Eastern Academic Journal

ISSN: 2367–738
Issue 3, pp. 99-110, September, 2018

РАЗРАБОТВАНЕ НА МНОГОЕЗИЧНИ .NET ПРИЛОЖЕНИЯ

Мария Г. Георгиева

РЕЗЮМЕ — *Разгледани са основните моменти при разработването на съвременни .NET приложения с многоезичен потребителски интерфейс. На базата на примерно приложение, създадено на C# (v.7.1) в среда за разработка MS Visual Studio 2017, последователно са коментирани и онагледени стъпките при планирането и програмирането на приложение, позволяващо по време на изпълнение (run-time) да се променя езиковата култура. Особено внимание е отделено на възникването на възможни форматни проблеми и начините за решаването им.*

Ключови думи: .NET платформа, глобализация, езикова култура, локализация, многоезичен, сериализация

DEVELOPING A MULTILINGUAL .NET APPLICATIONS

Maria G. Georgieva

ABSTRACT— *The main points of the development of modern .NET applications with multilingual user interface are explored. Based on an exemplary application created on C # (v.7.1) in MS Visual Studio 2017 development environment, the steps in the planning and programming of an application that allows run-time to change the language culture. Particular attention is paid to the emergence of possible format problems and the ways to solve them*

Keywords: .NET platform, culture, globalization, localization, multilingual, serialization

ВЪВЕДЕНИЕ

Важен момент в програмирането на съвременни приложения (Desktop или Web базирани) е разработването на многоезичен потребителски интерфейс, позволяващ на потребителя да работи на разбираем за него език. Самият процес на интернационализация на приложенията се състои от два момента (Brunetti, Boncinelli, 2013):

- Глобализация – адаптация на приложението за работа с различни езици и регионални настройки

Corresponding author at: New Bulgarian University, dep. of Informatics, 1618 Sofia, Bulgaria 21 Montevideo Str., e-mail: **m.georgieva@nbu.bg**

ISSN: 2367–738
Issue 3, pp. 99-110, September, 2018

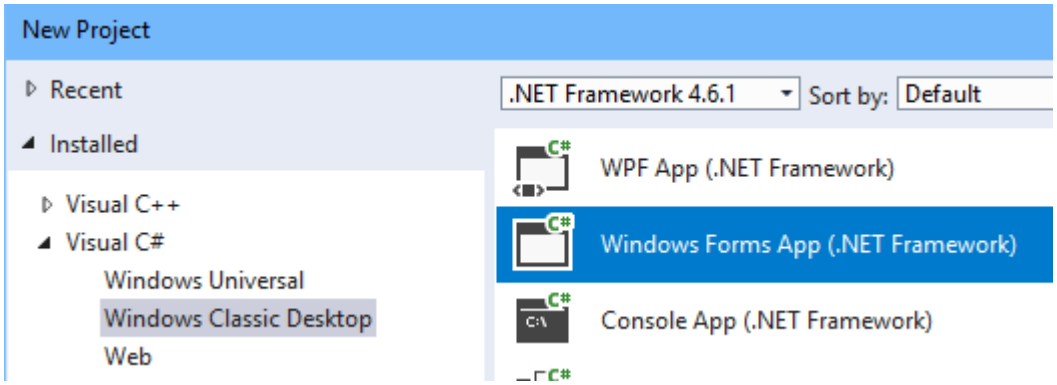
- Локализация – адаптация на приложението към националните особености на дадена страна, т.е. конфигуриране на приложението за дадена езикова култура – комбинация от езика и мястото, където се говори. Например:
  en-US – английски език; страна – USA
  en-GB – английски език; страна – Great Britain
  bg-BG – български език; страна – България

Глобализираното приложение поддържа локализиран потребителски интерфейс и е езиково и културно неутрално. То:

- коректно разпознава, обработва и визуализира характерните за дадена езикова култура формати – числа, дати, разделители, символи за валута и т.н. ;
- правилно и без проблеми сортира символни низове;
- съхранява и повторно възпроизвежда данни в необходимия формат

Създаването на многоезични приложения, поддържащи европейски регионални (и особено източно-европейски) настройки е съпроводено с решаването на определени форматни проблеми. Често, даже за един и същи език, регионалните настройки могат да се различават – например при американския(US) и английския (UK) английски. Не отчитането на характерните особености може да доведе до двусмисленост и даже до грешки и преустановяване изпълнението на приложението.

