Manifestressourcen und Ressourcen in .NET

Das ganze Drumherum

Über Ressourcen kann der Entwickler externe Informationen in ein Programm aufnehmen. Er kann Icons und Bilder einbinden, Texte in Textdateien vorhalten und weitere Metainformationen speichern. Das .NET Framework bietet eine gute Unterstützung für Ressourcen. dotnetpro fasst zusammen, was Entwickler über Ressourcen wissen müssen.

Zu Zeiten althergebrachter Windows-Entwicklung wurden Ressourcen als *rc*-Dateien bearbeitet, zu einer *res*-Datei übersetzt und durch den Linker in die *exe*- oder *dll*-Datei gepackt. Die Syntax und der Sprachumfang innerhalb der *rc*-Dateien waren fixiert. Neue Arten von Ressourcen waren nur über Umwege und Verrenkungen in dieses System einzubringen.

Mit dem .NET Framework hat sich das geändert. Es gibt jetzt zwei Arten von Ressourcen: Manifestressourcen und Ressourcen – ohne qualifizierenden Zusatz –, wobei Letztere auf Manifestressourcen aufbauen. Die Dokumentation ist insofern vollständig, als alle Details im Rahmen der Referenz beschrieben werden. Einen zusammenfassenden und einigermaßen erschöpfenden Überblick sucht man aber vergebens. Die beiden Begriffe werden in der Dokumentation außerdem nicht konsequent unterschieden. Wenn dort von Ressourcen die Rede ist, können je nach Kontext auch Manifestressourcen gemeint sein. In diesem Artikel wird darauf geachtet, die Begriffe konsequent zu unterscheiden.

Manifestressourcen sind beliebige Dateien, die ihren Weg in die Assembly finden. Sie werden dort zur Entwicklungszeit als Teil des Build-Prozesses eingebaut und können zur Laufzeit als generischer Stream ausgelesen werden. Eine Assembly kann dabei mehrere Manifestressourcen beinhalten. Aus diesem Grund haben die einzelnen Manifestressourcen eindeutige Namen.

Auf einen Blick

Autor

Als Manager Technology der SDX Software Development Experts AG verantwortet **Alexander Jung** die Evaluation und Implementierung neuer Technologien. Weiterhin ist er seit vielen Jahren für Enterprise-Kunden als Architekt und Consultant in komplexen Software-Entwicklungsprojekten tätig.

dotnetpro.code
A0406Ressourcen

Sprachen C#

Technik Ressourcen und Manifestressourcen

Voraussetzungen Visual Studio .NET

Serie
1. Ressourcen in .NET
2. Ressourcen und Globalisierung

Manifestressourcen auslesen

Die Klasse *Assembly* bietet drei Methoden im Zusammenhang mit Manifestressourcen. *Assembly.GetManifestResourceInfo()* ist relativ uninteressant. Wichtiger sind die beiden anderen Methoden:

- *Assembly.GetManifestResourceNames()* liefert die Namen der in der Assembly enthaltenen Manifestressourcen als *string[]*.
- *Assembly.GetManifestResourceStream(...)* liefert die angegebene Manifestressource als *Stream*. Einerseits stehen mit dieser Methode alle Möglichkeiten offen, die das .NET Framework im Zusammenhang mit Streams bietet, andererseits hat man selbst die Verantwortung, den Inhalt der Manifestressource korrekt zu interpretieren.

Listing 1 zeigt, wie man eine Textdatei ausliest, die als Manifestressource unter der Bezeichnung *ResTest.Res.Textfile1.txt* in der Assembly hinterlegt ist.

Listing 1

Lesen einer Textdatei als Manifestressource.

```
Assembly a= GetType().Assembly;
Stream strm=
   a.GetManifestResourceStream
   ("ResTest.Res.TextFile1.txt");
StreamReader sr= new StreamReader(strm);
string s= sr.ReadToEnd();
sr.Close();
```

Einige Klassen bieten Abkürzungen zu diesem Verfahren an. *System.Drawing.Icon* zum Beispiel bietet einen Konstruktor mit *Stream*-Argument und kann somit einen solchen Stream direkt verwenden, wie es Listing 2 demonstriert.

