduce their coding effort to access localized content, and extend the environment to meet additional requirements while leveraging a consistent programming model. This whitepaper will take you on a tour of the core features of the new and improved ASP.NET development experience for preparing applications for localization.

## Localization with .NET 1.x

The .NET Framework 1.x introduced a new architecture for localization that supports non-intrusive incremental deployment of culture support by dropping in new satellite assemblies (or, resource-only assemblies). This hub-and-spoke deployment paradigm allows developers to rely on the Common Language Runtime to manage the accurate selection of localized content through the **ResourceManager** class. One of the benefits of the **ResourceManager** is that it encapsulates a resource fallback mechanism to find the culture of "best fit" based on the application's runtime culture setting, thus removing the need for developers to write code to manage the process of loading the correct **ResourceSet** to retrieve localized content.

Resources can be consumed by Windows or Web applications, but Visual Studio 2003 specifically makes it easy for Windows developers to localize their Windows Forms. When a form's **Localizable** property is set

to **true**, resources are automatically generated for localizable **form** and **control** properties. Property values are pushed to default resources for later translation, and code is generated to populate controls at runtime from these resources, using a **ResourceManager** class instance. The **ResourceManager** respects the **CurrentUICulture** setting for the executing thread, and attempts to find resources in the **ResourceSet** matching that culture, otherwise resorting to the resource fallback process.

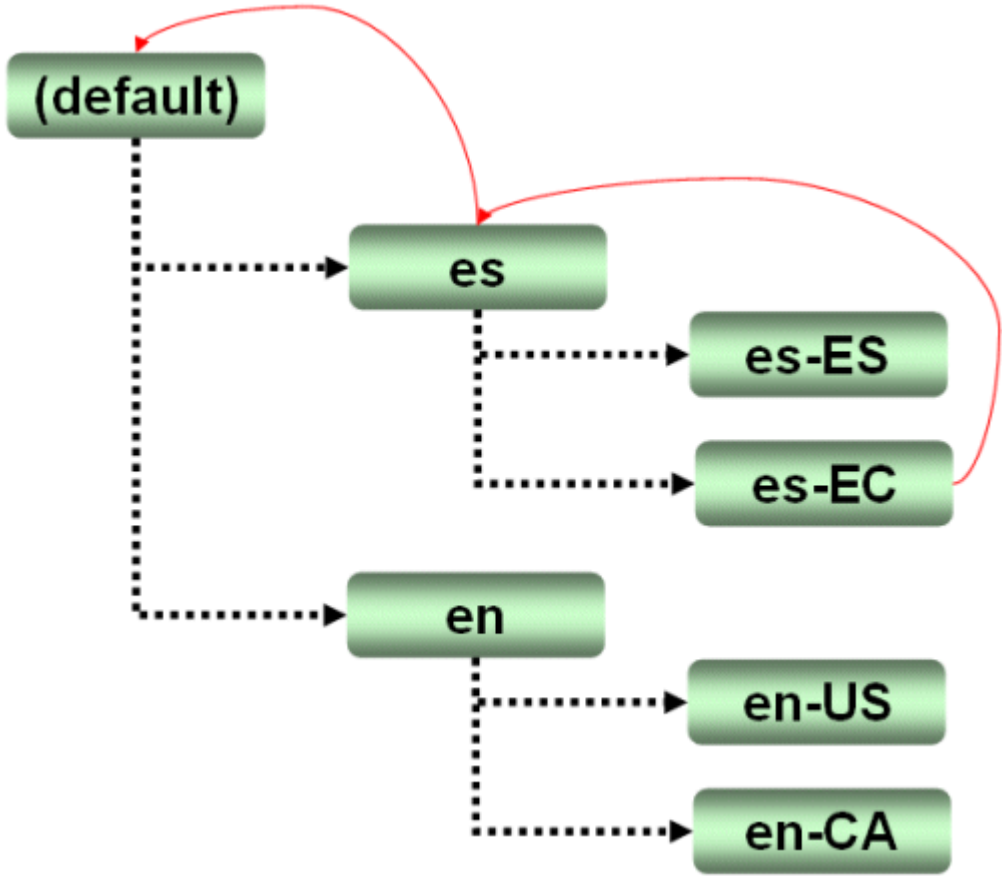


**Figure 1. This illustrates a simple view of resource fallback for a UICulture set to es-EC. The ResourceManager handles probing the es-EC satellite assembly for the requested key first, and subsequently falls back to the neutral Spanish culture es, ultimately looking in the default assembly resources for a value if still not found.**

**Note** Read more about the resource fallback process here.

The .NET Framework also includes many culture-aware classes that respect **CurrentCulture** settings for generating output such as formatted date, time, and currency values, further reducing the amount of tedious work required of developers.

**Note** Read more about culture-aware classes from this MSDN reference.

Although the deployment model of satellite assemblies, resource managers, and culture-aware classes are

also applicable to ASP.NET applications, there is a lack of developer tools to integrate this model into the programming paradigm of Web Forms. With Visual Studio 2003 and ASP.NET 1.1, generating resource entries for Web pages, accessing those resources at runtime, and setting the correct culture for each request requires manual effort. Thus, resources are not widely considered an integral part of the Web application localization story. Localizing Web content traditionally leads to duplicating entire sites for specific cultures, localizing static content in place, and writing custom code for any shared code base to manage the retrieval of localized data sources. This approach is significantly improved with ASP.NET 2.0.

**Note** The following MSDN article discusses architectural approaches to localization for ASP.NET 1.1.

## Introducing New ASP.NET 2.0 Localization Features

ASP.NET 2.0 builds on the foundation of .NET 1.x localization features, specifically improving workflows and functionality available to Web developers. The following is a summary of design goals driving this new generation of localization support:

- Supply tools to support resource generation for Web applications.
- Provide new declarative and runtime programming constructs for accessing resources.
- Simplify the process of applying the correct culture to page requests and automating **ResourceManager** instantiation.
- Support XCOPY deployment and removal of the compile step for small business sites.
- Support the enterprise developer with extensibility for all aspects of resource consumption and management.
- Ensure integrated support for new localization features throughout ASP.NET controls and other applicable runtime components and adapters.