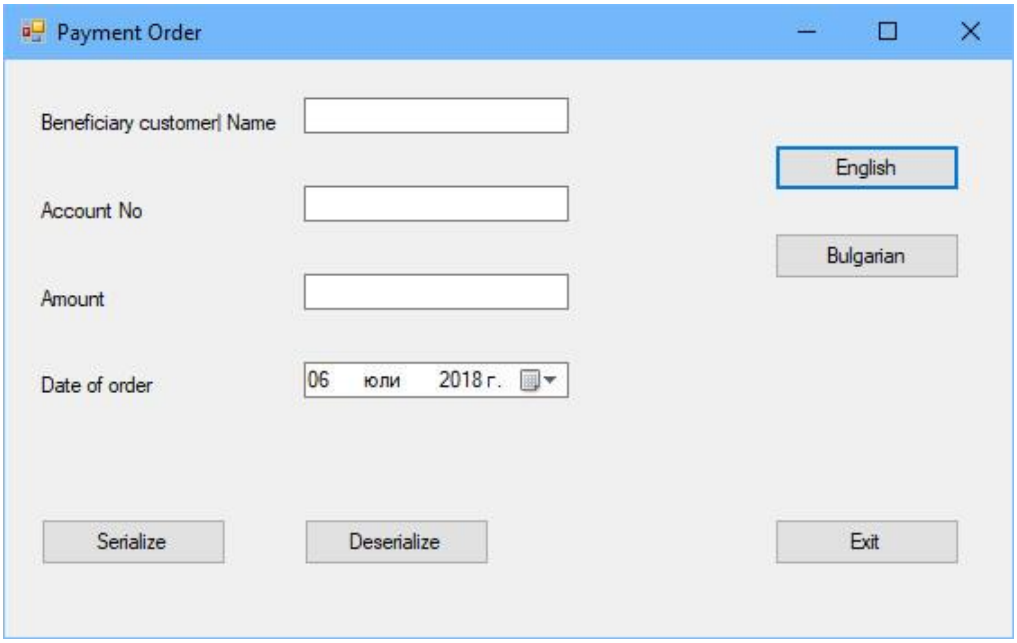
За да се избегнат тези проблеми последователно ще бъдат изпълнени отделните моменти от създаването на едно примерно приложение тип *Windows Classic Desktop/Windows Forms App* във *Visual Studio 2017*



Фигура 1 Създаване на приложението *MultiLingualApp*

Самата форма ще има първоначално няколко контроли – етикети, текстови полета, контрола за дата и час (DateTimePicker) и бутони, а впоследствие ще бъдат добавени панели и контроли за работа с файлове (OpenFileDialog и SaveFileDialog).

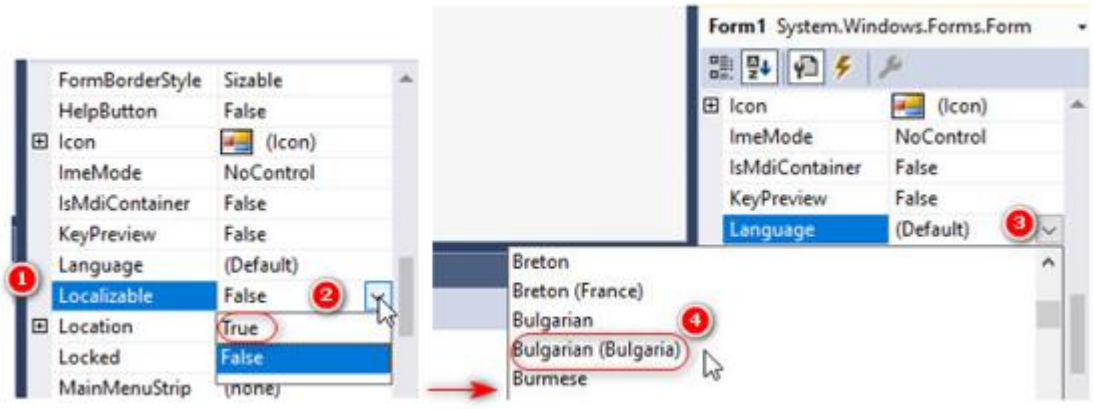
ISSN: 2367–738
Issue 3, pp. 99-110, September, 2018



Фигура 2 Общ вид на приложението

ГЛОБАЛИЗАЦИЯ И ЛОКАЛИЗАЦИЯ

В .NET самият процес на разработването на многоезични приложения е доста улеснен с помощта на пространството на имена *System.Globalization* и принадлежащите към него 20-на класа. За реализация е достатъчно само да се укаже за дадена форма, че свойството *Localizable* е вярно, след което може да се промени езика по подразбиране на формата (например *en-US* , да се смени с *bg- BG, fr-FR* или някакъв друг*)* .

