XML Serialization

Jarosław Kuchta

Ten dokument opisuje konstrukcję serializatora XML naśladującego standardowy XmlSerializer, ale wypełniający jego braki i zapewniający większą elastyczność.

# Potrzeba alternatywnego serializatora

Serializator XML firmy Microsoft pozostaje niezmieniony od początku mimo oczywistych braków, z których najważniejsze to:

* brak obsługi kolekcji generycznych (np. List<string>),
* brak obsługi słowników,
* brak obsługi typów i właściwości interfejsowych,
* brak serializacji kolekcji tylko do odczytu,
* brak obsługi referencji zwrotnych (gdy klasa odwołuje się do samej siebie bezpośrednio lub pośrednio),
* brak możliwości zarządzania kolejnością atrybutów,
* brak możliwości własnej konwersji właściwości złożonych na tekst.

XmlSerializer został napisany w celu szybkiego przekazywania danych przez sparowane operacje serializacji i deserializacji. Przy tworzeniu instancji serializatora podaje się typ, który ma być serializowany i deserializowany, oraz inne rozpoznawane typy. Typy są poddawane analizie strukturalnej i jest generowany wyspecjalizowany moduł implementujący operacje serializacji i deserializacji, które są wywoływane przez XmlSerializer. Moduł ten jest niewidoczny dla programisty, ale za to zapewnia wysoką wydajność zapisu/odczytu.

Problemy się pojawiają wówczas, gdy programista chce dopasować uzyskany format XML do swoich potrzeb, np. chce wygenerować plik XML do czytania przez inną aplikację, która narzuca swój format. Jest to trudne, a często wręcz niemożliwe. Jeśli struktura typu poddawanego analizie ma elementy, które nie są obsługiwane lub brak jest danych umożliwiających poprawną deserializację, to moduł implementujący nie zostanie wygenerowany i serializacja jest niemożliwa.

Inna kwestia to czytelność uzyskanego kodu XML dla człowieka. Standardowy serializator XML potrafi wstawiać do kodu nieoczekiwane atrybuty, jak własne deklaracje przestrzeni nazw czy określenia formatu dla dat. Domyślnie ignoruje spacje na początku i końcu łańcuchów tekstowych. Kolejność generowania atrybutów jest nieokreślona. To wszystko nieco zmniejsza czytelność.

W związku z powyższymi brakami powstało przynajmniej kilka alternatywnych serializatorów innych autorów. Najbardziej popularne to:

* SharpSerializer (910 tys. pobrań) – autor Paweł Idzikowski,
* ExtendedXmlSerializer (626 tys. pobrań) – autorzy Wojciech Nagórski i Michael DeMond,
* Platform.Xml.Serialization (321 tys. pobrań) – autor Thong Nguyen,
* XmlSerializerHelper (108 tys. pobrań) – autor Thomas Galliker,
* XSerializer (451 tys. pobrań) – autor Brian Friesen.

Jako alternatywę dla XML trzeba też rozważyć XAML, który jest specyficzną formą XML i został wprowadzony przez Microsoft jako format zapisu projektów komponentów wizualnych aplikacji WPF.

Warto też wspomnieć o alternatywnych względem XML formatach, jak JSON czy YAML.

## SharpSerializer

SharpSerializer to rozwiązanie open source dla zapisu danych w XML lub we własnym formacie binarnym. Ma on takie same możliwości jak standardowy XmlSerializer, a ponadto obsługuje:

* tablice wielowymiarowe, tablice zagnieżdżone, tablice tablic,
* właściwości polimorficzne, tj. takie, których wartości mogą być pochodne od zadeklarowanego typu właściwości,
* typy generyczne z atrybutami polimorficznymi (np. GenericClass<AbstractClass>),
* kolekcje i słowniki generyczne (np. List<string>, Dictionary<string, int>),
* kolekcje polimorficzne (np. Dictionary<IKeyInterface, AbstractClass>),
* kolekcje generyczne z argumentami polimorficznymi (np. IMyGenericInterface<MyAbstractClass>[,,]).