New features baked into Visual Studio 2005 and ASP.NET 2.0 simplify Web application localization by providing tools to help extract localizable content from Web pages, providing more integrated runtime support for resource consumption that complements the stateless request model, supplying modern declarative constructs for binding resources to page output, and providing new ways to automate culture selection for the round-trip. The following features specifically support these goals:

- **Strongly Typed Resources**—At the core of the .NET Framework 2.0 release is support for strongly-typed resources that provide developers with Intellisense and simplify code required to access resources at runtime.
- **Managed Resource Editor**—Visual Studio .NET 2.0 includes a new resource editor with better support for creating and managing resource entries including strings, images, external files, and other complex types.
- **Resource Generation for Web Forms**—Windows Forms developers have already enjoyed the benefits of automatic internationalization. Visual Studio 2005 will now support rapid internationalization by automatically generating resources for Web Forms, user controls, and master pages.
- **Improved Runtime Support**—ResourceManager instances are managed by the runtime and readily accessible to server code through more accessible programming interfaces.
- **Localization Expressions**—Modern declarative expressions for Web pages support mapping

resource entries to control properties, HTML properties, or static content regions. These expressions are also extensible, providing additional ways to control the process of attaching localized content to HTML output.

**Automatic Culture Selection**—Managing culture selection for each Web request can be automatically linked to browser preferences.

**Resource Provider Model**—A new resource provider model allows developers to host resources in alternate data sources such as flat files and database stores, while the programming model for accessing those resources remains consistent.

These features are flexible enough to provide out-of-the-box support for small businesses that require reliable and effective productivity-oriented solutions, yet still support the complex needs of large organizations for numerous deployment architectures with added extensibility. At the heart of this new model is an improved story for leveraging resources for Web applications that aligns with the Windows Forms programming model, giving additional consideration to the development cycle and runtime requirements of the Web. The following sections describe these new features in greater detail.

## Generating Local Resources

.NET resources make it possible to selectively replace content for specific languages and cultural regions so that the same code base can support multiple cultures. But there has been little incentive for developers to invest in this path for Web applications since generating resources for Web pages required significant manual effort. ASP.NET 2.0 provides a simple way to automatically generate resources for Web pages while supporting a complex assortment of content agents including HTML elements and attributes, static content, and server controls.

To generate resources for a particular Web Form, select **Generate Local Resource** from the **Tools** menu in Visual Studio 2005. Local resources can be created for any page open in Design View (that includes Web Forms, user controls, and master pages). This step automatically generates a default set of XML resources (.resx) for the page, and places them in a dedicated subdirectory for local resources named \App_LocalResources.

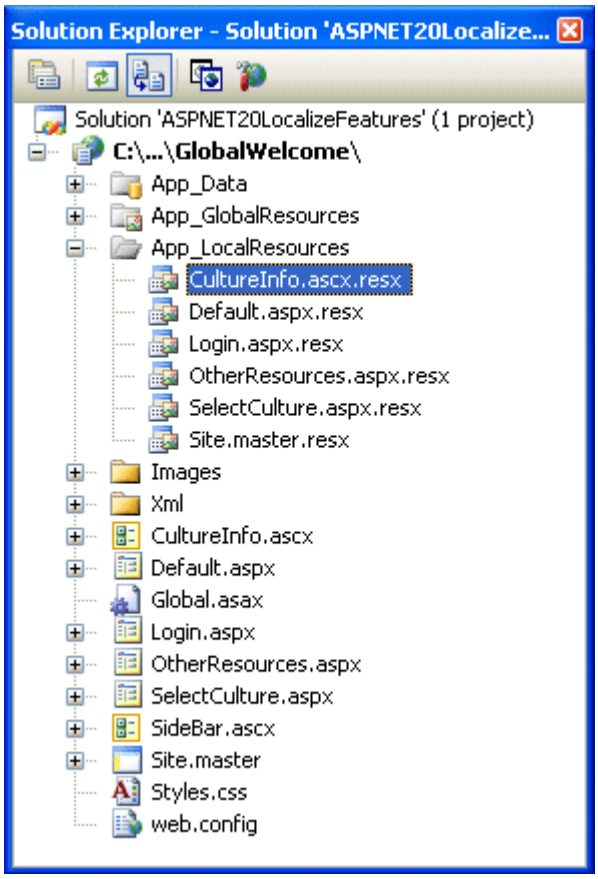


**Figure 2. Web Forms, user controls, and master pages can all be internationalized from within Visual Studio 2005. This particular view shows default resources generated for several pages.**

Local resources are stored in .resx files whose naming convention links it to the page they service (see Figure 2). For example, after generating resources for a master page named site.master, a new .resx file appears in \App_LocalResources named site.master.resx. For a Web Form named default.aspx, the generated resource is named default.aspx.resx.

The Visual Studio designer populates these resources by inspecting the page and its controls (ASP.NET server controls, custom controls, and HTML controls with **runat="server"**)—looking for properties marked with the **LocalizableAttribute**:

**C#**

```
[Localizable(true)] public string Text { get {...} set {...} }
```

**VB.NET**

```
<Localizable(true)> Public Property Text() As String Get ... End Get
```

```
Set (ByVal value As String) ... End Set }
```

Localizable properties for each server control are automatically pushed to resources, with each property identified by a unique key. This resource key includes a prefix that identifies the control and the property name. Each property value is set to the control's default unless values are specified within the control declaration. The following **LinkButton** declaration specifies a value for **PostBackUrl** and **Text** properties in its declaration:

```
<asp:LinkButton id="lnkSelectCulture" PostBackUrl="selectculture.aspx"
runat="server" Text="Language Preference"></asp:LinkButton>
```

Figure 3 shows the resource entries generated for this **LinkButton** after generating local resources for its host page (in the sample project, see cultureinfo.ascx). Resources are stored in the resource file (see cultureinfo.ascx.resx) and exclude non-localizable properties such as **PostBackUrl**, by default.

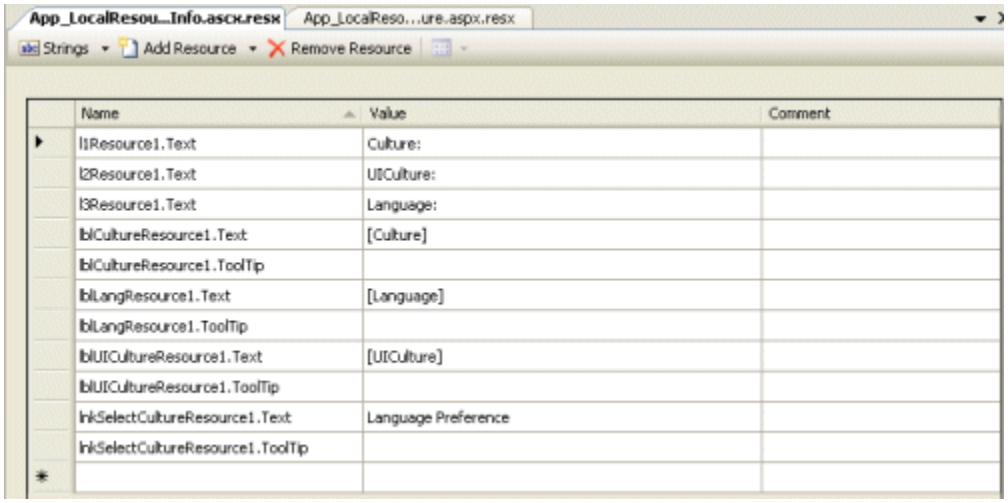


**Figure 3. Local resources are generated based on localizable properties for ASP.NET server controls, custom server controls, and HTML controls that are run on the server (runat="server")**