Listing 2

Lesen eines Icons als Manifestressource.

```
Assembly a= GetType().Assembly;
Stream strm= a.GetManifestResourceStream
   ( "ResTest.Res.logo.ico");
System.Drawing.Icon i= new Icon(strm);
```

Falls die entsprechende Manifestressource nicht existiert, liefert *Assembly.GetManifestResourceStream(...)* einfach *null* zurück.

Manifestressourcen anlegen

Wie werden aus Dateien Manifestressourcen? Grundsätzlich geschieht dies beim Bau der Assembly durch den Assembly Linker (*al.exe*). Mit der Option */embed* und ihren Parametern wird eine angegebene Datei in die Assembly eingebettet. Mit */link* kann man sie zu der Assembly hinzufügen und dabei als eigen-

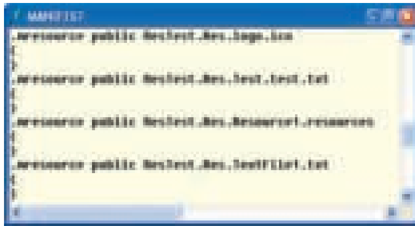


Abbildung 1 ildasm.exe spürt Manifestressourcen in Assemblies auf.

Tabelle I

Beispiele für Namen von Manifestressourcen.

Dateiname	Name der Manifestressource
Res\logo.ico	ResTest.Res.logo.ico
Res\Test\test.txt	ResTest.Res.Test.Test.txt
Res\1. Copy of Test\1. Copy of test.txt	ResTest.Res._1._Copy_of_Test.1. Copy of test.txt
Res\Copy (2) of Test\Copy (2) of test.txt	ResTest.Res.Copy__2__of_Test.Copy (2) of test.txt

ständige Datei bestehen lassen. Beiden Optionen kann man neben der Datei selbst auch den Namen der Manifestressource übergeben. Außerdem kann man die Manifestressource zu einer privaten, das heißt für andere Assemblies unsichtbaren Information erklären.

Der Namensteil „Manifest“ in „Manifestressource“ erklärt sich durch die Tatsache, dass die Namen der Ressourcen im Manifest abgelegt werden. Abbildung 1 zeigt einen entsprechenden Ausschnitt IL-Code, den man sich leicht mit *ildasm.exe* anschauen kann.

Integration in Visual Studio .NET

In Visual Studio .NET reduziert sich der Aufwand auf eine Einstellung im Solution Explorer. Nachdem man die entsprechende Datei in das Projekt aufgenommen hat, muss man lediglich in den Dateieigenschaften die Einstellung *Build Action* auf den Eintrag *Embedded Resource* setzen. Gelegentlich sind Sonderbehandlungen nötig, dazu später mehr. Visual Studio .NET fügt die Datei dann automatisch unter einem nach festen Regeln gebildeten Namen als public embedded Manifestressource in die Assembly ein.

Der Name der Manifestressource wird nach einer einfachen Konvention gebildet. Er entspricht im Allgemeinen dem Muster *<DefaultNamespace>.<Unterverzeichnisse>.Dateiname*. Der Default Namespace wird den Projekteinstellungen entnommen (*Projekteinstellungen/Common Properties/General*) und ist üblicherweise identisch mit dem Namen der Assembly. Mehrere Unterverzeichnisse werden nicht durch den Schrägstrich, sondern durch Punkte voneinander getrennt.

Man findet gelegentlich die Aussage, dass für den ersten Namensteil der Assembly-Name verwendet würde. Das ist nicht korrekt! Dies ist lediglich die Voreinstellung des Default Namespaces, wenn man in Visual Studio .NET ein neues Projekt anlegt.

Bei den Verzeichnisnamen – nicht bei den Dateinamen – kommen weitere Veränderungen hinzu:

- Beginnt das Verzeichnis mit einer Ziffer, so wird ein Unterstrich vorangestellt.
- Sonderzeichen, zum Beispiel Klammern, nicht aber der Punkt, und Leerzeichen werden durch Unterstriche ersetzt.

Tabelle 1 listet einige illustrative Beispiele auf. Der Default Namespace ist hier *ResTest*. Die Dinge können offenkundig leicht verwirrend werden, sodass man die Dateinamen möglichst einfach halten sollte.

Bei einer Änderung des Default Namespaces in den Projekteinstellungen werden implizit die Namen aller Manifestressourcen geändert. Codestellen, die mit den Namen der Manifestressourcen arbeiten, werden aber nicht angepasst. Die Gefahr ist also groß, dass das Projekt nicht mehr fehlerfrei lauffähig ist. Diese Auswirkung ist nicht unbedingt offensichtlich, zudem wird sie in der Dokumentation nicht explizit erwähnt.

