Jacek Matulewski

Rozdział 24. Serializacja do XML i JSON

W rozdziałach 21 i 22. opisany został projekt *Konsola*, w którym przed zamknięciem aplikacji do pliku XML zapisywany był obiekt `UstawieniaKonsoli`. Plik ten był wczytywany po jej ponownym uruchomieniu i na podstawie zapisanych w nim danych tworzony był obiekt ustawień. Taki scenariusz, w którym stan obiektu o określonym typie jest przechowywany w pliku jest bardzo częsty. W takiej sytuacji izomorficzna musi być struktura klasy lub struktury i pliku, do którego zapisywany jest stan obiektu. Taka czynność, a mianowicie przekształcenie danych do postaci, w której mogą być zapisane w plikach lub przesłane przez sieć do innego komputera, nazywa się **serializacją**.

Jeżeli typ jest znany, to można sobie wyobrazić, że korzystając z technologii *Reflection* jest możliwe napisanie ogólnej metody, która odczytując informacje o własnościach i polach wskazanego obiektu, zapisuje ich wartości do pliku XML. Na szczęście nie trzeba pisać takiego kodu samodzielnie – w platformach .NET i .NET Core jest klasa `XmlSerializer` w przestrzeni `System.Xml.Serialization`. Zapisywanie do pliku w formacie XML to tylko jedna z możliwości. W tym rozdziale przyjrzymy się również serializacji do innego popularnego formatu, a mianowicie JSON, ale możliwy jest również zapis do pliku binarnego i pliku w formacie SOAP.

Ogólnym celem serializacji jest taki zapis stanu obiektu, aby można go było odtworzyć niejako wznawiając działanie programu. Proces odwrotny do serializacji, a więc tworzenie obiektu i ustalanie jego stanu na podstawie zapisanych danych, nazywany jest **deserializacją**. My użyjemy automatycznej serializacji, w której typ serializowanych obiektów nie jest ani ozdabiany atrybutem `Serializable`, ani nie implementuje interfejsu `ISerializable`, ale warto wiedzieć o tych możliwościach kontroli tego procesu. Formalnie rzecz ujmując, osobnym procesem, wykonywanym po serializacji, czyli po przygotowaniu danych zawierających informacje o stanie obiektu, jest formatowanie, czyli zapis danych w pliku o jakimś konkretnym formacie. Tu tych dwóch etapów nie będziemy jednak podkreślać, bo też nie są widoczne w praktyce używania serializatorów w .NET.

XML

Wróćmy wobec tego do wspomnianego projektu *Konsola* z rozdziału 22. Zapisujemy już w nim obiekt ustawień typu `UstawieniaKonsoli` w pliku XML o nazwie *ustawienia.xml*. Naszym celem jest zamiana stosowanych w tej chwili metod, na nowe, w których korzystać będziemy z serializatora XML. W tym celu do folderu *Model* dodajmy plik klasy o nazwie *Serializator.cs* i umieśćmy w nim kod metody `ZapiszXml` widoczny na listingu 24.1.

Listing 24.1. Wczytywanie pliku XML za pomocą serializatora

```
using System;
using System.IO;
using System.Xml.Serialization;

namespace Konsola.Model
{
    public static class Serializacja
```

```
    {
        public static void ZapiszXml(this UstawieniaKonsoli ustawienia,
                                     string ścieżkaPliku)
        {
            StreamWriter strumień = new StreamWriter(ścieżkaPliku);
            try
            {
                XmlSerializer serializator =
                    new XmlSerializer(typeof(UstawieniaKonsoli));
                XmlSerializerNamespaces xmlns = new XmlSerializerNamespaces();
                xmlns.Add("", "");
                serializator.Serialize(strumień, ustawienia, xmlns);
            }
            catch (Exception exc)
            {
                Console.Error.WriteLine("Błąd: " + exc.Message);
            }
            finally
            {
                strumień.Close();
            }
        }
    }
```

W metodzie tej wykorzystaliśmy rozszerzoną konstrukcję `try..catch..finally`, której do tej pory nie poznaliśmy. Sekcje `try` i `catch` działają tak samo, jak dotąd tzn. że sekcja `catch` wykonywana jest w razie zgłoszenia wyjątku w sekcji `try` (zob. rozdział 5.). Natomiast sekcja `finally` wykonywana jest zawsze, bez względu na to, czy wystąpi błąd, czy nie, nawet wtedy, gdy w sekcjach `try` i `catch` obecna jest instrukcja `return`.