When resources are generated, the control declaration is also modified to declaratively associate properties with resource entries. Several new declarative expressions are recognized by the page parser and they trigger code generation to populate control properties with resource values at runtime. In other words, declarative expressions save you from typing the code.

Declarative localization expressions are a new construct to ASP.NET 2.0, resembling data binding statements but specifically designed for accessing resources. These expressions come in two forms: implicit and explicit. Implicit expressions are automatically inserted when local resources are generated,

and they support mapping multiple resource entries to a set of control properties in a single declaration. Explicit expressions are declarations developers can add to further control page localization.

The following implicit expression, identified by **meta:resourcekey**, was generated for the previous **LinkButton** declaration, and now specifies the resource key prefix for this control:

```
<asp:LinkButton id="lnkSelectCulture" PostBackUrl="selectculture.aspx"
runat="server" Text="Language Preference"
meta:resourcekey="lnkSelectCultureResource1" ></asp:LinkButton>
```

Resource entries for control properties that use the prefix **lnkSelectCultureResource1** will be automatically mapped to the appropriate control property at runtime. The value for each property in the control declaration remains untouched, including those values pushed to resources. These default values will appear in Design View to give context to the control within the page. When the page is parsed the localization expression is used to generate code that applies resources to control properties. Developers need not write code to instantiate a **ResourceManager** to access these local resources at runtime.

At runtime resource values take precedence but default values appear in Design View, even in the absence of resources. This can be very helpful to Web designers, allowing them to view pages with content at design time. Localized properties that appear in the Properties Window are also marked with special icons indicating that their values are drawn from resources. Figure 4 shows an ASP.NET **ImageButton** control that some properties are bound to global resources (indicated in blue) and others to local resources (indicated in red). These icons also have tooltips indicating that the resource is bound. Property binding through expressions will be discussed in greater detail in a later section.

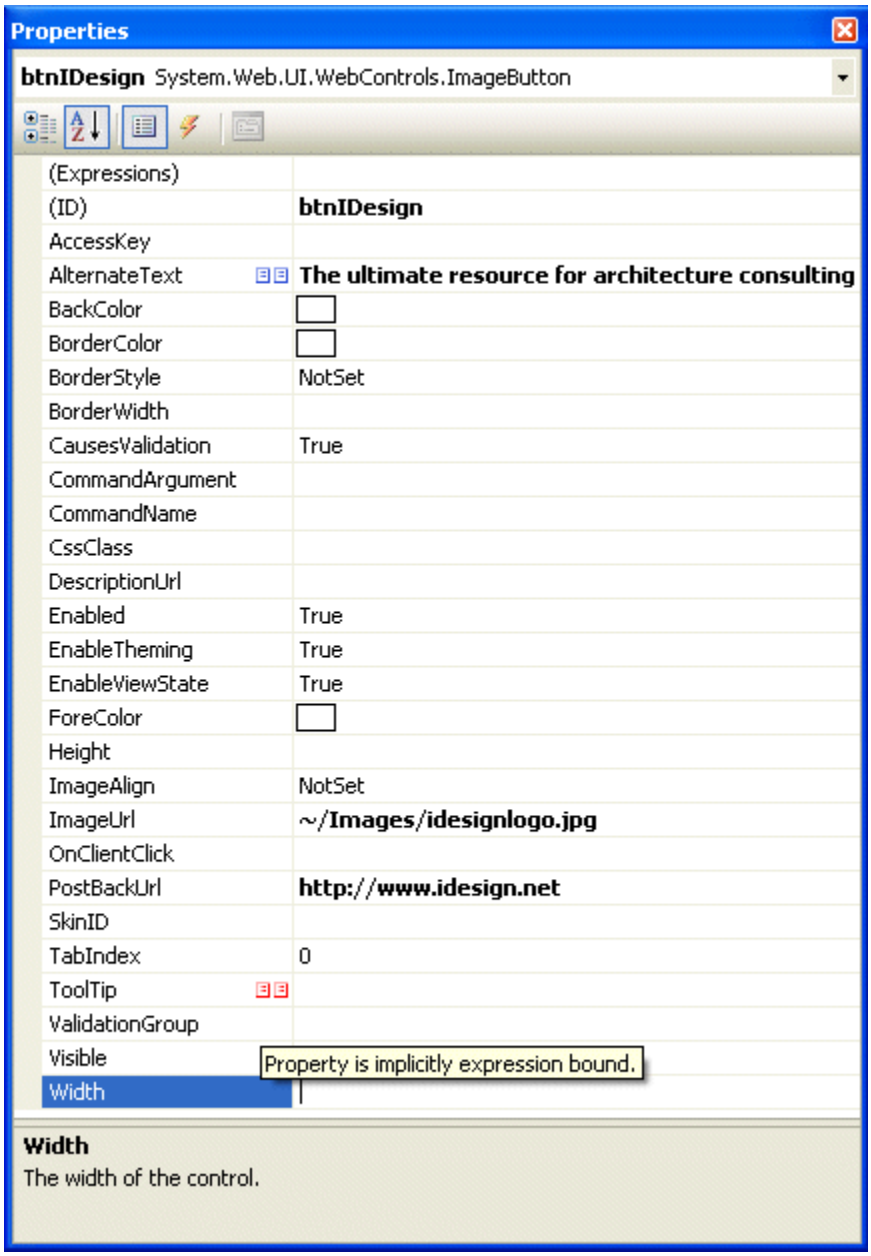


**Figure 4. Localized properties bound to localization expressions are easily spotted by a special icon shown in the Properties Window view.**

The discussion so far focused on generating local resources, which by default generates implicit localization expressions and resources for localizable control properties. It is also possible to bind properties to shared application resources. In addition, you can automate the generation of local resources for non-localizable control properties, other HTML elements, and static content. This is achieved by applying explicit localization expressions to those page and control elements. In other words, developers can declaratively indicate what additional content should be localized so that it can be included with

resource generation for the page, or selectively draw from an alternate source.

