Typsicherheit bei Lokalisierung

Ein großer Nachteil bei der standardisierten Vorgehensweise .NET Projekte zu lokalisieren, ist die fehlende Typsicherheit. Noch dazu ist immer einhergehend ein zwingender Wechsel der CultureInformation notwendig. Es gibt aber Szenarien, bei dem der Wechsel nicht von Vorteil ist.

Das Projekt LocalizeMe wurde genau aus diesen Gründen ins Leben gerufen. Es wird mit der sehr freizügigen Lizenz MIT zur Verfügung gestellt. Basierend auf .NET Standard 2.0 ist es möglich im klassischen .NET Framework zu bleiben, oder mit .NET 5+ zu arbeiten.

Nach der Erstellung einer neuen Konsolen App, wird das Package installiert:

```
PM> Install-Package LocalizeMe -Version 0.8.5
```

Das Paket besteht aus 2 Assemblys

- LocalizeMe.Standard
- LocalizeMe.Shared

Die Assembly „Shared“ enthält dabei die Informationen der Lokalisierung. Die Assembly „Standard“ enthält die Logik, die am Ende eine Klassenbibliothek erstellt, die alle relevanten Daten zur Verfügung stellt.

Es gibt 4 Typen der Lokalisierung, die in sogenannten Ressourcen unterteilt werden

1) Messages – Alle Textrelevanten Übersetzungen
2) Images – Alle Bilddateien, die lokalisiert werden sollen
3) Files – Alle anderen Dateien, die lokalisiert werden sollen
4) Enumerations – Alle im Projekt vorhandenen Enums können lokalisiert werden

Ein Projekt kann in mehrere Ressourcen aufgeteilt werden. Als Beispiel kann man eine „Message“ Ressource für die Oberfläche nutzen und eine zweite „Message“ Ressource als lokalisierte Fehlerbeschreibung.

Jede Ressource wird einmalig erstellt und ein Namespace wird der Ressource zugewiesen.

```
var locMessages = new LocalizeMeShared.Data.Base()
{
    NameSpace = "UILocalization",
    TypeOfResource = LocalizeMeShared.Enums.TypesOfResource.Messages
};
```

Nun können beliebig viele sogenannte RessourceData hinzugefügt werden.
Dabei gilt zu beachten, dass die Eigenschaft des Namens relevant ist für die einzelnen Lokalisierungen und mögliche Platzhalter hier eingefügt werden müssen. Deshalb ist eine Konstante zu empfehlen.

```
const string FullName = "Full name";
locMessages.ResourceData.Add(new LocalizeMeShared.Data.ResourceData()
{
    Name = FullName,
    Comment = "Der Name des Kunden",
    CultureCodeName = LocalizeMeShared.Enums.TypesOfCulture.de_AT,
    IsInvariant = true,
```

```
    Value = "Kundenname"
});

locMessages.ResourceData.Add(new LocalizeMeShared.Data.ResourceData()
{
    Name = FullName,
    CultureCodeName = LocalizeMeShared.Enums.TypesOfCulture.en_GB,
    Value = "Full name"
});
```

Die „ResourceData“ besteht aus mehreren Eigenschaften.
Der Name ist pro Namespace einzigartig (Name = FullName)
Eine Sprache muss gewählt werden (CultureCodeName)
Es muss eine Standardsprache definiert werden (IsInvariant = true)
Es gibt auch die Möglichkeit der expliziten Angabe von IsRightToLeft. Wenn diese nicht gesetzt wird, entscheidet das Programm beim Erstellen, ob die angegebene Sprache die entsprechende Ausrichtung hat.
Wenn man einen Kommentar bei der Standardsprache vergibt, wird dieser dann in der IDE angezeigt. Deshalb muss bei den Übersetzungen nur mehr die relevanten Werte gesetzt werden.
Es gibt noch optional den Wert Tag, oder für die Dateien (Bilder oder Daten)