Zugriff mithilfe von Type

Die Abhängigkeit vom Default Namespace und die Konvention, nach der die Namen gebildet werden, können zu subtilen Fehlern und zu – aus Wartungsgesichtspunkten – problematischen Abhängigkeiten führen. Man kann sich das Leben hier etwas einfacher machen. *Assembly.GetManifestResourceStream()* übernimmt in einer zweiten Ausprägung zusätzlich ein Argument vom Typ *Type*. Diesem *Type* wird der Namespace entnommen und als erster Teil des Namens der Manifestressource verwendet. Die beiden Varianten in Listing 3 liefern ein identisches Ergebnis, vorausgesetzt, der aktuelle Typ liegt im Namespace *ResTest*.

Auch hier bieten einzelne Klassen vereinfachte Möglichkeiten an. *System.Drawing.Icon* etwa stellt einen Konstruktor mit

Listing 3

Öffnen einer Manifestressource über Type.

```
Type t= typeof(ResTest.Form1);
Assembly a= t.Assembly;

// vollständige Namensangabe
Stream strm1= a.GetManifestResourceStream
   ("ResTest.Res.logo.ico");

// Type steuert Namespace bei
Stream strm2= a.GetManifestResourceStream
   (t, "Res.logo.ico");
```

Argumenten der Typen *Type* und *String* an. Da der Aufruf zum eigentlichen Lesen nun nicht mehr über eine Assembly erfolgt, wird diese ebenfalls dem Typ entnommen, wie es Listing 4 demonstriert.

Das Verhalten im Fehlerfall hängt von der einzelnen Klasse ab. *Icon* löst beispielsweise eine *System.ArgumentException* aus, wenn die Manifestressource nicht gefunden wird.

Übrigens ist dies eine der Stellen, an denen in der .NET-Framework-Dokumentation der Begriff „Manifestressource“ nicht verwendet wird. Dort ist lediglich die Rede von „resource“, was eigentlich auf die im nächsten Abschnitt behandelte Art von Ressourcen hinweist.

Der Einsatz des *Type*-Argumentes erlaubt zwei Vorgehensweisen, die etwas günstiger sind, als den Namen immer vollständig anzugeben. Wenn eine Manifestressource einer Klasse unmittelbar zugeordnet ist, so kann man sie in das

Listing 4

Lesen eines Icons als Manifestressource (optimiert).

```
System.Drawing.Icon i= new Icon
   (typeof(ResTest.Form1), "Res.logo.ico");
```

gleiche Verzeichnis legen und muss als Namen der Manifestressource nur noch den Dateinamen angeben. Vorausgesetzt wird dabei, dass die Hierarchie der Verzeichnisse der Hierarchie der Namespaces entspricht.

Hat man hingegen getrennte Unterverzeichnisse für Ressourcen, so kann man in diese Unterverzeichnisse eine kleine Zugriffsklasse legen, deren einziger Zweck es ist, den Namespace beizusteuern. Listing 5 illustriert dieses Vorgehen.

Listing 5

Zugriffsklasse für Manifestressourcen.

```
namespace ResTest.Res
{
    public sealed class ResourceAccess
    {
    }
}

[...]

// Verwendung der Zugriffsklasse
Type t= typeof(ResTest.Res.ResourceAccess);
Stream strm= t.Assembly.GetManifest-
   ResourceStream(t, "TextFile1.txt");
```

Eine Änderung am Default Namespace – die hoffentlich mit der Anpassung der Namespaces der bereits existierenden Typen einhergeht – hat nun keine versteckte Auswirkung mehr auf den Code. Darüber hinaus hat diese Lösung aber noch weitere Vorteile: Sie arbeitet ohne weiteres Zutun Assembly-übergreifend. Der Name der Manifestressource hat sich ja bereits auf den Dateinamen reduziert, aber selbst diesen kann man – zum Preis des Pflegeaufwandes – in der Zugriffsklasse in Konstanten hinterlegen.

Ressourcen

Die beschriebenen Mechanismen für Manifestressourcen bieten eine gute technische Grundlage und erheblich mehr Flexibilität als in Prä-.NET-Zeiten. Sie sind jedoch auf einem vergleichsweise niedrigen Abstraktionsniveau angesiedelt.