## Global Application Resources

Automatic generation of local resources for each page can lead to duplicate entries for common terms and redundant translation efforts. Fortunately, ASP.NET 2.0 has an intrinsic reuse model with master pages and user controls so that headings, menus, sidebars, and other sections of HTML can be shared among Web forms. The fact that each master page, user control and Web form owns its own set of local resources reduces resource duplication. Still, resource entries such as glossary items, error messages, and functionality drivers such as directional attributes can be useful when consolidated and shared among all pages, or even reused by multiple applications.

While local resources are generated automatically for each page through the designer, global application resources are created manually. That means adding a new application resource (.resx) to the solution and placing it in a dedicated directory specifically for global resources, \App_GlobalResources. This is consistent with the method for generating resources for 1.x applications; however, a new resource editor simplifies creating and editing those resource entries, and global resources are now strongly-typed with Intellisense support which is very convenient during development. Global resources, like local resources, participate in the new page parsing and runtime model for ASP.NET 2.0. They can be bound to control properties using explicit localization expressions and a **ResourceManager** is automatically instantiated and cached, removing the need for developers to manage this lifecycle to access resources at runtime.

Visual Studio 2005 includes a new Managed Resource Editor shown in Figure 3 and Figure 5. Figure 3 illustrates the resource strings editor, similar to the resource editor in Visual Studio 2003. Figure 5, on the other hand, illustrates the list of pre-defined resource categories supported by the editor, including strings, icons, and other image file types, audio file types, and other files including XML. Outside of these predefined types supported by the resource editor it is also possible to programmatically insert other complex types into resources. The underlying resource file is XML-based.

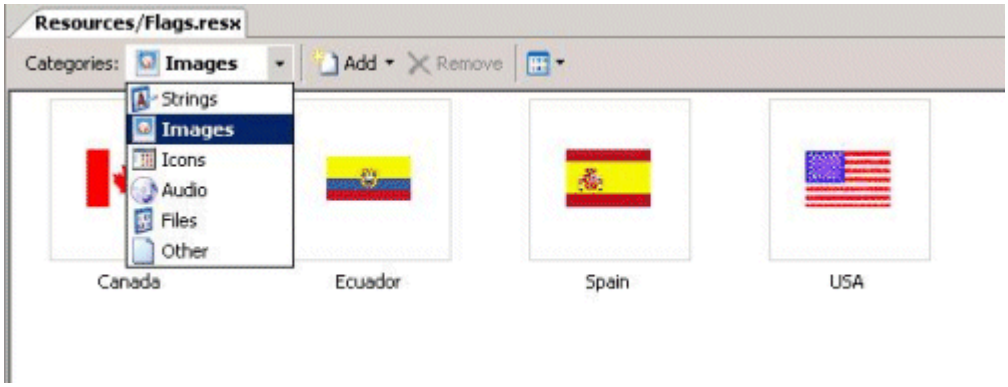

**Figure 5. The new Managed Resource Editor provides integrated support for numerous predefined data types, and supplies alternate views of resource data including thumbnail view as shown here for the Images resource category.**

When file-based resources such as images, sound, and XML files are inserted through the editor, they are defined as **ResxFileRef** entries by default. For example, the following describes resource entries created for an external image and XML file, respectively:

```
<data name="Spain" type="System.Resources.ResXFileRef,
System.Windows.Forms">
<value>..\Images\Spain.gif;System.Drawing.Bitmap, System.Drawing,
Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a</value> </data> <data
name="supportedCultures" type="System.Resources.ResXFileRef,
System.Windows.Forms">
<value>..\Xml\supportedCultures.xml;System.String, mscorlib,
Version=2.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089;Windows- 1252</value> </data>
```

The resource compiler resolves each file reference, and the runtime handles loading the referenced file into a stream, converting it to the correct data type. In the case of images, the type is **System.Drawing.Bitmap**, and XML is returned as a **System.String** by default. For pre-compiled Web applications, resources are embedded in output assemblies. That means the actual files need not be deployed with the site. Consistent with the new compilation model for ASP.NET 2.0 (see this article for more details), resources and related files can also be deployed for full runtime compilation.

One of the noticeable benefits of global resources is that they are compiled into a strongly-typed class that makes entries accessible directly by their resource key. For example, a global application resource named Flags.resx is accessed as runtime type **Resources.Flags**. Intellisense is available through the intrinsic **Resources** type, and if a type converter is available resource items are returned as their native data type. In the case of Images, the data type is **System.Drawing.Bitmap**, as shown in Figure 6.

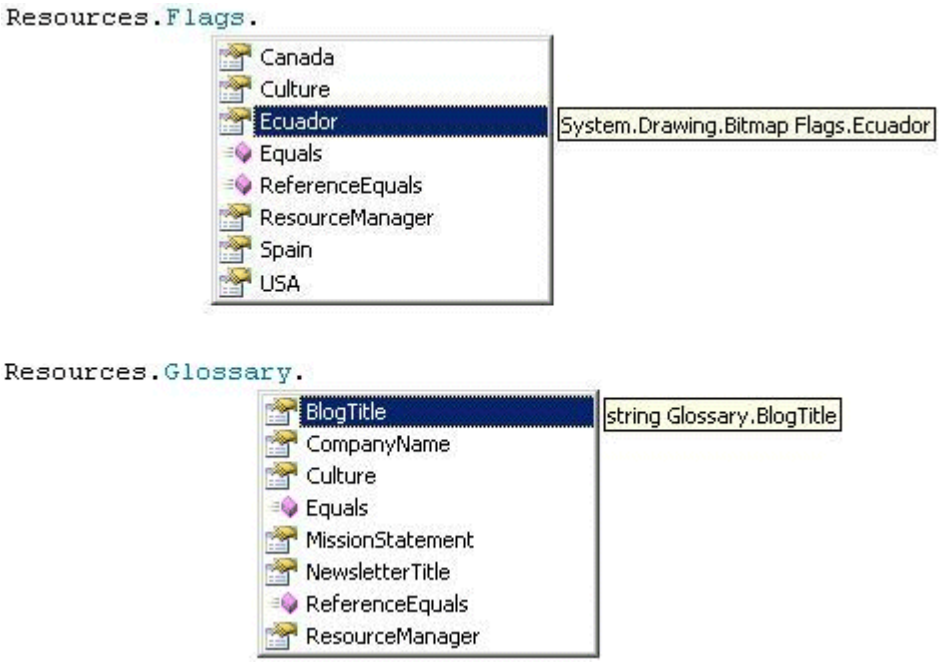


**Figure 6. Global resources are compiled into strongly-typed resources and are accessible through the intrinsic Resources object.**

Image-based resources can be useful to Web control developers for encapsulating embedded graphics used by the control, but are more generally useful to Windows Forms developers that can readily consume the binary image format for display. Regardless, strongly-typed resources and Intellisense increase productivity for accessing resources at runtime. For example, Figure 6 also demonstrates how a strongly-typed Glossary resource makes it easy to retrieve named string values.