Zasadniczą pracę wykonuje klasa `XmlSerializer`. To właśnie jest serializator. Argumentem jego konstruktora jest typ obiektu, który będzie serializowany (obiekt typu `Type` uzyskiwany operatorem `typeof`). Kluczową metodą serializatora jest `Serialize`, do której musimy przesłać strumień wykorzystywany do zapisania tekstu wyjściowego oraz sam serializowany obiekt. W naszym przypadku obiektu ustawień typu `UstawieniaKonsoli`. Metoda nie zwraca żadnej wartości – zapisze informacje o obiekcie do pliku za pomocą wskazanego strumienia.

Zastąpmy teraz wywołanie poprzedniej metody `Zapisz` z klasy `PomocnikXml`, wywołaniem nowej metody `ZapiszXml` z klasy `Serializacja`. Podobnie, jak poprzednia, także nowa metoda jest rozszerzeniem klasy `UstawieniaKonsoli` (modelu), więc ich wywołanie wygląda podobnie. Pokazuje to listing 24.2. Jeżeli uruchomimy teraz program jeden raz, zapisany wcześniejszą plik XML zostanie wczytany (jeszcze przy użyciu starej metody), a przed zakończeniem programu, zostanie zapisany już nową metodą.

Listing 24.2. Zmodyfikowana metoda Main projektu Konsola

```
static void Main(string[] args)
{
    #region Inicjacja
    ...
    #endregion

    Menu kontroler = new Menu(model);
```

```
        kontroler.Uruchom();


        ~~model.Zapisz(ścieżkaPliku);~~
        model.ZapiszXml(ścieżkaPliku);
        Console.WriteLine("Ustawienia zapisane do pliku " + ścieżkaPliku);
    }
```

Uzyskany w ten sposób plik wygląda inaczej niż ten, który tworzyliśmy samodzielnie (listing 24.3), nawet pomimo tego, że pozbyliśmy się przestrzeni nazw, które domyślnie dodawane są do pliku (do tego służy utworzenie pustej przestrzeni nazw i użycie jej w metodzie `Serialize`). W nowym pliku nie ma oczywiście komentarza. Ponadto serializator nie skorzystał z atrybutów – tytuł okna został umieszczony w osobnym elemencie `Tytuł` pod koniec pliku. Inna jest też struktura obu plików – w nowym pliku odpowiada dokładnie typowi serializowanego obiektu (nie ma dodatkowego poziomu tworzonego przez elementy `kolory` i `rozmiary`). Ponadto wszystkie nazwy elementów dokładnie odpowiadają nazwom pól i własności z klasy `UstawieniaKonsoli` i struktury `Rozmiar`.

Listing 24.3. Porównanie struktury pliku zapisanego przez serializator i pliku zapisywanego z użyciem LINQ to XML (z lewej plik zapisany przez serializator, z prawej – zapisany przez nas z użyciem LINQ to XML)

```
<?xml version="1.0" encoding="utf-8"?>
<UstawieniaKonsoli>
  <KolorTła>DarkBlue</KolorTła>
  <KolorCzcionki>Gray</KolorCzcionki>
  <RozmiarOkna>
    <Szerokość>120</Szerokość>
    <Wysokość>30</Wysokość>
  </RozmiarOkna>
  <RozmiarBufora>
    <Szerokość>120</Szerokość>
    <Wysokość>9001</Wysokość>
  </RozmiarBufora>

<Tytuł>c:\Users\jacek\OneDrive\Wydawnictwa\
_C#. Lekcje programowania\źródła\R24
Serializacja\Konsola\Konsola\bin\Debug\netc
oreapp3.1\Konsola.dll</Tytuł>
</UstawieniaKonsoli>
```

```
<?xml version="1.0" encoding="utf-8"
standalone="yes"?>
<!--Ustawienia zapisane przez program
Konsola.dll-->
<ustawienia
tytuł="c:\Users\jacek\OneDrive\Wydawnictwa\
_C#. Lekcje programowania\źródła\R24
Serializacja\Konsola\Konsola\bin\Debug\netc
oreapp3.1\Konsola.dll">
  <kolory>
    <tło>DarkBlue</tło>
    <czcionka>Gray</czcionka>
  </kolory>
  <rozmiary>
    <okno>
      <X>120</X>
      <Y>30</Y>
    </okno>
    <bufor>
      <X>120</X>
      <Y>9001</Y>
    </bufor>
  </rozmiary>
</ustawienia>
```