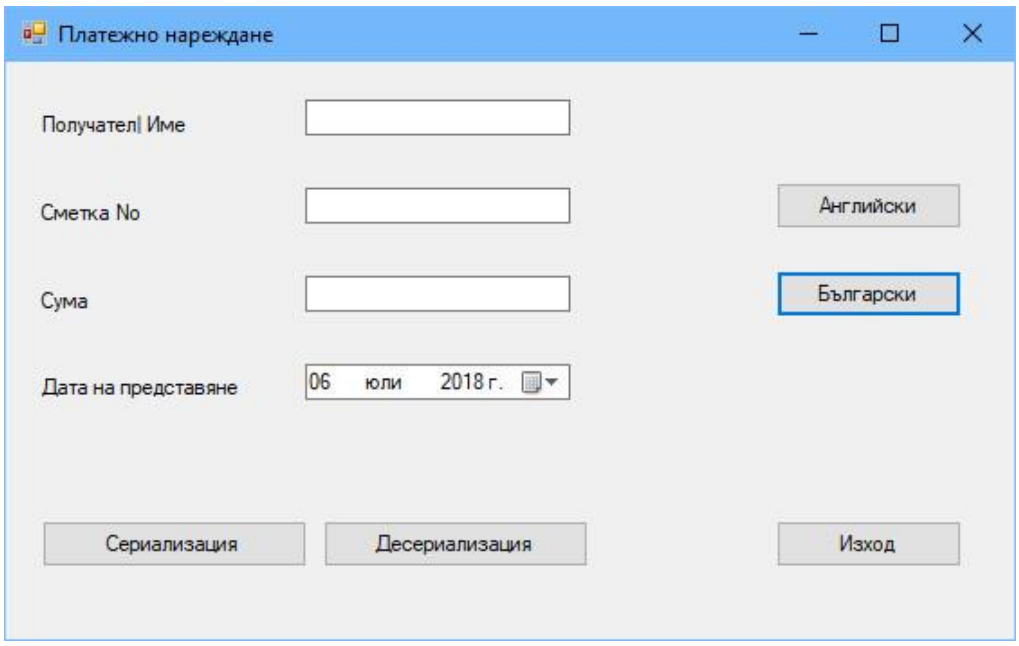


Фигура 3 Подготовка на формата за многоезичен интерфейс

Това води до създаване на ресурси за добавения език, които са разположени в поддиректории на директорията на самото приложение. За сменения език последователно се променят надписите на съответните контроли. В едно приложение могат да се използват множество езикови култури, като за всяка една от тях се генерират съответните поддиректории и ресурси.

Налице са няколко характерни особености за този процес:

- Контролите могат да са разположени на различни места в различните езикови варианти на формата
- Техните размери могат да са различни (често надписите на бутоните за различните езици имат различна дължина)
- Нови контроли могат да се добавят само във варианта за езика по подразбиране, след което се превеждат наименованията им и в останалите езикови форми на интерфейса (по принцип формата е една, но към нея има различни ресурсни файлове)



Фигура 4 Вид на формата за български език bg-BG

За да могат езиковите култури да се сменят успешно по време на работата на самото приложение (run-time) трябва да се включи и пространството на имена *System.Threading* . Самата промяна може да се инициира с помощта на бутони, радио-бутони, опция от меню и т.н.

СМЯНА НА ЕЗИКОВАТА КУЛТУРА

С помощта на бутоните *"English"* и *"Bulgarian"* може да се инициира промяна на езика на интерфейса, а методът *ChaneLanguage(string lng)*, с който се осъществява това, е показан на фигура 5.

Методът позволява да се променят текстовете на контролите във формата. Основен момент в него е използването на класа *CultureInfo*, съдържащ информация за езика, формата на датите (къс, дълъг формат, разделител, подредба), формата на числата, валутата и т.н.

Двата оператора:

```
Thread.CurrentThread.CurrentCulture = culture;
Thread.CurrentThread.CurrentUICulture = culture;
```

осъществяват промяна съответно на текущата езикова култура и на езиковата култура за графичния потребителски интерфейс.