SharpSerializer został napisany dla potrzeb technologii Silverlight, gdzie nie było serializatora binarnego, zapewniając znacznie większą wydajność przekazywania danych niż w formacie tekstowym XML. Dzisiaj to rozwiązanie jest przestarzałe.

## ExtendedXmlSerializer

ExtendedXmlSerializer to konfigurowany serializer XML wspierający:

* deserializację XML uzyskanego ze standardowego serializatora Microsoftu,
* serializację klas, struktur, klas generycznych, typów prymitywnych, list i słowników generycznych, tablic i typów wyliczeniowych,
* serializację klas z właściwościami interfejsowymi,
* serializację referencji zwrotnych i referencji przez identyfikatory,
* deserializację starszych wersji XML,
* szyfrowanie właściwości,
* serializację dopasowaną przez użytkownika,
* rozpoznawanie atrybutów identyfikacyjnych: XmlElementAttribute, XmlRootAttribute, XmlTypeAttribute,
* rozpoznawanie dodatkowych atrybutów: XmlIgnoreAttribute, XmlAttributeAttribute, XmlEnumAttribute,
* własne metody zapisu/odczytu nawet, gdy klasa nie implementuje interfejsu IXmlSerializable.

Konfigurowalność serializatora ExtendedXmlSerializer oznacza, że można (i należy) tworzyć instancje serializatora przez utworzenie instancji klasy ConfigurationContainer i wywołanie szeregu jej metod. Mechanizm ten jest nazywany Fluent API. Przykład

IExtendedXmlSerializer serializer = new ConfigurationContainer().ConfigureType<Subject>()

.Member(x => x.Message)

.Name("Text")

.Create();

Takie podejście jest bardzo elastyczne, jednak pracochłonne. Przy dużej liczbie składowych użycie konfiguratora staje się osobnym projektem.

## Platform.Xml.Serialization

Ten serializator stanowi część biblioteki między-platformowej zawierającej użyteczne klasy i rozszerzenia dla C# i .NET. Biblioteka została napisana w latach 2003-2008 i wydana jako open source na licencji BSD w ramach organizacji Platform.NET. Copyright obejmuje lata 2003-2013.

Rozwiązanie przestarzałe. Na dodatek dokumentacja nie istnieje.

## XmlSerializerHelper

XmlSerializerHelper to nakładka na standardowy XmlSerializer. Daje możliwości dodawania metod rozszerzających. Dostępny na GitHubie. Dopasowany do .NET Standard 2.0.

Brak dokumentacji dyskwalifikuje projekt.

## XSerializer

XSerializer to uniwersalny serializator XML/JSON. Obsługuje właściwości i typy, których nie obsługuje standardowy serializator, takie jak interfejsy i słowniki. Może zapewnić proste zastąpienie standardowego serializatora Microsoftu. Używa tych samych atrybutów [XmlElement], [XmlAttribute] itp.

Przy serializacji JSON zapewnia większą wydajność niż serializator firmy Newtonsoft. Daje też lepsze wsparcie przy deserializacji zmiennych i pól typu dynamic.

Wspiera koncepcję szyfrowania pól i właściwości opatrzonych atrybutem [Encrypt] poprzez interfejs IEncryptionMechanism.

Aktualna wersja ma numer 0.4.4. Z tego wniosek, że sam autor nie uznaje projektu za dojrzały.

## XAML Reader/Writer

Alternatywą dla serializacji XML może być zapis/odczyt danych w formacie XAML. Ten format został wprowadzony przez Microsoft jako język oparty na XML opisu interfejsu użytkownika dla potrzeb Windows Presentation Foundation. XAML charakteryzuje się wysokim stopniem uporządkowania i niezawodną obsługą różnych konstrukcji programistycznych. Od „zwykłego” XML odróżnia go przede wszystkim:

* Automatyczna obsługa wielu różnych przestrzeni nazw – zarówno standardowych w .NET, jak i tworzonych przez programistę. Każdy zapisany obiekt jest przypisany do odpowiedniej przestrzeni nazw, w tym do nazw własnych przestrzeni aplikacji.
* Automatyczne zapisywanie prostych właściwości w postaci atrybutów XML.
* Możliwość definiowania konwerterów klas złożonych na wartości tekstowe (i z powrotem), co upraszcza zapisywanie takich klas.