Wie angekündigt gibt es eine zweite Form von Ressourcen. Diese laufen in der Dokumentation unter dem Begriff „Ressourcen“.

Allerdings wird dieser Begriff gelegentlich auch verwendet, wenn es um „Manifestressourcen“ geht.

Bei diesen Ressourcen handelt es sich vergleichbar einer Hashtable um Pakete von Schlüssel-Wert-Paaren, bei denen der Schlüssel ein eindeutiger String und der Wert ein beliebiges Objekt ist. Das gesamte Paket wird in der Regel in Form einer Datei mit der Endung *resources* als Manifestressource bereitgestellt. Üblicherweise werden diese Ressourcen in der Dokumentation im Zusammenhang mit dem Thema Globalisierung behandelt, da sie die Grundlage für die Lokalisierung von Windows Forms bilden. Aber auch ohne sich mit diesem Thema auseinander zu setzen, bieten Ressourcen in vielen Fällen deutliche Vorteile gegenüber einfachen Manifestressourcen:

- Sie sind deutlich effizienter bei kleinen Informationseinheiten, wenn man etwa Meldungs- oder Tooltip-Texte der besseren Wartbarkeit wegen auslagert, da sie nicht stream-basiert arbeiten.
- Sie sind im Gegensatz zu Manifest–ressourcen typisiert.

Die notwendigen Klassen, die mit diesen Ressourcen arbeiten können, liegen im Namespace *System.Resources*. Die wichtigste Klasse dort ist *ResourceManager*. Darüber hinaus gibt es weitere Klassen für das Lesen und Schreiben von Ressourcen.

Zu unterscheiden sind also:

- Eine Datei mit der Ergänzung *resources* landet als Manifestressource in der Assembly.
- Eine Ressource (ohne Zusatz) ist ein einzelner Wert in einer *resources*-Datei.

Die Klasse ResourceManager

System.Resources.ResourceManager ist die zentrale Klasse, wenn es darum geht, Ressourcen während der Laufzeit zu verwenden. Diese Klasse benötigt einen Verweis auf eine *resources*-Datei, die als Manifestressource abgelegt ist. Dabei ist lediglich zu beachten, dass die Erweiterung *resources* nicht Bestandteil des Namens ist, der dem Konstruktor als Argument übergeben wird. Das durchbricht zwar etwas die Konventionen, macht es aber zum Beispiel einfacher, einen Namen aus einem Datentyp abzuleiten, wie es beispielsweise in Listing 6 zu sehen ist.

Wenn der *ResourceManager* einmal angelegt ist benötigt man nur noch die Methode *GetObject()*, um eine Ressource zu liefern. Für Werte vom Typ *String* gibt es noch die typsichere Variante *GetString()*. Der Einsatz ist damit denkbar einfach, wie Listing 7 demonstriert.

Hat man eine *resources*-Datei als echte Datei, so kann man mithilfe der statischen Methode *ResourceManager.CreateFileBasedResourceManager()* einen *ResourceManager* anlegen. Das kann etwa relevant sein, wenn eine Anwendung die Ressourcen nachträglich zur Laufzeit manipulieren soll. Diese Möglichkeit wird weiter unten beschrieben.

Listing 6

Verwendung der Klasse ResourceManager.

```
ResourceManager rm= new ResourceManager
  ("ResTest.Res.Resource1",
   GetType().Assembly);
string s= rm.GetString("test");
Image i= (Image)rm.GetObject("bild");
```

Der Code in Listing 6 nutzt bereits einen wesentlichen Vorteil: die Typisierung. Das von der Methode *GetObject()* gelieferte Objekt ist bereits durch das .NET Framework erzeugt worden. Man muss sich keine Gedanken mehr darüber machen, ob die Informationen aus der Manifestressource korrekt interpretiert werden. Man kann – wie im Beispiel geschehen – mit Basistypen oder Interfaces arbeiten. Und last but not least: Sollte sich ein Fehler eingeschlichen haben und hinter *bild* verbirgt sich ein inkompatibles Objekt, etwa ein Dateiname statt des Bildes selbst, so wird eine Ausnahme zur Laufzeit, und zwar eine *System.InvalidCastException*, dieses Problem sehr schnell offensichtlich machen. Sollte die Manifestressource gar nicht erst gefunden werden, so wird eine *System.Resources.MissingManifestResourceException* ausgelöst.