Zmiana struktury oznacza, że nie możemy już użyć wcześniejszej metody wczytującej obiekt ustawień z pliku XML, a musimy przygotować nową korzystającą z serializatora. Taką metodę pokazuje listingu 24.4. Należy ją dodać do klasy `Serializacja` z pliku *Serializacja.cs*. Ponownie używamy konstrukcji `try..catch..finally`, aby zapewnić zamknięcie strumienia.

Listing 24.4. Metoda deserializująca zapisany plik XML do obiektu ustawień

```
public static UstawieniaKonsoli CzytajXml(string ścieżkaPliku)
{
    StreamReader reader = new StreamReader(ścieżkaPliku);
    try
    {
        XmlSerializer ser = new XmlSerializer(typeof(UstawieniaKonsoli));
        UstawieniaKonsoli ustawienia = (UstawieniaKonsoli)ser.Deserialize(reader);
```

```
            return ustawienia;
        }
        catch(Exception exc)
        {
            Console.Error.WriteLine("Błąd: " + exc.Message);
            return PomocnikUstawieńKonsoli.UstawieniaDomyślne;
        }
        finally
        {
            reader.Close();
        }
    }
```

Do deserializacji wykorzystujemy tę samą klasę `XmlSerializer`, której wcześniej użyliśmy do serializacji. Tym razem wykorzystujemy jednak jej metodę `Deserialize`. Jej jedynym argumentem jest strumień, z którego będzie odczytywany plik XML (do jego stworzenia korzystamy z klasy `StreamReader`). Metoda zwraca obiekt typu `object`, który trzeba samodzielnie zrzutować na właściwy typ tj. na `UstawieniaKonsoli`.

Teraz wystarczy podmienić metodę służącą do wczytywania pliku XML (listing 24.6), aby transformacja projektu została zakończona.