Jeśli właściwość jest złożona, to jest zapisywana jako element XML (z nazwą identyfikującą klasę wartości) wewnątrz elementu, którego nazwa identyfikuje samą właściwość i obiekt nadrzędny. Daje to łatwą serializację właściwości polimorficznych, gdyż komponent zagnieżdżony może być różnych klas pochodnych. Przykład:

<MyClass …>

<MyClass.MyProperty>

<MyComponent …>

</MyComponent>

</MyClass.MyProperty>

</MyClass>

* Obsługa właściwości dołączanych (attached), czyli właściwości pewnej klasy (zazwyczaj zamkniętej) zaimplementowanych w innej klasie. Przykładem są właściwości Grid.Row i Grid.Column dołączane do każdego komponentu wizualnego.
* Obsługa rozszerzeń składniowych przez zapis w nawiasach klamrowych. Najczęściej wykorzystywane są rozszerzenia powiązań komponentów przez {Binding}. Powiązania są utrzymywane po załadowaniu komponentów i w ten sposób właściwości jednych komponentów wiążą się z innymi właściwościami innych komponentów tak, że zmiana jednych pociąga za sobą zmianę innych. Dla przykładu można powiązać treść wyświetlanego komponentu z zasobem aplikacji. Jeśli zasoby się zmienią (np. poprzez wybór języka użytkownika), to treść wyświetlana też się zmieni.

Początkowo pliki formacie XAML były edytowane ręcznie lub tworzone za pomocą narzędzi projektowania graficznego Visual Studio. Odczyt plików był automatyczny przy ładowaniu komponentów interfejsu użytkownika lub ręczny poprzez klasę XamlReader. Z czasem pojawiła się też klasa XamlWriter umożliwiająca programistom samodzielne zapisywanie zarówno komponentów interfejsu, jak instancji zwykłych klas (POCO).

Użycie formatu XAML jest ograniczone do .NET Framework, gdyż wymaga biblioteki PresentationFramework (niedostępnej w .NET Core). Podejmowane są próby otwartoźródłowych implementacji zapisu XAML np. w Portable.Xaml.XamlWriter.

## Formaty JSON i YAML

Format JSON (JavaScript Object Notation) to lekki format zapisu i wymiany danych obiektowych w postaci tekstowej będący podzbiorem składni języka JavaScript. Zarówno XML, jak i JSON są stosowane w technologiach usług webowych. Starsze technologie (SOAP, WCF) wykorzystywały XML, nowsze (REST) wykorzystują JSON. Pliki (i pakiety danych) JSON są mniejsze niż odpowiadające im pliki XML, gdyż nie zawierają znaczników końcowych. Co do łatwości czytania danych przez człowieka, zdania są różne. Z jednej strony struktura JSON jest bardziej przejrzysta, z drugiej trudniej jest odnaleźć koniec struktury (właśnie przez brak znaczników końcowych), co może mieć znaczenie przy dłuższych plikach.

Podobnym w strukturze do JSON jest YAML. Jest nieco bardziej czytelny dla człowieka, ale trudniejszy w parsowaniu.

Aktualnie dostępne są dwie implementacje serializatora JSON – jedna firmy Microsoft w przestrzeni nazw System.Text.Json, druga firmy Newtonsoft w przestrzeni nazw Newtonsoft.Json. Serializator Newtonsoft jest bardziej zaawansowany. Podstawowe różnice są następujące:

* Newtonsoft.Json potrafi ignorować wielkość liter przy deserializacji.
* Newtonsoft.Json zezwala na komentarze i spacje końcowe.
* System.Text.Json stosuje specjalny zapis znaków nie-ASCII i znaków charakterystycznych dla HTML dla zapobiegania atakom typu cross site scripting.
* System.Text.Json ma ograniczenie głębokości zagnieżdżenia struktur obiektowych do 64. Newtonsoft.Json nie ma ograniczenia.
* Newtonsoft.Json domyślnie serializuje pola (a nie tylko właściwości).
* Newtonsoft.Json i System.Text.Json stosują inne opcje do ignorowania wartości null.
* Newtonsoft.Json pozwala na serializację i deserializację liczb w cudzysłowach. Przy System.Text.Json można to osiągnąć przez opcje i atrybuty.
* System.Text.Json nie obsługuje atrybutów z przestrzeni nazw System.Runtime.Serialization, takich jako [DataMember] i [IgnoreDataMember].
* Oba serializatory różnią się w obsłudze referencji zwrotnych.
* System.Text.Json nie obsługuje typów Datatable, DbNull, TimeSpan, TimeZoneInfo, BigInteger.
* System.Text.Json jest bardziej wydajny przy serializacji i deserializacji od Newtonsoft.Json.

Brak znaczników klas elementów w JSON utrudnia obsługę właściwości i kolekcji polimorficznych. Wymaga to zaimplementowania własnych konwerterów rozróżniających klasy elementów przez serializację dodatkowych pól informujących o typach.

## Testy i wnioski

Przeprowadzono testy trzech serializatorów XML (dla których dostępna jest dokumentacja) oraz serializatora Newtonsoft.Json. Testu z XAML nie udało się przeprowadzić na projekcie, którego targetem jest .Net 6 (brak biblioteki).

Standardowy **XmlSerializer** pokazał błąd przy tworzeniu instancji:

**NotSupportedException**: Cannot serialize member DocumentModel.Settings.DocVars of type DocumentModel.DocumentVariables, because it implements IDictionary.

**SharpSerializer** wygenerował plik o nieoczekiwanej strukturze:

<Complex name="Root" type="DocumentModel.Document, DocumentModel, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" id="1">

<Properties>

<Null name="Name" />

<Null name="Path" />

<Simple name="FullName" value="" />

<Complex name="DocumentProperties">

<Properties>

<Reference name="Document" id="1" />

<Simple name="Title" value="Tytuł dokumentu" />

**ExtendedXmlSerializer** wyświetlił komunikat:

**System.InvalidOperationException**: 'An attempt was made to format 'System.Nullable`1[System.Int32] ActiveRecord' as an attribute, but there is not a registered converter that can convert its values to a string. Please ensure a converter is registered for the type 'System.Nullable`1[System.Int32]' by adding a converter for this type to the converter collection in the ConverterExtension.'

Znaleziono w sieci wersję nowszą (3.7.6), jeszcze nie opublikowaną, co do której autor (Mike-E-angelo) zadeklarował:

Accounted for converter registration with nullable structures #534

Mimo tego, błąd występował dalej.

**Newtonsoft. JSON** serializator wygenerował plik poprawny, ale dla czytelności brakuje w nim nazw elementów:

{

"DocumentProperties": {

"Title": "Tytuł dokumentu",

"Subject": "Jakiś temat",

"Category": "Jakaś kategoria",

"ContentStatus": "Jakiś stan",

"Description": "To jest jakiś komentarz, nie bardzo wiadomo jaki.",

"Keywords": "CustomProperties, ExtendedProperties",

"Creator": "Jarosław Kuchta",

"Created": "2021-06-08T13:25:00+02:00",

"LastModifiedBy": "Jarosław Kuchta",

"Modified": "2021-06-08T13:32:00+02:00",

"LastPrinted": "2021-06-08T13:29:00+02:00",

"Revision": "3",

"Application": "Microsoft Office Word",

"ApplicationVersion": "16.0000",

"Company": "Jakaś firma",

"DocumentSecurity": "RecommendedReadOnly",

"HeadingPairs": [

{

"Name": "Tytuł",

"Count": 1

}

],

**Wniosek**: Żaden z analizowanych tu serializatorów nie nadaje się do testowanego rozwiązania. Modyfikacja istniejącego rozwiązania (open source) jest zazwyczaj trudna, a przy rozwiązaniu *żywym* (ExtendedXmlSerializer) nie wchodzi w grę.