Die Designer von Visual Studio .NET verwenden diese Ressourcen nach dem gleichen Strickmuster, wie es in Listing 7 zu sehen ist. Jedem Formular ist eine *resources*-Datei zugeordnet. In dieser finden sich die Informationen, die sich nur schlecht im Code abbilden lassen: beispielsweise das Image eines Buttons oder andere Daten, für es im Code kein Pendant gibt. Dazu zählen etwa solche Informationen, die nur für den Designer relevant sind, zum Beispiel, ob die Elemente gesperrt sind.

Wie bereits erwähnt, liegen Ressourcen in *resources*-Dateien in einem bi-

Listing 7

Code zum Auslesen eines Icons.

```
System.Resources.ResourceManager resources =
    new System.Resources.ResourceManager
    (typeof(Form1));
[...]
this.Icon =
((System.Drawing.Icon)(resources.GetObject
    ("$this.Icon")));
```

nären Format vor. Es gibt verschiedene Wege, solche Dateien zu erzeugen. Typischerweise wird eine *resx*-Datei als Quelle benutzt. Hierbei handelt es sich um eine äquivalente Darstellung im XML-Format, die mithilfe des *Resource File Generators* (*Resgen.exe*) in eine *resources*-Datei übersetzt werden kann.

In Visual Studio .NET wird eine *resx*-Datei über *File/New* und den Eintrag *Resource Template* angelegt. Verwendet man im Projekt *Add New Item*, so lautet der Eintrag *Assembly Resource File*. Als *Build Action* wird *Embedded resource* angegeben. Im Gegensatz zu anderen Dateitypen macht Visual Studio .NET bei *resx*-Dateien eine Sonderbehandlung und verwendet die Datei nicht direkt als Manifestressource. Sie wird vielmehr zunächst in eine *resources*-Datei übersetzt, und diese landet in der Assembly.

Es gibt zwei weitere Möglichkeiten, *resources*-Dateien anzulegen, die jedoch beide nicht unmittelbar von Visual Studio .NET unterstützt werden: Einfache benannte Zeichenketten, wie sie häufig benötigt werden, können in normalen Textdateien angegeben werden. Die einzelnen Zeilen haben dabei die Form *name =wert*. Weitere Infos dazu liefert [1]. Mithilfe von *resgen.exe* lässt sich eine solche Datei in eine *resx*- oder *resources*-Datei übersetzen.

Man kann Ressourcendateien in den angesprochenen und sogar in eigenen Formaten auf relativ einfache Art selbst lesen und schreiben. Dazu folgen weiter unten weitere Erläuterungen.

resx-Dateien sind XML-Dokumente. Sie enthalten zu Beginn ein XML Schema, gefolgt von einigen Meta-Informationen in den Elementen *resheader*, die nicht weiter interessant sind, und in *data*-Elementen die eigentlichen Einträge. Die *data*-Elemente enthalten in Attributen den Namen der Ressource, den Typ in voll qualifizierter Form, das heißt mit Angabe der Assembly – falls die Angabe fehlt, wird *System.String* als Typ unterstellt –, und im Element *value* den Inhalt des Objektes als String.

Zur Umwandlung der Objekte in *Strings* und umgekehrt werden die den Typen zugeordneten *TypeConverter* verwendet, die zum Beispiel auch im Fenster *Properties* in Visual Studio .NET Verwendung finden. Bei einzelnen Objekten, zum Beispiel Images, die in binärer Form abgelegt werden, wird über das Attribut *mimetype* auf ein Objekt zur Konvertierung verwiesen, zum Beispiel für *base64*. Die Werte sind in den durch Visual Studio .NET angelegten *resx*-Dateien dokumentiert. Listing 8 zeigt einige Beispiele. Das Bearbeiten dieser *resx*-Dateien ist in Visual Studio .NET derzeit nur in rudimentärer Form möglich. Sie werden wie alle anderen XML-Dateien als Text oder im *DataGrid* angeboten.