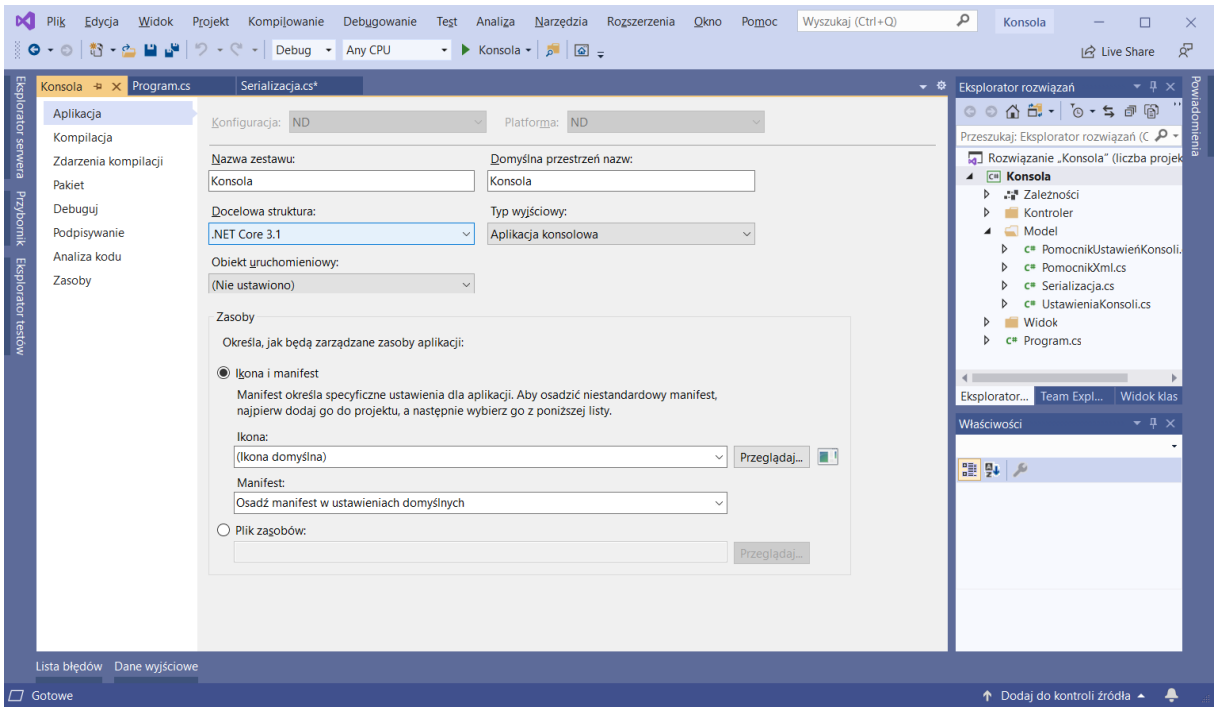
Listing 24.6. Zmiana metody wczytującej ustawienia z pliku XML

```
    static void Main(string[] args)
    {
        #region Inicjacja
        UstawieniaKonsoli model;
        try
        {
            ~~model = PomocnikXml.Czytaj(ścieżkaPliku);~~
            model = Serializacja.CzytajXml(ścieżkaPliku);
        }
        catch (Exception exc)
        {
            model = PomocnikUstawieńKonsoli.UstawieniaDomyślne;
            Console.Error.WriteLine("Błąd: " + exc.Message + ". Stosuję ustawienia
domyślne");
        }
        #endregion

        Menu kontroler = new Menu(model);
        kontroler.Uruchom();

        model.ZapiszXml(ścieżkaPliku);
        Console.WriteLine("Ustawienia zapisane do pliku " + ścieżkaPliku);
    }
```

JSON

W platformie .NET możliwa była serializacja do plików w formacie XML, formacie SOAP, który bardzo przypominał XML oraz pliku binarnego. W platformie .NET Core od wersji 3.0 do tego zestawu dołączył także

format JSON (ang. *JavaScript Object Notation*). To możliwość potrzebna i wykorzystywana szczególnie w aplikacjach webowych ASP.NET Core. Wcześniej do obsługi formatu JSON korzystano zazwyczaj z bardzo popularnego darmowego pakietu NuGet o nazwie *Newtonsoft.Json*[1]. Od wersji 3.0 platformy .NET Core, pakiet ten został włączony do samej platformy, choć z kilkoma zmianami. Dostępny jest w przestrzeni nazw `System.Text.Json`.



Rysunek 24.1. Ustawienia projektu, m.in. wersja platformy .NET Core

Aby go przetestować, zamieńmy format pliku, do którego zapisywane są ustawienia w aplikacji *Konsola*. Ponieważ będę korzystał z wbudowanego serializatora, należy upewnić się, że aplikacja przeznaczona jest dla platformy .NET Core 3.0 lub nowszej. Możemy to zrobić w ustawieniach projektu (rysunek 24.1). Następnie do pliku *Serializacja.cs* dodajmy dwie metody `ZapiszJson` i `CzytajJson` widoczne na listingu 24.7. Są bardzo podobne do metod z listingów 24.1 i 24.4, tylko tym razem wykorzystywany jest serializator JSON. Serializator ten nie korzysta ze strumieni, a pobiera lub zwraca zwykły tekst, który musimy sami odczytać lub zapisać do pliku tekstowego.