**Wniosek końcowy**: potrzebny jest własny serializator XML.

# Wymagania dla własnego serializatora XML

Wymagania dla własnego serializatora XML są następujące:

1. Musi być możliwy do podstawienia w miejsce standardowego serializatora, tzn. musi mieć takie same konstruktory, właściwości i metody publiczne.
2. Musi mieć wszystkie możliwości standardowego serializatora, w tym:
   1. obsługiwać wszystkie atrybuty konfiguracyjne z przestrzeni nazw System.Xml.Serialization,
   2. obsługiwać metody serializacji warunkowej (ShouldSerialize),
   3. obsługiwać metody własnej serializacji z interfejsu IXmlSerializable,
   4. obsługiwać zdarzenia deserializacji (XmlDeserializationEvents),
   5. obsługiwać mapowanie typów (XmlMapping),
3. Musi wypełnić braki obsługi typów standardowego serializatora, w tym:
   1. przy serializacji kolekcji generycznych (np. List<string>) zapewnić opcjonalne pomijanie właściwości Capacity,
   2. przy deserializacji kolekcji elementów abstrakcyjnych i polimorficznych (np. List<AbstractClass>) rozpoznawać klasy pochodne umieszczone w tym samym asemblacie i dopuścić podawanie klas pochodnych z innych asemblatów,
   3. umożliwiać serializację i deserializację kolekcji tylko do odczytu,
   4. przy serializacji i deserializacji słowników umożliwić wybór jednej lub kilku właściwości elementu jako klucza tak, by można było zmienić słownik na kolekcję,
4. Musi zapewniać obsługę typów i właściwości interfejsowych przy podaniu klas implementujących.
5. Musi zapewniać wierność serializacji i deserializacji właściwości tekstowych:
   1. zawierających znaki Unicode,
   2. rozpoczynających się lub kończących się spacjami,
   3. zawierających znaki niewidoczne lub specjalne dla notacji XML.
6. Powinien zapewniać obsługę referencji, w tym referencji zwrotnych przy podaniu właściwości (jednej lub kilku) reprezentujących klucz obiektu.
7. Powinien zapewnić programiście możliwość zarządzania kolejnością atrybutów, a domyślnie generować je w kolejności deklaracji właściwości przy pierwszeństwie właściwości odziedziczonych z klas nadrzędnych.
8. Powinien zapewnić możliwość własnej konwersji właściwości złożonych na tekst.
9. Powinien zapewnić możliwość deklaracji wartości domyślnej [DefaultValue] właściwości (pomijanej przy serializacji).
10. Powinien zapewnić możliwość deklaracji właściwości reprezentującej zawartość elementu [ContentProperty], dla której serializator nie generuje nazwy właściwości.
11. Powinien zapewnić szerokie możliwości wyboru opcji serializacji i deserializacji, w tym:
    1. wyboru sposobu formatowania nazw elementów, atrybutów i wartości wyliczanych (bez zmian, zamiana na wielkie/małe litery, pierwsza litera wielka/mała) przy serializacji oraz uwzględniania/ignorowania wielkości liter przy deserializacji.
    2. wyboru lub pomijania pól wszystkich przy serializacji,
    3. wyboru lub pomijania właściwości nieoznaczonych atrybutami [XmlElement], [XmlAttribute],
    4. rozpoznawania lub ignorowania wybranych atrybutów z przestrzeni nazw System.Runtime.Serialization i System.ComponentModel.DataAnnotations,
    5. wyboru formatu serializacji daty i czasu (domyślnie międzynarodowy W3C),
    6. wyboru tekstu reprezentującego wartości logiczne true i false przy serializacji, a przy deserializacji kilku zestawów,
    7. kultury zapisu i odczytu danych liczbowych.
12. Wydajność serializacji i deserializacji nie jest krytyczna, ale wskazana jest optymalizacja wydajności przy zachowaniu funkcjonalności.

# Interfejs standardowego serializatora

Dla utrzymania zgodności własnego serializatora z klasą standardowego serializatora (System.Xml.Serialization.XmlSerializer) istotne jest zadeklarowanie dla klasy własnego serializatora tych samych konstruktorów i metod publicznych oraz tych samych zdarzeń (w klasie System.Xml.Serialization.XmlSerializer nie ma właściwości). Poniższe trzy tabele wyjaśniają szczegóły.