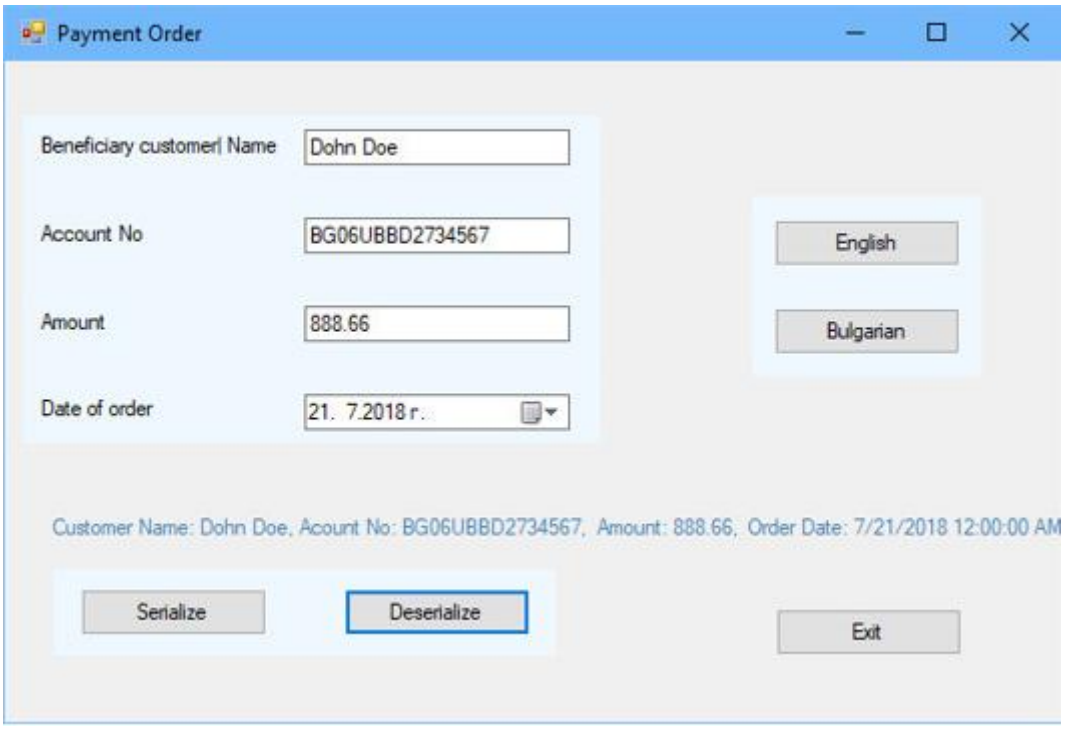
```
public void ChangeLanguage(string lng)
{
    culture = new CultureInfo(lng);
    Thread.CurrentThread.CurrentCulture = culture;
    Thread.CurrentThread.CurrentUICulture = culture;

    ComponentResourceManager resources = new ComponentResourceManager(this.GetType());
    this.Text = resources.GetString("$this.Text", culture);
    foreach (Control c in this.Controls)
    {

        resources.ApplyResources(c, c.Name, culture);
    }
}
```

Фигура 5 Метод за смяна на езиковата култура

В показания метод променливата lng **може** да има стойност ***bg-BG***, например.

Важно е да се отбележи, че промяната не се отнася до езиковата култура и регионалните настройки на операционната система, а е валидна само в рамките на изпълняваното приложение.

За удобство на потребителя много често във формите се използват и т.н. "контроли-контейнери" – *Panel, TabControl, SplitContainer, GroupBox* и т.н.



Фигура 6 Вид на формата с добавени панели

При наличие на контроли - контейнери методът от фиг.6 ще се окаже неефективен и смяна на текста на контролите, включени в отделните контейнери, няма да се осъществи. За да е успешна промяната, методът трябва да се преработи и да се използва рекурсия за обхождане на всички контроли. Новият метод ще изглежда така:

ISSN: 2367–738
Issue 3, pp. 99-110, September, 2018

```
public void ChangeLanguage(Control obj, string in_lang)
{
    culture = new CultureInfo(in_lang);
    Thread.CurrentThread.CurrentCulture = culture;
    Thread.CurrentThread.CurrentUICulture = culture;
    this.Text = resourse.GetString("$this.Text", culture);
    foreach (Control c in obj.Controls)
    {
        if (c.HasChildren)
            ChangeLanguage(c, in_lang);
        resourse.ApplyResources(c, c.Name, new CultureInfo(in_lang));
    }
}
```

Фигура 7 Променен метод за обхождане и смяна с използване на рекурсия

Параметрите при първоначалното извикване на метода и влизане в рекурсията ще бъдат съответно *ChangeLanguage(this,lang);*

СЪЗДАВАНЕ И РАБОТА С РЕСУРСНИ ФАЙЛОВЕ

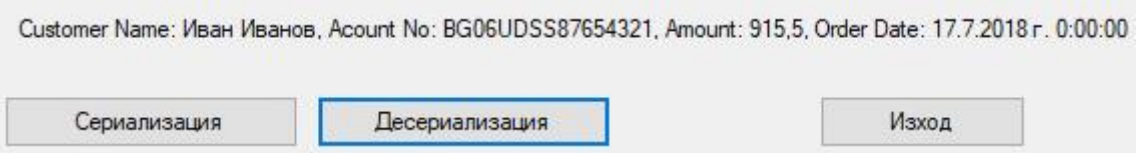
В голямата част от приложенията в хода на работата им се налага визуализация на информация за потребителя – чрез контроли *Label*, съобщения от тип *MessageBox* и т.н. За коментирания пример може да се създаде клас *PaymentOrder*, да се пренапише метода *ToString()* на класа и да се визуализира с помощта на контрола Label състоянието на създадена инстанция на класа.

```
[Serializable]
6 references
public class PaymentOrder
{
    private string mDate;
    private double amount;
    private string name;
    private string acountIBAN;
    private string lang;
    2 references
    public string MyDate
    {
        get { return mDate; }
        set { mDate = value; }
    }
    2 references
    public double Amount
    {
        get { return amount; }
        set { amount = value; }
    }
    2 references
    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    2 references
    public string AcountIBAN
    {
        get { return acountIBAN; }
        set { acountIBAN = value; }
    }
    1 reference
    public override string ToString()
    {
        return "Customer Name: "+name+", Acount No: "+acountIBAN+", Amount: "+amount.ToString()+", Order Date: "+ mDate;

    }
    2 references
    public string Lang
    {
        get { return lang; }
        set { lang = value; }
    }
}
```