Both local and global resources can be accessed declaratively or programmatically in order to generate localized content. These techniques will be discussed in the following sections.

## Implicit Localization Expressions

As mentioned earlier, when local resources are generated it triggers modifications to control declarations on the page. The default behavior is for an implicit localization expression to be added to server controls, indicated by the parse-time attribute **meta:resourcekey**.

```
<asp:LinkButton id="lnkSelectCulture" PostBackUrl="selectculture.aspx"
runat="server" Text="Language Preference"
meta:resourcekey="lnkSelectCultureResource1" ></asp:LinkButton>
```

This expression specifies the expected prefix for all resource entries related to the control's properties—thus the term *implicit expression*. Automatic resource generation only considers localizable properties, but

in reality any resource entry using this prefix followed by a valid property name will be bound to that property in the compiled page code. In the above example, **lnkSelectCultureResource1** is the resource key prefix, and resource entries shown in Figure 3 use this prefix for each property that applies to the same control instance (that is, **lnkSelectCultureResource1.Text**).

This declarative statement (**meta:resourcekey**) indicates to the ASP.NET page parser that it should generate code to retrieve property values from default local resources. The resulting code ultimately uses runtime methods for accessing resources, with the help of **GetLocalResourceObject**. In the sample code provided, the compiled cultureinfo.ascx page contains the following code that creates and initializes the **LinkButton**, **lnkSelectCulture**:

```
LinkButton button1 = new LinkButton(); base.lnkSelectCulture = button1;
button1.ApplyStyleSheetSkin(this.Page); button1.ID =
"lnkSelectCulture"; button1.PostBackUrl = "selectculture.aspx";
button1.Text = "Language Preference"; button1.Text =
Convert.ToString(base.GetLocalResourceObject(
"lnkSelectCultureResource1.Text"), CultureInfo.CurrentCulture);
button1.ToolTip = Convert.ToString(base.GetLocalResourceObject(
"lnkSelectCultureResource1.ToolTip"), CultureInfo.CurrentCulture);
```

Implicit localization expressions are applied to all server control declarations when local page resources are generated. You can suppress this behavior by indicating that the control should not be localized using this alternate syntax:

```
<asp:LinkButton id="lnkSelectCulture" PostBackUrl="selectculture.aspx"
runat="server" Text="Language Preference" meta:localize="false" >
</asp:LinkButton>
```

Individual control properties that are not considered important to your localization strategy can be manually removed from local resources using the Managed Resource Editor shown in Figure 3 (remember, all localizable properties are pushed to resources by default). This reduces the amount of code generated by the page parser to apply local resource entries to control properties, since it will only reflect those entries present in the page's default local resources. This also means that additional key values added to localized resources will not be applied at runtime since no code is generated to do so.

Localization expressions such as this leverage the new Expression Builder feature to be released with the .NET Framework 2.0. Implicit localization expressions save developers time by generating code from a single declaration to populate all localizable server control properties. This removes a manual step currently required for 1.x Web applications where typically custom data binding statements or custom code is required to pull values from a localized resource or data source.

The next sections will review how additional resource generation can be automated for the localization of

static text and specific control properties including those that are not marked by **LocalizableAttribute**.

## Explicit Localization Expressions

Although it is very handy to automatically generate resources for localizable control properties, developers need a solution that will also support localizing specific property values and other blocks of content. Explicit localization expressions provide a way to declaratively assign specific resource entries to server control properties and other HTML elements. For example, the following **ImageButton** control declaration uses an explicit expression to set its **AlternateText** property:

```
<asp:ImageButton id="btnIDesign" Runat="server"
ImageUrl="~/Images/idesignlogo.jpg" AlternateText='<%$ Resources:
MissionStatement %>' PostBackUrl="http://www.idesign.net" />
```

Explicit localization expressions follow this syntax:

```
<%$ Resources: [resourceType], resourceKey%>
```

In the above example, **resourceType** is omitted, which means the value for **MissionStatement** is drawn from local page resources. This removes the need for the implicit assignment of bulk resource values to control properties. The **resourceKey** value indicates the resource entry to pull. More importantly, you may have additional resource entries that are not associated to a specific control property that you want to apply declaratively.

Explicit expressions can also be generated through the Properties Window using the Expressions dialog box shown in Figure 7. This dialog supports creating explicit expressions that bind control properties to local or global resources.