Tab. . Konstruktory klasy XmlSerializer

|  |  |
| --- | --- |
| Konstruktor | Znaczenie |
| XmlSerializer () | Inicjuje nowe wystąpienie klasy XmlSerializer. |
| XmlSerializer (Type type) | Inicjuje nowe wystąpienie klasy XmlSerializer, które może serializować obiekty określonego typu w dokumentach XML i deserializować dokumenty XML do obiektów określonego typu. |
| XmlSerializer (XmlTypeMapping) | Inicjuje wystąpienie klasy XmlSerializer przy użyciu obiektu, który mapuje jeden typ na inny. |
| XmlSerializer (Type type, String defaultNamespace) | Inicjuje nowe wystąpienie klasy XmlSerializer, które może serializować obiekty określonego typu w dokumentach XML i deserializować dokumenty XML do obiektów określonego typu. Określa domyślną przestrzeń nazw dla wszystkich elementów XML. |
| XmlSerializer (Type type, Type[] extraTypes) | Inicjuje nowe wystąpienie klasy XmlSerializer, które może serializować obiekty określonego typu w dokumentach XML i deserializować dokumenty XML do obiektu określonego typu. Jeśli właściwość lub pole zwraca tablicę, to parametr extraTypes określa obiekty, które można wstawić do tej tablicy. |
| XmlSerializer (Type type, XmlAttributeOverrides overrides) | Inicjuje nowe wystąpienie klasy XmlSerializer, które może serializować obiekty określonego typu w dokumentach XML i deserializować dokumenty XML do obiektów określonego typu. Każdy obiekt do serializacji może zawierać wystąpienia klas, które to przeciążenie może zastąpić innymi klasami. |
| XmlSerializer (Type type, XmlRootAttribute root) | Inicjuje nowe wystąpienie klasy XmlSerializer, które może serializować obiekty określonego typu w dokumentach XML i deserializować dokument XML do obiektu określonego typu. Określa również klasę stosowaną jako element główny XML. |
| XmlSerializer (Type type, XmlAttributeOverrides overrides, Type[] extraTypes, XmlRootAttribute root, String  defaultNamespace) | Inicjuje nowe wystąpienie klasy XmlSerializer, które może serializować obiekty typu Object w wystąpieniach dokumentów XML i deserializować wystąpienia dokumentów XML do obiektów typu Object. Każdy obiekt, który ma być serializowany, może zawierać wystąpienia klas, które to przeciążenie zastępuje przez inne klasy. To przeciążenie określa również domyślną przestrzeń nazw dla wszystkich elementów XML i klasę stosowaną jako element główny XML. |
| XmlSerializer (Type type, XmlAttributeOverrides overrides, Type[] extraTypes, XmlRootAttribute root, String defaultNamespace, String location) | Inicjuje XmlSerializer nowe wystąpienie klasy, które może serializować obiekty typu Object w wystąpieniach dokumentów XML i deserializować wystąpienia dokumentów XML do obiektów typu Object. Każdy obiekt, który ma być serializowany, może zawierać wystąpienia klas, które to przeciążenie zastępuje przez inne klasy. To przeciążenie określa również domyślną przestrzeń nazw dla wszystkich elementów XML i klasę stosowaną jako element główny XML. Ostatni parametr określa lokalizację typów: |