Фигура 8 Клас PaymentOrder

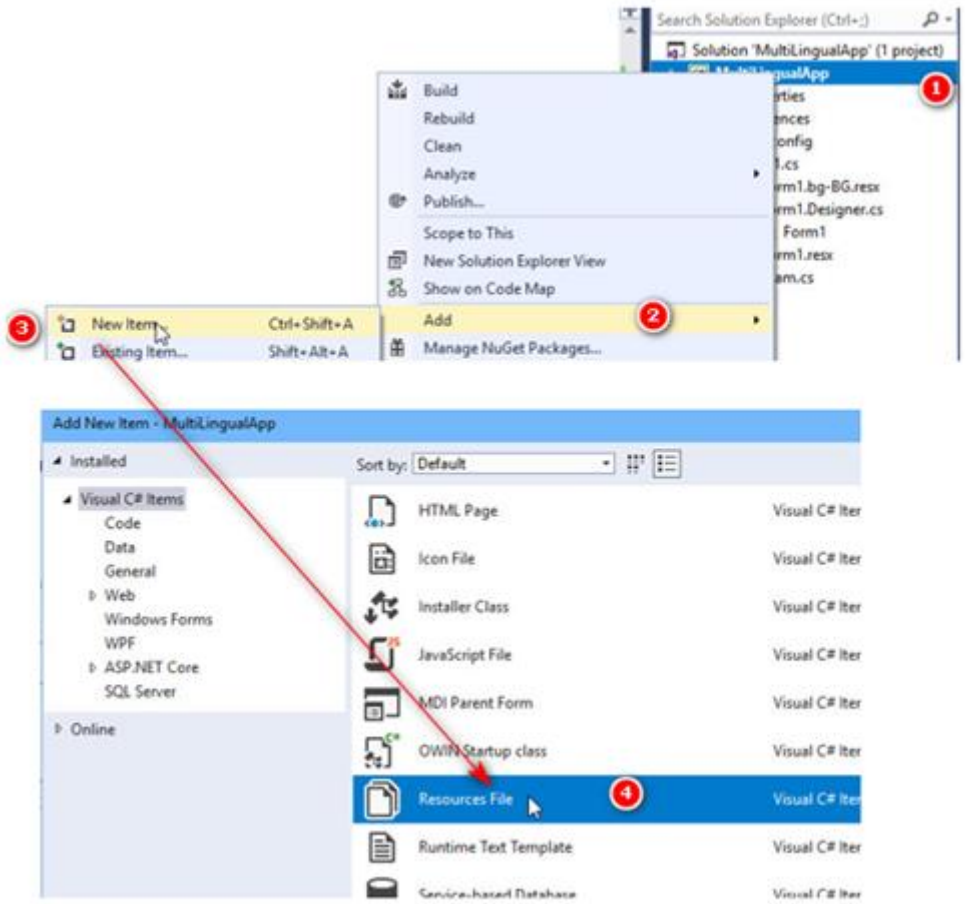
ISSN: 2367–738
Issue 3, pp. 99-110, September, 2018

При смяна на езиковата култура обаче, понякога показваната информация ще изглежда неподходящо.



Фигура 9 Смесване на две езикови култури

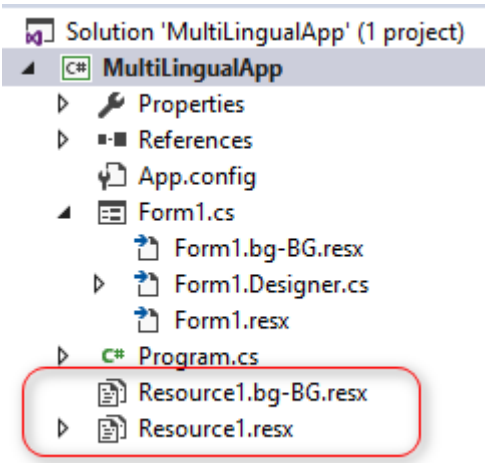
В такива случаи се генерира допълнителен ресурсен файл (Albahari, Albahari, 2018). Един от възможните начини е с помощта на *MS Visual Studio*, избирайки от контекстното меню в *Solution Explorer* добавяне на нов ресурсен файл.