Listing 24.7. *******

```
using System;
using System.IO;
using System.Text.Json;
using System.Xml.Serialization;

namespace Konsola.Model
{
    public static class Serializacja
    {
        public static void ZapiszXml ...
        public static UstawieniaKonsoli CzytajXml ...

        public static void ZapiszJson(this UstawieniaKonsoli ustawienia,
                                      string ścieżkaPliku)
```

[1] To najczęściej pobierany pakiet na stronie *nuget.org/packages*. W marcu 2020 pobrany został już 425 milionów razy.

```
        {
            JsonSerializerOptions jsonOptions = new JsonSerializerOptions()
            {
                WriteIndented = true
            };
            string jsonText = JsonSerializer.Serialize(ustawienia, jsonOptions);
            File.WriteAllText(ścieżkaPliku, jsonText);
        }

        public static UstawieniaKonsoli CzytajJson(string ścieżkaPliku)
        {
            string jsonText = File.ReadAllText(ścieżkaPliku);
            UstawieniaKonsoli ustawienia =
                JsonSerializer.Deserialize<UstawieniaKonsoli>(jsonText);
            return ustawienia;
        }
    }
}
```

W metodzie `Main` z klasy `Program` zastąpmy wywołania metod `ZapiszXml` i `CzytajXml` przez nowe metody `ZapiszJson` i `CzytajJson` (listing 24.8). Zmieńmy również przechowywaną w polu `ścieżkaPliku` nazwę pliku, do którego zapisywane są ustawienia. I uruchommy program.

Listing 24.8. Zamiana metod serializacji

```
using System;

namespace Konsola
{
    using Model;
    using Kontroler;

    class Program
    {
        private const string ścieżkaPliku = "ustawienia.json";

        static void Main(string[] args)
        {
            #region Inicjacja
            UstawieniaKonsoli model;
            try
            {
                model = Serializacja.CzytajJson(ścieżkaPliku);
            }
            catch (Exception exc)
            {
                model = PomocnikUstawieńKonsoli.UstawieniaDomyślne;
                Console.Error.WriteLine("Błąd: " + exc.Message + ". Stosuję ustawienia
domyślne");
            }
```

```
            #endregion

            Menu kontroler = new Menu(model);
            kontroler.Uruchom();

            model.ZapiszJson(ścieżkaPliku);
            Console.WriteLine("Ustawienia zapisane do pliku " + ścieżkaPliku);
        }
    }
}
```

Ponieważ nie ma jeszcze pliku *ustawienia.json*, wczytanie ustawień z pliku nie powiedzie się i użyte zostaną ustawienia domyślne. Plik zostanie zapisany, gdy zamkniemy aplikację wybierając z jej menu pozycję o numerze `0`. Sprawdźmy jego zawartość. Ku naszemu rozczarowaniu w tym pliku znajdziemy tylko parę nawiasów klamrowych, bez żadnej zapisanej wartości. Powodem jest to, że w odróżnieniu od wcześniej używanego serializatora XML, nowy serializator JSON obsługuje jedynie właściwości. A w klasie `UstawieniaKonsoli` wykorzystywane są tylko pola. To oczywisty brak, który został już zgłoszony społeczności rozwijającej .NET Core i podobno prace nad uzupełnieniem braku już trwają[2].

Jeżeli mimo to już teraz chcemy skorzystać z nowego serializatora, aby zapisać ustawienia do pliku w formacie JSON, musimy zmienić klasę `UstawieniaKonsola` tak, żeby jego publiczne pola zmienić we własności[3]. Pokazuje to listing 24.9. To pociąga za sobą konieczność kilku zmian w kodzie dwóch plików: *PomocnikUstawieńKonsoli.cs* oraz *PomocnikXml.cs*. Drugi nie jest już używany, więc po prostu go usuńmy z projektu (należy go zaznaczyć w *Eksploratorze rozwiązań* i nacisnąć klawisz *Del*). Natomiast w pliku *PomocnikUstawieńKonsoli.cs*, w którym znajdują się definicje dwóch własności `UstawieniaBieżące` i `UstawieniaDomyślne`, należy tworząc obiekty `Rozmiar` (własności `RozmiarOkna` i `RozmiarBufora` obiektu ustawień) podać jawnie typ tworzonego obiektu (listing 24.10).