Tab. . Metody publiczne klasy XmlSerializer

|  |  |
| --- | --- |
| Metoda | Znaczenie |
| bool CanDeserialize (System.Xml.XmlReader xmlReader) | Określa czy ten serializator może deserializować dokument dostępny przez dany element XmlReader. |
| XmlSerializationReader CreateReader () | Zwraca obiekt używany do odczytywania dokumentu XML do serializacji. |
| XmlSerializationWriter CreateWriter () | Po przesłonięciu w klasie pochodnej zwraca składnik zapisywania używany do serializacji obiektu. W klasie standardowego serializatora nie zaimplementowana. |
| Deserialize (Stream stream) | Deserializuje dokument XML dostępny przez określony element Stream. |
| Deserialize (TextReader reader) | Deserializuje dokument XML dostępny przez określony element TextReader. |
| Deserialize (XmlSerializationReader reader) | Deserializuje dokument XML dostępny przez określony element XmlSerializationReader. |
| Deserialize (XmlReader  reader) | Deserializuje dokument XML dostępny przez określony element XmlReader. |
| Deserialize (XmlReader  reader, String style) | Deserializuje dokument XML dostępny przez określony element XmlReader w określonym stylu kodowania. |
| Deserialize (XmlReader  reader, XmlDeserializationEvents events) | Deserializuje dokument XML dostępny przez określony element XmlReader umożliwiając zastępowanie zdarzeń występujących podczas deserializacji. |
| Deserialize (XmlReader  reader, String style, XmlDeserializationEvents events) | Deserializuje dokument XML dostępny przez określony element XmlReader w określonym stylu kodowania umożliwiając zastępowanie zdarzeń występujących podczas deserializacji. |

Tab. . Zdarzenia występujące w klasie XmlSerializer

|  |  |
| --- | --- |
| Metoda | Znaczenie |
| XmlAttributeEventHandler UnknownAttribute | Występuje, gdy serializator napotka atrybut XML nieznanego typu podczas deserializacji. |
| XmlElementEventHandler UnknownElement | Występuje, gdy serializator napotka element XML nieznanego typu podczas deserializacji. |
| XmlNodeEventHandler UnknownNode | Występuje, gdy serializator napotka węzeł XML nieznanego typu podczas deserializacji. |
| UnreferencedObjectEventHandler UnreferencedObject | Występuje podczas deserializacji strumienia XML zakodowanego przy użyciu protokołu SOAP, gdy serializator napotka rozpoznany typ, który nie jest używany lub do którego nie ma odwołań. |

# Konstrukcja własnego serializatora

Własnemu serializatorowi nadano nazwę QXmlSerializer (dla odróżnienia od standardowego serializatora).

Ze względu na wielość metod własny zdecydowano na utworzenie własnego serializatora w kilku plikach:

* QXmlSerializer.Base.cs – zawiera definicje skopiowane ze standardowego serializatora,
* QXmlSerializer.Main.cs – zawiera główne metody serializacji i deserializacji,
* QXmlSerializer.Serialization.cs – zawiera właściwe metody serializacji,
* QXmlSerializer.Deserialization.cs – zawiera właściwe metody deserializacji,
* QXmlSerializer.Helper.cs – zawiera metody pomocnicze.

## Definicje bazowe

Ponieważ znaczna część definicji standardowego serializatora opiera się na konstruktorach, nie można zadeklarować kompletnego interfejsu zapewniającego pełną zgodność własnego serializatora ze standardowym. W zamian za to zdefiniowano bazową część własnego serializatora w osobnym pliku QXmlSerializer.Base.cs. Część ta zawiera definicje zmiennych ze standardowego serializatora, konstruktory (patrz tab. 1), metody Serialize i Deserialize (patrz tab. 2) oraz deklaracje zdarzeń (tab. 3) standardowego serializatora.

Z prywatnych zmiennych standardowego serializatora zachowano trzy, którym nadano dostęp chroniony. Zmienna \_events z konieczności musi być polem, pozostałe zmienne to właściwości.

Tab. . Zmienne przeniesione z klasy XmlSerializer do części QXmlSerializer.Base

|  |  |  |
| --- | --- | --- |
| Metoda | Typ | Znaczenie |
| \_events | XmlDeserializationEvents | Definiuje pola zdarzeń wg tab. 3. |
| DefaultNamespace | string? | Domyślna przestrzeń nazw przekazywana przez konstruktory |
| DefaultNamespaces | XmlSerializerNamespaces | Zawiera dwie domyślne przestrzenie nazw Xml: xsi oraz xsd, wykorzystywane do wyrażania wartości nil, oraz typów danych xml. |

Aby implementacja konstruktorów była możliwa w innej części, wszystkim konstruktorom przypisano wywołanie jednej z dwóch metod inicjacji zdefiniowanych jako partial. Podobnie metody serializacji i deserializacji przypisano do metod partial.