Фигура 10 Създаване на ресурсен файл

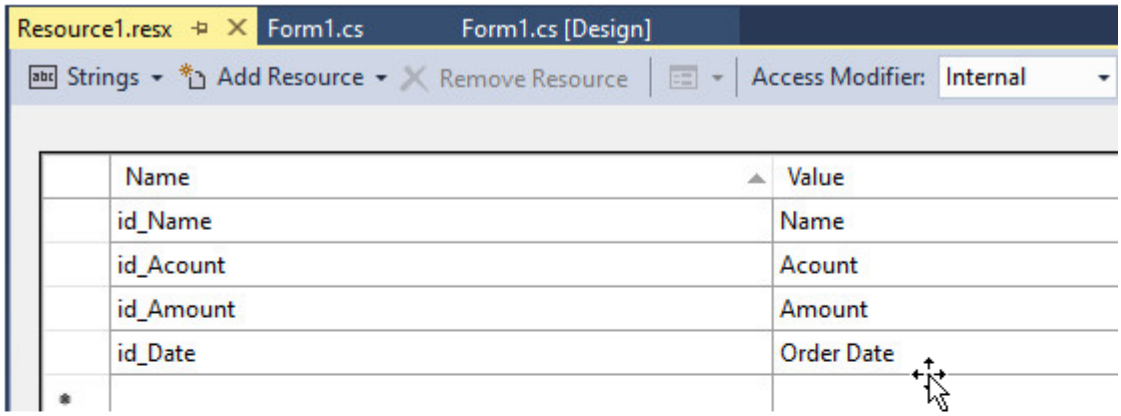
Този файл се създава първоначално за езика по подразбиране, след което се копира под същото име с добавено обозначение за езиковата култура – например *Resource1.resx* и Reso*urce1.bg-BG.resx* и новия файл се добавя като *Existing Item* към проекта.

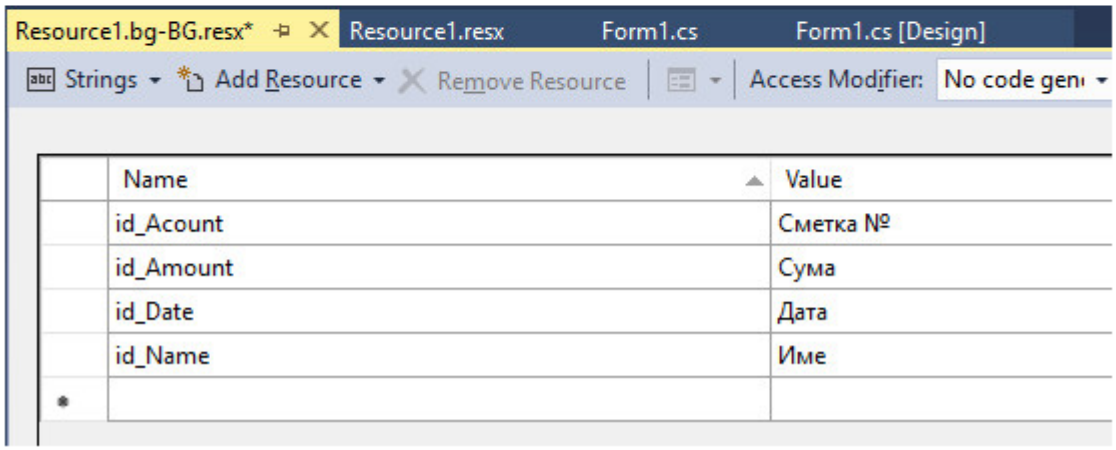
ISSN: 2367–738
Issue 3, pp. 99-110, September, 2018



Фигура 11 Добавяне на езикова версия на ресурсния файл

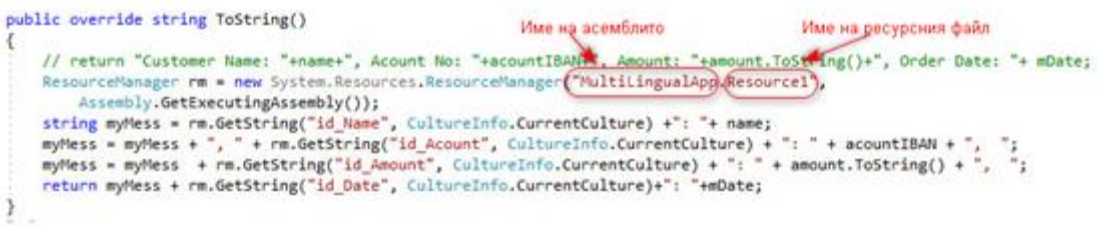
На фиг. 12 са показани съдържанията на основния файл с ресурси и българската му версия след превеждане на значенията.





Фигура 12 Съдържание на ресурсните файлове за двете култури

След генерирането на двата ресурсни файла може да се пренапише отново метода *ToString()* на класа *PaymentOrder*.