Listing 8

Ausschnitt aus einer resx-Datei.

```
<data name="listBox1.Location" type="System.Drawing.Point, System.Drawing,
Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a">
    <value>8, 48</value>
</data>
<data name="button1.FlatStyle" type="System.Windows.Forms.FlatStyle, System.Windows.Forms,
Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b77a5c561934e089">
    <value>Standard</value>
</data>
<data name="button1.Visible" type="System.Boolean, mscorlib, Version=1.0.5000.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089">
    <value>True</value>
</data>
<data name="label2.Image" type="System.Drawing.Bitmap, System.Drawing, Version=1.0.5000.0,
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" mimetype="application/x-
microsoft.net.object.bytearray.base64">
    <value>Qk32BAAAAAAAAH... ///////////w==</value>
</data>
```

Auf externe Dateien verweisen

Der Designer von Visual Studio .NET bettet Icons und andere Bilder direkt in die *resx*-Datei ein. Das birgt den Nachteil, dass nach einer Bearbeitung des Icons dieses explizit erneut zugewiesen werden muss. Es gibt aber auch eine Möglichkeit, aus der *resx*-Datei auf die Bitmap-Datei zu verweisen, ohne dass eine Kopie angelegt wird. Entsprechend schlägt sich eine Änderung nach einem Rebuild unmittelbar nieder. Das Mittel dazu ist die Klasse *ResXFileRef*. Diese wird statt des eigentlichen Objektes in der *resx*-Datei angegeben. Listing 9 zeigt ein Beispiel. Als Wert erhält sie den Pfad zur Datei sowie durch Semikolon abgetrennt den voll qualifizieren Typnamen. Der Pfad muss vollständig angegeben werden, da *Resgen.exe* offensichtlich sein eigenes Verzeichnis als Arbeitsverzeichnis verwendet.

Während der Übersetzung wird das Objekt vom Typ *ResXFileRef* gar nicht erst erzeugt. Vielmehr geht der zugeordnete Type Converter hin und erzeugt aus den Informationen das referenzierte Objekt, im Beispiel das Icon, das dann seinerseits in der *resources*-Datei landet. Wie Listing 10 demonstriert, lässt sich dieses Objekt ohne Unterschied zur anderen Vorgehensweise auslesen.

Wenn *resgen.exe* von der Kommandozeile aus verwendet werden soll, ist es notwendig, wie in Listing 9 dargestellt, die angegebenen Assemblies voll zu qualifizieren. Arbeitet man hingegen nur in Visual Studio .NET, so können die Angaben *Version*, *Culture* und *PublicKeyToken* entfallen, was die Sache einfacher macht.

Ressourcendateien lesen und schreiben

Die Klasse *ResourceManager* ist ideal für die häufigste Anforderung, eine Ressource gezielt zu lesen. Es gibt jedoch auch Fälle, in denen man die gesamte Ressourcendatei verarbeiten oder gar selbst Ressourcendateien schreiben will. An dieser Stelle kommen *ResourceReader* und *ResourceWriter* ins Spiel.

Das .NET Framework bietet jeweils einen Reader und einen Writer für *resx*-Dateien (*ResXResourceReader* und *ResXResourceWriter*) und *resources*-Dateien (*Re-*

Listing 9

Verweise auf externe Dateien/ Objekte in resx-Dateien.

```
<data name="externes_icon"
 type=" System.Resources.ResXFileRef,
 System.Windows.Forms, Version=1.0.5000.0,
 Culture=neutral,
 PublicKeyToken=b77a5c561934e089">
    <value>(kompletter pfad)\logo.ico;Sys-
tem.Drawing.Icon, System.Drawing</value>
</data>
```

sourceReader und *ResourceWriter*). Die Gemeinsamkeiten sind über Interfaces gekapselt (*IResourceReader* und *IResourceWriter*), sodass man theoretisch auch eigene Formate – zum Beispiel in einer Datenbank abgelegte Ressourcen – unterstützen könnte.

Reader und *Writer* sind einfach aufgebaut: Ein *ResourceReader* liefert über die Methode *GetEnumerator()* einen *IDictionaryEnumerator*, über den sich die Schlüssel-Wert-Paare abarbeiten lassen. Die Schlüssel sind hierbei vom Typ *String*, die Werte sind die korrekt erzeugten Objekte, also ein *System.String* für eine Zeichenkette oder ein *System.Drawing.Bitmap* für ein Bild.

Ein *ResourceWriter* ist das direkte Gegenstück. Er bietet mit der Methode *AddResource()* die Möglichkeit, ein Schlüssel-Wert-Paar hinzuzufügen. Abschließend kann die Ressource dann mit der Methode *Generate()* in einem Rutsch geschrieben werden.