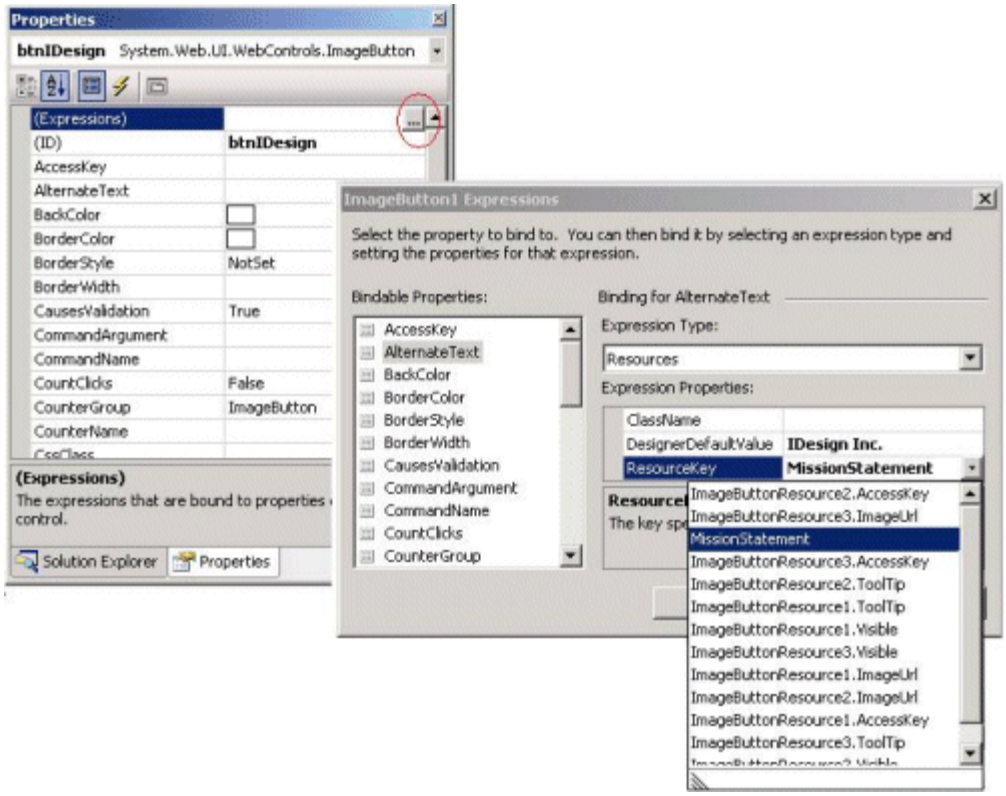


**Figure 7. Using the Expressions dialog developers can intuitively map resources to properties and generate explicit localization expressions**

With this dialog box, developers create explicit expressions through the Properties window just as they would set other control properties. If the **ClassName** is omitted from Expression Properties (see Figure 7), a dropdown list of available local resource keys is presented (assuming those resource entries have been created). Once again, this reduces the amount of work for developers in creating resource entries and generating code to populate properties at runtime.

**Note** Explicit expressions cannot be created for those properties that already have resources mapped through implicit expression. To access shared resources, the **ClassName** should indicate the name of a valid resource file in the global resources directory. Unlike local resources, explicit expressions targeting global resources do not automatically generate those resources. Luckily, the IDE has provided a little help for generating global resources. When you add a new Assembly Resource File to the project, you will be prompted to place it in the global resource directory, \App_GlobalResources. The directory will be created for you if it does not yet exist for the project. You can also hook into extensibility features of the IDE to further automate interaction with global resources, but that goes beyond the scope of this article.

Implicit and explicit localization expressions can be mixed so that specific properties are pulled from shared resources, while other properties are pulled from local resources. The following example shows the same **ImageButton** control declaration retrieving its **AlternateText** from a shared resource, Glossary.resx, while all other properties are drawn from the local resource using implicit syntax:

```
<asp:ImageButton ID="btnIDesign" Runat="server"
```

```
ImageUrl="~/Images/idesignlogo.jpg" AlternateText='<%$
Resources:Glossary, MissionSatatement%>'
PostBackUrl="http://www.idesign.net"
meta:resourcekey="ImageButtonResource1" /></td>
```

The use of explicit localization expressions makes it possible to exercise granular control over which properties are localized, from where. In the case where a control declaration includes both explicit and implicit expressions, page resources are generated as follows:

- Properties bound by explicit expressions that specify a shared resource are omitted from local resource generation.
- Properties bound by explicit expressions for local resources generate an entry for **resourceKey** if it does not yet exist.
- Remaining localizable properties that are not bound by explicit expressions generate an entry for **resourceKey.propertyName** with the value specified in the control declaration, or the control's default property value.

As mentioned earlier, when the page is parsed, code is automatically generated from these declarations to draw property values from the local or global resources. In the above example, the resulting code creates an **ImageButton** control and sets its **AlternateText** property from the global Glossary resource, and remaining localizable properties are set by local page resources:

```
ImageButton button1 = new ImageButton(); button1.ID = "btnIDesign"; //
other initialization code button1.AlternateText = (string)
base.GetGlobalResourceObject("Glossary", "MissionStatement");
button1.ImageUrl = (string)
base.GetLocalResourceObject("ImageButtonResource1.ImageUrl");
button1.ToolTip = (string)
base.GetLocalResourceObject("ImageButtonResource1.ToolTip");
button1.Visible = (bool)
base.GetLocalResourceObject("ImageButtonResource1.Visible",
typeof(Control), "Visible");
```

**Note** You should always use implicit expressions to map local page resource entries to control properties. This includes non-localizable properties for which entries can be manually generated. Use explicit expressions to specify global resource values.

## Localizing HTML Elements and Static Text

Generating resources for server control properties is made easy with implicit and explicit localization expressions. But, to prepare a page for localization, consideration must also be given to other content such as the HTML page title, directional attributes, and static content. Localization expressions can also be applied to the **@Page** directive and other sections of HTML to declaratively indicate other areas for localization prior to generating page resources.

### HTML Controls

HTML Controls cannot participate in the benefits of implicit or explicit expressions unless they are run on

the server (**runat="server"**). Once marked as server controls, local resources are automatically generated for the control's localizable properties. Consistent with ASP.NET and custom server controls, HTML server controls can also be bound to implicit or explicit expressions, the latter of which can be generated through the Expressions dialog mentioned earlier.

HTML elements found in the page header can also be declaratively bound to resources, which can be useful for page titles and style sheet links. In fact the HTML page title element is special because it is also a **Page** property that can be set through the **@Page** directive. By default, an implicit expression is assigned to each page when local resources are generated:

```
<%@ Page Language="C#" CodeFile="Default3.aspx.cs" Inherits="_Default"
Culture="auto" meta:resourcekey="PageResource1" UICulture="auto" %>
```

A resource is generated with the key **PageResource1.Title**, using the value of the **<title>** element as default. The **<title>** element can override this implicit resource mapping using an explicit expression. The following example demonstrates a page title that draws from global resources instead of local page resources:

```
<head runat="server"> <title><asp:Literal runat="server" text='<%$
Resources: PageResource1.Title %>' /></title> </head>
```

Normally, the default page title will be held in the master page, and content pages will (or, should) only override this setting if your site demands a custom title for every page.

### Directional Attributes

Improved support for directionality settings has been added through the provision of a new **Direction** property, supported by controls such as the **<asp:Panel>**. Using a shared resource that indicates overall directionality for the application, based on culture, default **"LTR"** direction can be indicated in a global resource, and overrides to this value based on culture can specify **"RTL."**

Localization resources are not supported in the <html> tag, however you can supply a rendering statement that draws resources from the appropriate resource type as shown here:

```
<html dir='<% =GetLocalResourceObject("Direction")%>' > ... </html>
```

A panel can also be used to affect directionality of contained controls:

```
<asp:panel runat="server" direction='<%$ Resources: Direction %>'> ...
</asp:panel>
```

For more information related to directional support in Visual Studio .NET please visit

http://www.microsoft.com/middleeast/msdn/arabicsupp.aspx#22.