- FileContent
- FileName
- TypeOfFile

Eine Besonderheit ist das Setzen von Platzhaltern.

```
const string Age = "Customers age {0}";
locMessages.ResourceData.Add(new LocalizeMeShared.Data.ResourceData()
{
    Name = Age,
    Comment = "Das Alter des Kunden",
    CultureCodeName = LocalizeMeShared.Enums.TypesOfCulture.de_AT,
    IsInvariant = true,
    Value = "Der Kunde ist {0} Jahre alt"
});

locMessages.ResourceData.Add(new LocalizeMeShared.Data.ResourceData()
{
    Name = Age,
    CultureCodeName = LocalizeMeShared.Enums.TypesOfCulture.en_GB,
    Value = "The customer is {0} years old"
});
```

Mit Enumerationen und Dateien funktioniert es genau gleich, wobei den Enums noch eine Besonderheit zukommt. Der Name muss dabei explizit derselbe sein, wie der Name der Enum, die man übersetzten möchte.

```
const string EnumRedColor = "RedColor";

var locEnums = new LocalizeMeShared.Data.Base()
{
    NameSpace = "Enumerations",
    TypeOfResource = LocalizeMeShared.Enums.TypesOfResource.Enumerations
};
locEnums.ResourceData
    .Add(
```

```
        new LocalizeMeShared.Data.ResourceData()
        {
            Name = EnumRedColor,
            Comment = "Red Color",
            CultureCodeName = LocalizeMeShared.Enums.TypesOfCulture.de_AT,
            IsInvariant = true,
            Value = "Rot"
        });
locEnums.ResourceData
    .Add(
        new LocalizeMeShared.Data.ResourceData()
        {
            Name = EnumRedColor,
            CultureCodeName = LocalizeMeShared.Enums.TypesOfCulture.en_GB,
            Value = "Red"
        });
```

Mit diesem kleinen Beispiel können wir nun sehr einfach die Assembly erstellen, die zur Lokalisierung erwünscht ist.

Dazu müssen die Basis Daten an die Factory übergeben, ein Ziel-Framework ausgewählt und ein Export Pfad angegeben werden.

```
var locListOfNamespaces = new List<LocalizeMeShared.Data.Base>();
locListOfNamespaces.Add(locMessages);
locListOfNamespaces.Add(locEnums);

var locFactory = new LocalizeMe.Standard.Factory();
locFactory.CleanFolder(Path.Combine(Environment.GetFolderPath(Environment.Special
Folder.Desktop), @"LocalizedProjekt")) //Vorbereitung des Pfades, wo das Projekt
erstellt wird
        .CreateProject(LocalizeMeShared.Enums.TypesOfProject.Net6) //Das
Zielframework bestimmen
        .CreateResourceFiles(locListOfNamespaces) //Die Daten hinzufügen
        .Compile(); //Erstellen des Projekts
```

Nach dem die Konsolen App nun ausgeführt wurde, erscheint auf dem Desktop ein Ordner mit dem Namen „LocalizedProjekt“ darin enthalten ist eine Assembly, die den Code für eine typsichere Lokalisierung enthält. Diese kann nun in ein neues Projekt eingefügt werden.

Der Code für den Namespace „UILocalization“ sieht nun wie folgt aus:

```
var locLocalizedProjekt = new LocalizedProjekt.LocalizeMe();

Console.WriteLine(locLocalizedProjekt.GetLocalizedUILocalization.FullName);
Console.WriteLine(locLocalizedProjekt.GetLocalizedUILocalization.SetCustomersAge(
"4"));

locLocalizedProjekt.SetAllToCulture(IS_Logic.Cultures.United_Kingdom);

Console.WriteLine(locLocalizedProjekt.GetLocalizedUILocalization.FullName);
Console.WriteLine(locLocalizedProjekt.GetLocalizedUILocalization.GetCustomersAge)
;
```

Dabei ist zu beachten: Der Namespace“UILocalization“ wurde zu „GetLocalizedUILocalization“. Die Konstante „Fullname“ wurde auf Grund des Werts geändert auf „FullName“. Da die Sparche „de_AT“ als Standarsprache gesetzt wurde, muss nach der Erstellung des Objekts keine explizite Sprache angegeben werden. Es wird automatisch die Standardsprache genommen. Bei den Platzhaltern wird keine Methode erstellt, sondern es werden eine WriteOnly Eigenschaft und eine ReadOnly Eigenschaft erstellt. Man muss entsprechend den Wert einmal setzen, um ihn dann jederzeit per „Get“ wieder lesen zu können. Das Resultat erscheint wie folgt:

Bei den Enumerationen kann nun folgender Code hinzugefügt werden

```
namespace dotnetProTest
{
    internal enum Colors
    {
        RedColor
    }
}
```

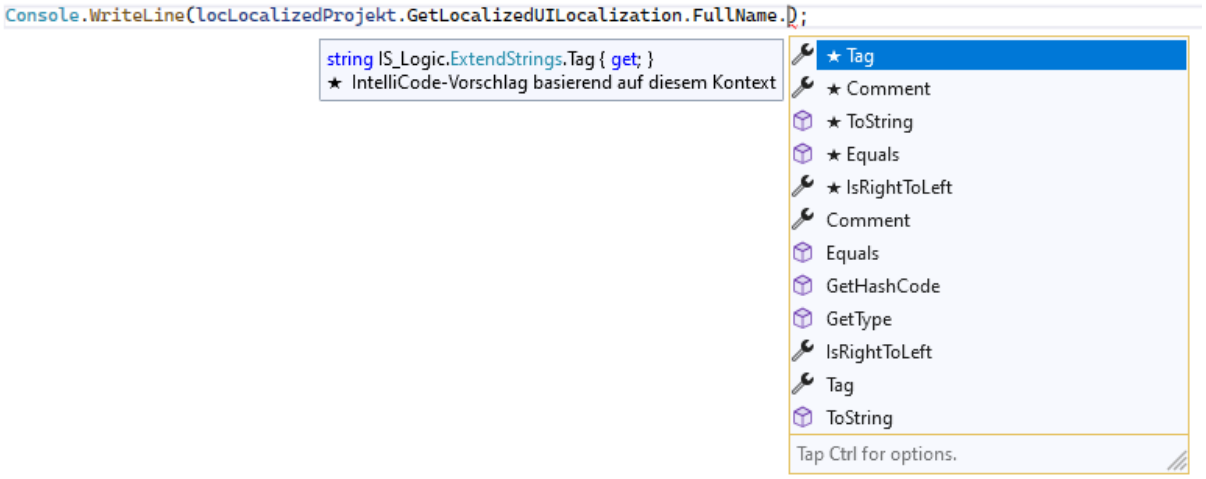
```
using Extensions;

var locLocalizedProjekt = new LocalizedProjekt.LocalizeMe();

Console.WriteLine(dotnetProTest.Colors.RedColor.GetLocalizedText(locLocalizedProjekt.GetLocalizedEnumerations.GetActualLocalization));

locLocalizedProjekt.SetAllToCulture(IS_Logic.Cultures.United_Kingdom);

Console.WriteLine(dotnetProTest.Colors.RedColor.GetLocalizedText(locLocalizedProjekt.GetLocalizedEnumerations.GetActualLocalization));
```

Das Ergebnis:

Zusätzlich wird auch ein Interface erstellt, mit dem Namen ILocalizeMe. Dieses Interface kann von DI Komponenten genutzt werden, um eine zentrale Stelle für die Lokalisierung zu bekommen.

Es können die Sprachen auch pro Namespace geändert werden und somit auch unterschiedlich genutzt werden.

Jede Methode/Eigenschaft, die generiert wird, erhält auch entsprechende Erweiterungen



Fazit:

Lokalisierung ist ein komplexes Thema. Mit diesem kleinen Tool ist man typsicher und schnell unterwegs. Ein explizites Beispiel, auch mit Bildern gibt es auf der GitHub Seite: [Intelli-Soft/TypeSafeLocalization: Test Project for NuGetPackage IntelliSoft.LocalizeMe (github.com)]

Zur Zeit gibt es eine Einschränkung: Die Code Erstellung funktioniert nur auf Windows Maschinen (auf anderen Plattformen wurde es nicht getestet)