Listing 24.9. Dostosowanie typu serializowanego obiektu do ograniczeń serializatora JSON

```
using System;

namespace Konsola.Model
{
    public struct Rozmiar
    {
        public int Szerokość { get; set; }
        public int Wysokość { get; set; }

        public override string ToString()
        {
            return "" + Szerokość + " x " + Wysokość;
        }
    }

    public class UstawieniaKonsoli : ICloneable
    {
        public ConsoleColor KolorTła { get; set; }
        public ConsoleColor KolorCzcionki { get; set; }
```

[2] Zob. *https://github.com/dotnet/runtime/issues/876*

[3] Jeżeli nie chcielibyśmy tego robić, koniecznie byłoby przygotowanie obiektu pośredniego (z własnościami), do którego zapisywane byłyby dane z obiektu typu `UstawieniaKonsoli`.

```
        public Rozmiar RozmiarOkna { get; set; }
        public Rozmiar RozmiarBufora { get; set; }
        public string Tytuł { get; set; }

        public override string ToString() ...
        public object Clone() ...
    }
}
```

Listing 24.10. Zmiany wynikające z zastąpienia pól własnościami

```
using System;

namespace Konsola.Model
{
    class PomocnikUstawieńKonsoli
    {
        public static UstawieniaKonsoli UstawieniaBieżące
        {
            get
            {
                return new UstawieniaKonsoli()
                {
                    KolorTła = Console.BackgroundColor,
                    KolorCzcionki = Console.ForegroundColor,
                    RozmiarOkna = new Rozmiar()
                                  {
                                      Szerokość = Console.WindowWidth,
                                      Wysokość = Console.WindowHeight
                                  },
                    RozmiarBufora = new Rozmiar()
                                    {
                                       Szerokość = Console.BufferWidth,
                                       Wysokość = Console.BufferHeight
                                    },
                    Tytuł = Console.Title
                };
            }
        }

        public static UstawieniaKonsoli UstawieniaDomyślne { get; } =
            new UstawieniaKonsoli()
            {
                KolorTła = ConsoleColor.Black,
                KolorCzcionki = ConsoleColor.Gray,
                RozmiarOkna = new Rozmiar() { Szerokość = 120, Wysokość = 30 },
                RozmiarBufora = new Rozmiar() { Szerokość = 120, Wysokość = 9001 },
                Tytuł = System.Reflection.Assembly.GetEntryAssembly().Location
```

```
                };
            }
        }
```

Jeżeli teraz ponownie uruchomimy program, serializator będzie próbował wczytać pusty plik JSON, co spowoduje, że rozmiary okna i bufora będą zerowe. Można temu zaradzić, wybierając pierwszą pozycję menu, ładującą domyślne wartości. Zamknięcie teraz aplikacji spowoduje, że na dysku powstanie plik *ustawienia.json* widoczny na listingu 24.11. Jak widać, rzeczywiście ma format JSON.

Listing 24.11. Plik JSON zapisany przez serializator

```
{
  "KolorT\u0142a": 1,
  "KolorCzcionki": 7,
  "RozmiarOkna": {
    "Szeroko\u015B\u0107": 120,
    "Wysoko\u015B\u0107": 30
  },
  "RozmiarBufora": {
    "Szeroko\u015B\u0107": 120,
    "Wysoko\u015B\u0107": 9001
  },
  "Tytu\u0142": "c:\\Users\\jacek\\OneDrive\\Wydawnictwa\\_C#. Lekcje
programowania\\\u017Ar\u00F3d\u0142a\\R24
JSON\\Konsola_2a\\Konsola\\bin\\Debug\\netcoreapp3.1\\Konsola.dll"
}
```

Zadanie

Korzystając z serializacji do formatu JSON zapisz do pliku listę osób z projektu omawianego w poprzednim rozdziale.