Фигура 13 Пренаписан метод ToString() на класа PaymentOrder

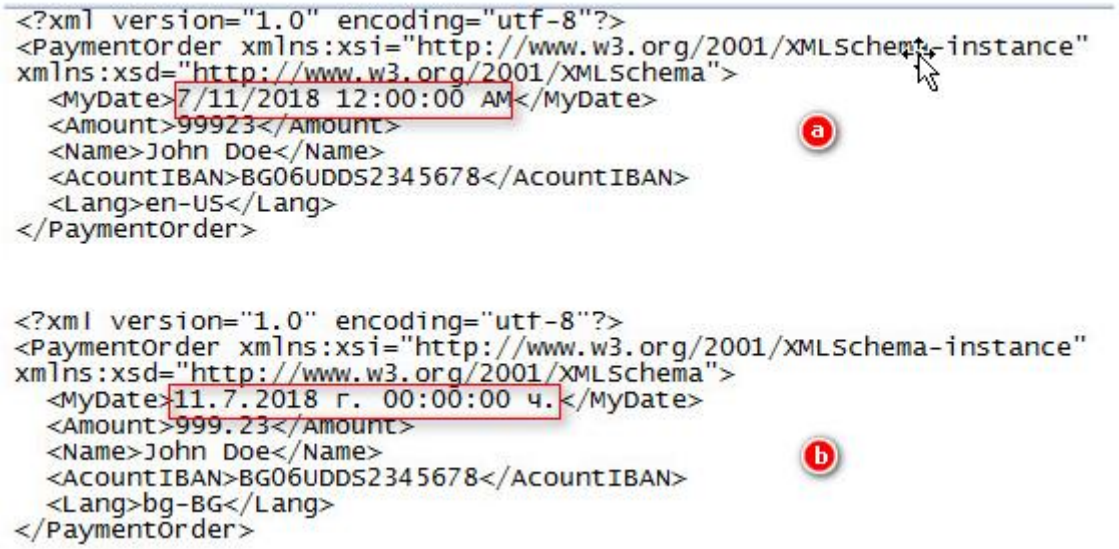
За да се работи по този начин с ресурсни файлове е необходимо да бъдат включени допълнително пространствата на имена *System.Reflection* и *System.Resources.*

СЕРИАЛИЗАЦИЯ, ДЕСЕРИАЛИЗАЦИЯ И ФОРМАТНИ ПРОБЛЕМИ

Използването на различните езикови версии не създава никакви проблеми до момента, в който се налага експортиране на данни, които да бъдат ползвани след това – например в база данни, или както е в примера – с xml сериализация и запис във файл.

Сериализацията в приложението е на базата на създадения клас *PaymentOrder* и за да се осъществи към проекта трябва да се включат допълнително пространствата на имена *System.Xml.Serialization* и *System.IO* (за работа с потоци и файлове) (Price, 2017).

На фиг. 14 са показани съдържанията на xml файлове, създадени при съответните култури. Много ясно се вижда например разликата в записа на съответните дати – разделител и подреждане на деня и месеца.



Фигура 14 Xml файлове с данни от сериализация на обекти тип PaymentOrder

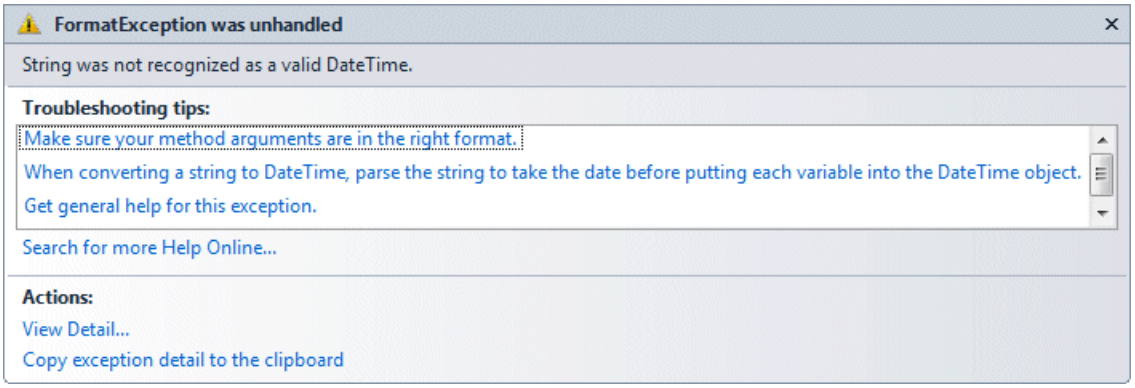
При десериализация на такъв файл и установяване на стойността на контролата DateTimePicker с помощта на

dateTimePicker1.Value = DateTime.Parse(payment2[1]*.MyDate);*

за различните езикови култури датата ще се окаже един път 11.07.2018, а другия път – 07.11.2018, тъй като форматът за *en-US* е mm/dd/year, а за *bg-BG* – dd.mm.year.