Bei den existierenden *Reader*- und *Writer*-Klassen hat man je zwei Konstruktoren. Einer übernimmt einen Stream, der andere einen Dateinamen. Listing 11 zeigt, wie man eine als Manifestressource in der Assembly enthaltene *resources*-Datei liest. Mit *Assembly.GetManifestResourceStream()* und einem *ResourceReader* für diesen *Stream* ist das problemlos machbar.

Listing 10

Das extern referenzierte Icon einlesen.

```
ResourceManager rm= new ResourceManager
  ("ResTest.Res.Resource1",
   GetType().Assembly);
System.Drawing.Icon i= (Icon)rm.GetOb-
ject("externes_icon");
```

Listing 11

Ressourcen aus einer Manifestressource auslesen.

```
Assembly a= GetType().Assembly;
Stream strm= a.GetManifestResourceStream
  ("ResTest.Res.Resource1.resources");
IResourceReader rr= new ResourceReader
  (strm);
IDictionaryEnumerator en =
  rr.GetEnumerator();
while (en.MoveNext())
{
    [...]
}
```

Das Anlegen einer Ressourcendatei ist ebenso einfach und wird in Listing 12 gezeigt. Natürlich kann man Manifestressourcen nicht nachträglich manipulieren.

Man könnte dies zum Beispiel verwenden, um Programmeinstellungen generisch zu sichern.

ASP.NET

Zum Schluss noch zwei Anmerkungen zu ASP.NET-Anwendungen: Da der Name der Assembly aufgrund der Arbeitsweise von ASP.NET nicht im Voraus bekannt ist, kann nicht über den Typ und seine Metainformationen *(typeof, GetType())* gearbeitet werden. Stattdessen muss die Assembly gezielt geladen werden.

Listing 13 setzt die Arbeitsweise von Listing 1 für eine ASP.NET-Anwendung um.

Der zweite Hinweis ist eine Warnung: Die beschriebene Möglichkeit, eine *resources*-Datei direkt über *ResourceManager.CreateFileBasedResourceManager()* zu lesen, hält die Datei gesperrt. Man kann zwar *ResourceManager.ReleaseAllResources()* aufrufen, um sie freizugeben. Das bewirkt jedoch auch, dass mit der nächsten Anforderung die Ressourcen erneut geladen werden müssen.

Listing 12

Eine Ressourcendatei erzeugen und schreiben.

```
IResourceWriter rw= new
  ResXResourceWriter("test.resx");
rw.AddResource("bool", true);
rw.AddResource("string", "text");
rw.AddResource("string[]",
  new string[]{"a", "b", "c"});
rw.Generate();
rw.Close();
```

Listing 13

ResourceManager in ASP.NET verwenden.

```
Assembly a= Assembly.Load("ResASP");
Stream strm= a.GetManifestResourceStream
  ("ResASP.Res.TextFile1.txt");
StreamReader sr= new StreamReader(strm);
string s= sr.ReadToEnd();
sr.Close();
```

In einer ASP.NET-Anwendung durchbricht das Sperren einer Datei die XCOPY-Deployment-Fähigkeit und damit die Möglichkeit, im laufenden Betrieb Assemblies und andere Dateien zu löschen beziehungsweise durch neuere Versionen zu überschreiben. Daher rät die Dokumentation vom Einsatz explizit ab. Da man in einer Webanwendung aber kaum *resources*-Dateien manipulieren wird, kann man diese eigentlich immer als Manifestressource – nötigenfalls in einer eigens erstellten Assembly – bereitstellen.

Beispielprogramm

Das Beispielprogramm auf der Heft-CD setzt alle hier nur als Fragmente gezeigten Codestücke in lauffähiger Form um.

Im SDK findet sich in *C:\Programme\Microsoft Visual Studio .NET 2003\SDK\v1.1\Samples\Tutorials\resourcesandlocalization\resxgen* ein Beispielprogramm, mit dem Bilder in *resx*-Dateien konvertiert werden. Zwar wird dieses dann von anderen Beispielen verwendet, um das eigene Programm zu übersetzen, aber mit dem Einsatz von *ResXFileRef* wäre dieses Ziel einfacher erreichbar.