### Static Text

Localization expressions are useful for setting control properties and other HTML elements; however, many Web pages being readied for localization already contain significant blocks of static content intermixed with ASP.NET controls. A new ASP.NET **Localize** control is provided to mark static content as localizable, so that it can be included in resource generation. If a **meta:resourcekey** is specified prior to issuing the command to generate resources, the key specified in the control will be used as the prefix (this applies to other controls as well):

```
<asp:Localize id="welcomeContent" runat="server"
meta:resourcekey="welcome">Welcome!</asp:Localize>
```

In this example, a new local resource entry is generated for the **Text** property of the **Localize** control, with the resource prefix **"welcome"** (**welcome.Text**). To explicitly populate static text from a global resource, the **Text** property can be assigned by an explicit localization expression:

```
<asp:Localize id="welcomeContent" runat="server" text='<%$ Resources:
Glossary, welcomeText%>'>Welcome!</asp:Localize>
```

Consistent with other scenarios, these declarative statements are parsed into code that requests resources to set control properties, in this case the **Text** property. Any HTML markup that appears within the declaration of this control will be included in the resource generation, which can complicate the translation process so it is best to avoid including markup.

The significance of the **Localize** control over its base class, the **Literal** control, is that it is treated like a **Literal** control at runtime; however, the designer ignores it and allows developers to directly edit static content in Design View (unlike the **Literal** control, which is bound by a container in Design View).

## Resource Localization and Deployment

New declarative statements, automatic generation of page resources, and the presence of strongly-typed global resources collectively make it much easier for developers to prepare Web applications for localization. Default resources stored in the \App_LocalResources and \App_GlobalResources directories (local and global resources, respectively) can be translated to supported cultures, their translated versions copied to the same directory as its source. Naming conventions for translated resources follow the same rules as with 1.x. The culture code is part of the translated .resx filename. Figure 8 shows the expanded view of a sample project with page resources translated to Spanish, French, and Italian.

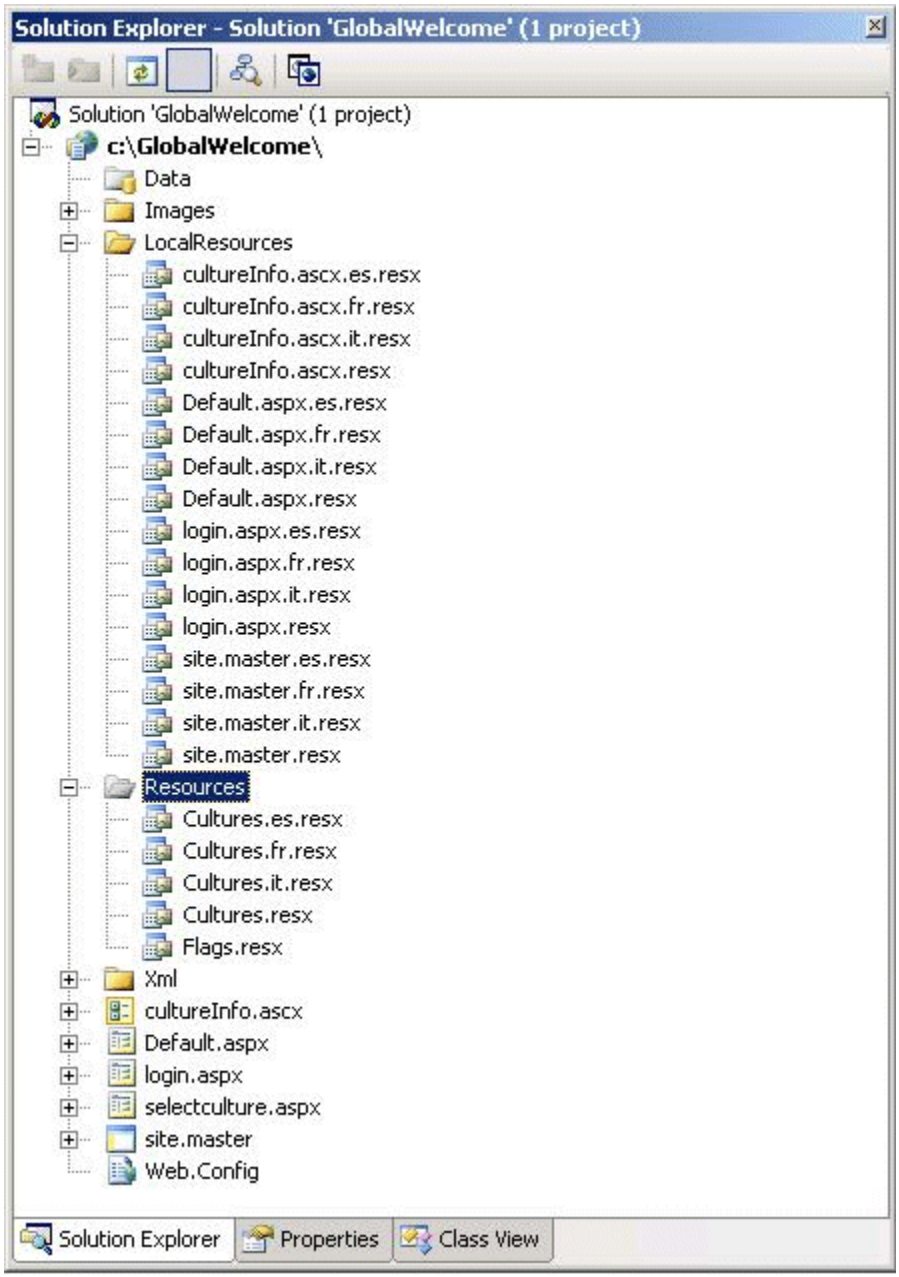


**Figure 8. Consistent with 1.x resource naming conventions, .resx files should be named according to culture code.**

.NET Framework 1.x applications are typically shipped with localized satellite assemblies, but a new set of deployment options exists for ASP.NET 2.0:

1. The ASP.NET 2.0 runtime continues to support the traditional deployment model where the \bin directory contains local assembly dependencies, and subdirectories per supported culture contain satellite (resource-only) assemblies.
2. ASP.NET 2.0 supports site pre-compilation which treats Web pages (that is, pages, user controls, and master pages), their local resources, and their code separation files as a single unit,—

compiling them into deployable assemblies. Global resources are also pre-compiled into assemblies. Any referenced resource entries, such as images or XML files, are also compiled into assembly components. Simply put, the pre-compilation of an ASP.NET 2.0 Web application becomes a binary drop that provides a higher level of protection for source files through encapsulation.