[1] payment2 е инстанция на класа PaymentOrder, чийто свойства са получили значения при десериализацията

Сериозен проблем ще има и ако съхранената дата е например 25.07.2017. Тогава при десериализация програмата би спряла изпълнението си със съобщение за грешка



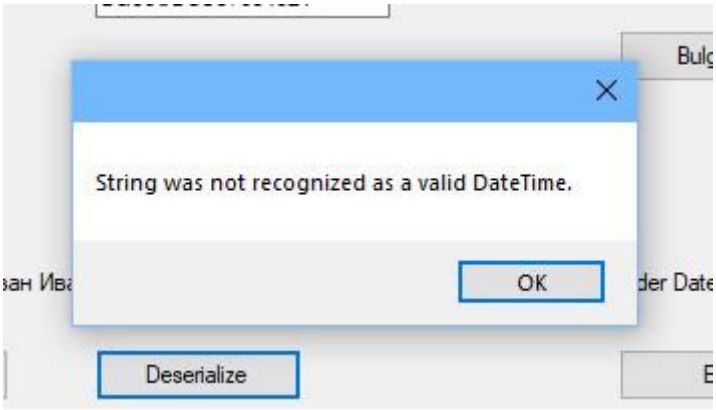
Фигура 15 Изключение в работата на приложението, дължащо се на грешен формат

Затова е препоръчително, винаги когато има подобни потенциално опасни конвертирания, да се прихване и отработи прекъсването (Sharp, 2015)

```
try
{
    dateTimePicker1.Value = DateTime.Parse(payment2.MyDate);

}
catch(Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

Фигура 16 Използване на оператори *try … catch* за отработване на изключението

В този случай ще получим съобщението:



Фигура 17 Съобщение за грешка при неправилен формат

но програмата ще продължи изпълнението.

Проблем би имало и при формат на реални числа, записани и четени с различни езикови култури:

```
1;350;0,1087;
2;350,12;0,6177;
3;350,24;0,2765;
4;350,36;0,417;
5;350,48;0,0526;
6;350,6;0,3615;
7;350,72;0,951;
8;350,84;0,8294;
9;350,96;0,8578;
10;351,08;0,131;
11;351,2;0,6442;
```

```
1;350;0.2529;
2;350.12;0.4018;
3;350.24;0.3943;
4;350.36;0.2913;
5;350.48;0.9837;
6;350.6;0.0589;
7;350.72;0.8452;
8;350.84;0.0344;
9;350.96;0.0043;
10;351.08;0.5277;
11;351.2;0.3198;
```

Фигура 18 Разлика във формата на дробните числа за различни езикови култури

От горното изображение, представляващо запис на две различни измервания на спектър (номер на пиксел, дължина на вълната и стойност на интензитет), ясно се вижда влиянието на езиковата култура. Четенето на такъв файл при интерфейс с различна езикова култура би довело до грешки във визуализацията на спектъра.

От показаното на фиг.14 се вижда, че десетичното число Amount за различните култури ще бъде запазено в единия случай като 999.23, а в другия като 99923. В много приложения с европейска езикова култура авторите категорично указват какъв трябва да бъде десетичният знак и не допускат въвеждане на други разделители.

Тези форматни проблеми могат лесно да се избегнат, ако при сериализацията и десериализацията се предвиди използването на информацията за текущата езикова култура.

За малкото приложение по-горе въпроса се решава, като се въведе допълнителна променлива, в която ще се съхранява текущото значение на културата при сериализацията на обекта и създаването на файла.

Промените в кода, които ще осигурят правилното възпроизвеждане на датата ще бъдат:

```
// dateTimePicker1.Value = DateTime.Parse(payment2.MyDate);
dateTimePicker1.Value = DateTime.Parse(payment2.MyDate, new
System.Globalization.CultureInfo(payment2.Lang));
```

т.е. допълнителното свойство, което беше въведено и което е записано също във файла, позволява винаги правилно да се форматира информацията, съобразно езика на който е създадена, независимо от текущата култура.

ЗАКЛЮЧЕНИЕ

От кратките примери по-горе става ясно, че създаването на многоезични приложения в .NET трябва да следва определена последователност от действия, при това програмистите трябва да се съобразяват с възможните форматни проблеми. Приведеният на фигурите source code (C#) е универсален, позволява коректно решаване на задачата за глобализация и локализация на приложения в реално време.

ISSN: 2367–738
Issue 3, pp. 99-110, September, 2018

ЛИТЕРАТУРА

Albahari, J., Albahari, B. (2018). *C# 7.0 in a Nutshell.* O'Reilly Media, Inc., pp: 761-776

Brunetti, R., Boncinelli, V. (2013). *Exam Ref 70-485 Advanced Windows Store App Development using C# (MCSD)* . O'Reilly Media, Inc., pp: 252-267

Price, M. J. (2017). *C# 7.1 and .NET Core 2.0 – Modern Cross-Platform Development, Third Edition.* Packt., pp: 630-686

Sharp, J. (2015). *Microsoft Visual C# Step by Step (8th Edition) (Developer Reference).* Microsoft Press, pp: 168-191