Weitere einfache Beispiele finden sich unter C:\Programme\Microsoft Visual Studio .NET 2003\SDK\v1.1\ QuickStart\howto\samples\resources. ||||||

[1] Resources in Text File Format. .NET Framework Developer's Guide: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconresourcesintextfileformat.asp

Showcase & Services

dotnetpro präsentiert

Showcase & Services

Ihre neue Anzeigenrubrik für Produkt- und Serviceangebote

Ihr Ansprechpartner hierfür ist:

Frank Rimkus

Tel.: 0431/329 26 93, Fax: 040/360 335 07 22

e-Mail: Frank.Rimkus@dotnetpro.de

dotnetpro

Anzeigenpreise der Rubrik Showcase & Services

1/8 Seite	350,00 €
1/4 Seite	720,00 €
1/3 Seite	950,00 €
1/2 Seite	1.500,00 €
2/3 Seite	1.900,00 €
3/4 Seite	2.200,00 €
1/1 Seite	2.900,00 €

Alle Preise zzgl. gesetzl. MwSt.

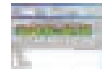
Bitte beachten Sie den Anzeigenschluß der DotNetPro 07-08/04: Mittwoch, 24. Mai 2004

Erscheinungstermin: 16. Juni 2004

Wir realisieren zusammen mit IHNEN Leistungsstarke .NET Softwarelösungen

Individualsoftwareentwicklung
Anbindungen an SAP
Migrationen nach .NET
Smart Devices
Coaching
und vieles mehr.

Softwaretools:
QualityAnalyzer für VB5/6 **249,00 €**



Sourcecodeanalyse, Qualitätsverbesserung, Performancesteigerung,
Erleichtert den Umstieg nach VB.NET
Demoversion unter www.conprise.de

.NET-Webspace:
Einführungspreis, monatlich nur **9,95 €**
100 MB für Ihre ASP.NET-Seiten, MS-SQL-Server Datenbank, ODBC-Unterstützung, eigene Domän, Windows 2003 Server mit .NET Framework 1.1.

Conprise
Consulting & Softwaretools

Conprise IT-Consulting GmbH
Bornimer Str. 4a
D-14612 Falkensee

Fon: 03322 / 234 99 - 0
Fax: 03322 / 234 99 - 4

eMail: info@conprise.de
Web: www.conprise.de

Microsoft CERTIFIED Partner

Vorschau

Die Ausgabe 7/8 • 2004 erhalten Sie ab 16. Juni

THEMEN IM NÄCHSTEN HEFT

.NET kann’s

Es mag ja immer noch Leute geben, die glauben, dass .NET eingeschränkt ist und verschiedene Dinge nicht kann. Wir zeigen Ihnen, dass dem nicht so ist. Ob grafische Effekte, Fuzzy Logic oder Office steuern: Alles geht – NET kann’s eben doch.

Geschwindigkeit in verteilten Systemen

Betriebsame Geschäftigkeit auf der einen Seite, lähmende Langeweile auf der anderen und dazwischen das Nadelöhr, das an dieser ungleichen Auslastung schuld ist. Wie Sie solche Szenarien vermeiden können, erfahren Sie in der nächsten dotnetpro.

Komponenten-Designer im Eigenbau

Die Komponente wäre der Renner geworden, hätte der Programmierer doch nur einen Designer mit eingebaut. Damit Ihnen der Erfolg Ihrer Komponenten nicht versagt bleibt, zeigt dotnetpro, wie Sie ihnen einen Designer spendieren.

Ode an IDisposable

„Jetzt nicht“, murmelt der Garbage Collector und lässt das Objekt am Leben. Dass dadurch Ihr Programm aus dem Tritt kommt, interessiert ihn herzlich wenig. Damit das nicht passiert, stellt dotnetpro das Interface IDisposable vor und zeigt, wie Sie es geschickt einsetzen.

Privat ist nicht

Private Methoden sind vor den Zugriffen Dritter sicher? Glauben Sie das ja nicht. dotnetpro zeigt, wie Sie auch private Methoden aufrufen können.

www.dotnetpro.de

LESEN SIE IN WEITEREN REDTEC-PUBLIKATIONEN

Creative Live, Ausgabe 2/2004
Vom Fax zur Animation: Wie ein Designer ein Haus in vier Tagen errichtet. Worauf müssen Leveldesigner beim Entwickeln von Spielen achten? Anschauliche Animationsprojekte: über das Legoland und einen Werbefilm einer Studi-Kneipe. Die Heft-DVD enthält unter anderem eine Demo von 3ds max 6. **www.creative-live.com**