3. ASP.NET 2.0 also supports dynamic runtime compilation for all source files. That means that Web content, supporting code files, other source code, resources (local and global), and other supporting application files can all be deployed in raw source format. The runtime compiler is then responsible for parsing pages and generating assemblies on the fly through a build provider model. This model provides the greatest amount of flexibility, but is primarily useful only for lightweight Web applications that are frequently changing, and do not require a higher level of source-code security and assembly version control.

In the case of options 2 and 3 the ASP.NET 2.0 compiler automatically compiles .resx source files found in the \App_LocalResources and \App_GlobalResources directories into assemblies. In the case of option 3, local resources are compiled prior to page compilation, the first time the page is accessed, or when the page source or its .resx file has been modified. Because there is a build provider for .resx, they can be deployed without a compile step.

As mentioned in option one, backward compatible support for 1.x assemblies, neutral resources, and associated satellite assemblies still exists. This deployment model does is not compatible with localization expressions, nor are these resources accessible through the default runtime resource provider model. If this is a required feature, you can leverage custom expression builders and/or resource providers to integrate alternate resource sources, including 1.x resource assemblies.

## Runtime Resource Providers

To work with a **ResourceManager** in ASP.NET 1.x applications, it was necessary to instantiate the **ResourceManager** in code, prior to the execution of data binding statements or other mechanisms that retrieved values from resources. ASP.NET 2.0 automatically instantiates resource managers as needed, when local and global resources are accessed, removing the need to write code to manage this process. By using declarative statements to populate control properties and HTML elements with resource values, no code need be written to generate a fully localized page!

A default **ResourceProviderFactory** is responsible for instantiating resource providers for local and global resources at runtime, and at design time through the associated designer factory. Developers can write code to access values in local resources using **GetLocalResourceObject**, exposed through the **Page** object:

```
if (this.Context.User.Identity.IsAuthenticated)      mnuLogin.Text =
GetLocalResourceObject("Login"); else      mnuLogin.Text =
String.Format(GetLocalResourceObject("LogoutUser",
this.Context.User.Identity.Name);
```

This can be used to override declarative localization expressions when runtime decisions are necessary for

resource-dependent HTML output. To access global resources, the **Page** object exposes a different method:

```
string cultures =
(string)this.GetGlobalResourceObject("supportedCultures");
```

For global resources, it is more likely developers will leverage Intellisense and strongly-typed resource access, shown here retrieving an XML resource stored as a **String** type:

```
string cultures = Resources.Cultures.supportedCultures;
```

Ultimately, **GetLocalResourceObject** and **GetGlobalResourceObject** gain access to the appropriate resource through the configured resource provider. Therefore, if a custom resource provider is created to access resources in alternate stores, these methods would be able to retrieve from those stores through **GetGlobalResourceObject**.

## Preferred Culture Selection

Traditionally with 1.x ASP.NET applications, two key approaches were considered for the selection of culture for each request. For applications that duplicated the entire site in a culture-specific subdirectory, the **<globalization>** element in web.config indicated which culture and UI culture should be used by the runtime thread requesting resources from those subdirectories:

```
<system.web> <globalization culture="es-ES" uiCulture="es">
</system.web>
```

For applications that used a single code-base, these declarative settings are not useful. Instead, runtime code is necessary to manually set the culture for each request thread, so that the culture can be dynamically chosen for each user making the request. The user's preferred culture can be collected from a database profile, from an HTTP cookie, or from the Web browser's language settings. In either case, setting the request thread's UI culture determines which localized resources the **ResourceManager** will draw from at runtime, and the culture setting affects culture-aware formatting as mentioned earlier.

ASP.NET 2.0 introduces a new feature to automate culture selection at runtime based on Web browser preferences. The **@Page** directive, which also supports culture and UI culture settings, can be used to indicate that a particular page should be run based on the browser preferences:

```
<%@ Page UICulture="auto" Culture="auto">
```

The web.config file for the application can also apply this setting for the entire application:

```
<system.web> <globalization culture="auto:en-US" uiCulture="auto:en">
```

```
</system.web>
```

The colon after **auto** allows you to specify a default culture in the event HTTP headers are not available. Since culture must be set to a specific culture, the example above shows **en-US** as the culture.

The end result is that the runtime automatically detects the **ACCEPT_LANG** header sent by the browser, and sets the thread very early in the page lifecycle to the first language in the user's language preference list. If a profile is also stored for application users, or if users can select a specific culture through the site, developers must write code to override the **auto** setting handled by the runtime. One technique for this is demonstrated in the sample code.

## Conclusion

It is imperative to define a globalization strategy early in the development lifecycle, in order to more quickly accommodate demands for future product releases that can reach global markets. These new localization features for ASP.NET 2.0 will make it easier for developers to follow through on a strategy to localize an application, while reducing the overhead of the development cycle. Automatic generation of page resources and a new Managed Resource Editor make creating resources for Web applications more natural. Declarative localization expressions make it possible to ensure all necessary page elements are localized and assist with the automatic generation of appropriate page resources. A new runtime model means developers can have the option to no longer compile satellite assemblies, instantiate resource managers, and set the request thread according to culture. Ultimately, this makes for a strong case in shipping Web applications with a single code base, which ultimately reduces costs and efforts associated with delivering a localized solution.

## Additional Resources

Resource Fallback Process

Developing World-Ready Applications

Globalization Architecture for ASP.NET 1.x

ASP.NET Directional Support

ASP.NET 2.0 Internals

My web log: www.dasblonde.net (RSS on Globalization)

IDesign

---

**About the author**

Michèle Leroux Bustamante is Chief Architect of IDesign Inc., Microsoft Regional Director for San Diego, Microsoft MVP for XML Web Services and BEA Technical Director. She has over a decade of development experience development applications with VB, C++, Java, C# and VB.NET and working with related technologies such as ATL, MFC and COM. At IDesign Michele provides training, mentoring and high-end architecture consulting services, focusing on Web services, scalable and secure architecture design for .NET, interoperability and globalization architecture. She is a member of the International .NET Speakers Association (INETA), a frequent conference presenter, conference chair of SD's Web Services track, and is frequently published in several major technology journals. Michele is also Events Director for IASA (International Association of Software Architects), Web Services Program Advisor to UCSD Extension. Her book, Windows Communication Framework (O'Reilly) is available online at www.thatindigogirl.com, and will be published in 2006. Reach her at mlb@idesign.net, or visit www.idesign.net and www.dasblonde.net.

Top of Page