Tab. . Metody cząstkowe zadeklarowane w części QXmlSerializer.Base

|  |  |
| --- | --- |
| Metoda | Znaczenie |
| Init (Type type, XmlAttributeOverrides? overrides, Type[]? extraTypes, XmlRootAttribute? root, string? defaultNamespace, string? location) | Metoda inicjalizacji serializatora. Parametry jak w najbardziej rozbudowanej wersji konstruktora serializatora klasy XmlSerializer (patrz tab. 1) |
| Init (XmlTypeMapping xmlTypeMapping) | Metoda inicjalizacji serializatora. Parametry jak dla jednej z wersji serializatora konstruktora klasy XmlSerializer (patrz tab. 1) |
| SerializeObject (XmlWriter xmlWriter, object? obj, XmlSerializerNamespaces? namespaces, string? encodingStyle, string? id) | Metoda serializacji obiektu. Parametry jak dla najbardziej rozbudowanej metody Serialize standardowego serializatora (patrz tab. 2) |
| object? DeserializeObject (XmlReader xmlReader, string? encodingStyle, XmlDeserializationEvents events) | Metoda deserializacji obiektu. Parametry jak dla najbardziej rozbudowanej metody Deserialize standardowego serializatora (patrz tab. 2) |
| bool CanDeserialize (XmlReader xmlReader) | Metoda sprawdzenia, czy obiekt może być deserializowany. Parametry jak dla metody CanDeserialize standardowego serializatora (patrz tab. 2) |

Deklaracje zdarzeń odwołują się wprost do pola \_events.

## Część główna

Część QXmlSerializer.Main.cs zawiera dodatkowe konstruktory oraz główne metody serializacji i deserializacji.

Dodatkowe konstruktory mają jeden dodatkowy parametr SerializationOptions, który reguluje opcje serializacji i deserializacji. Wszystkie konstruktory zadeklarowane w części bazowej są powtórzone z dodanym parametrem SerializationOptions. Podobnie obie metody Init zostały przekierowane do nowych metod Init z dodanym parametrem SerializationOptions.

Metoda Init z parametrem XmlTypeMapping na razie pozostaje niezaimplementowana.

Metoda Init z parametrem Type (i pozostałymi) jest podstawową metodą inicjacji serializatora. Tworzy ona komponent XmlSerializationInfoMapper, który analizuje typy elementów przeznaczonych do serializacji, przypisuje im nazwy elementów i atrybutów oraz rejestruje je w słowniku KnownTypes. Do analizy jest przekazany typ główny i typy dodatkowe podane jako parametry metody Init. Ponieważ XmlSerializationInfoMapper przy analizie wykorzystuje niektóre opcje serializacji, dlatego opcje te muszą być dostępne przy inicjacji i w konstruktorach. Jeśli inicjacja następuje przez standardowe konstruktory, to wykorzystywane są opcje domyślne.

## Opcje serializacji

Klasa SerializationOptions zawiera opcje serializacji i deserializacji (patrz tab. 6).

Tab. . Opcje serializacji i deserializacji w klasie SerializationOptions

|  |  |  |  |
| --- | --- | --- | --- |
| Metoda | Typ | Domyślnie | Znaczenie |
| IgnoreTypesWithout­ParameterlessConstructor | bool | false | Typy do deserializacji muszą mieć konstruktor bez parametrów. Jeśli typ znaleziony podczas skanowania dostępnych typów nie ma publicznego konstruktora bez parametrów, zostanie zgłoszony wyjątek, chyba że ta opcja jest ustawiona. |
| IgnoreMultiple­TypeRegistration | bool | false | Zazwyczaj typy są rejestrowane do serializacji/deserializacji z unikalnymi nazwami. Jeśli więc aplikacja próbuje zarejestrować ten sam typ pod dwiema różnymi nazwami, zgłaszany jest wyjątek, chyba że ta opcja jest ustawiona. Należy zauważyć, że gdy typ jest zarejestrowany z wieloma nazwami, można go rozpoznać pod różnymi nazwami podczas deserializacji. W przypadku serializacji używana jest tylko jedna nazwa. |
| AcceptAllNot­IgnoredProperties | bool | false | Czy właściwości nieoznaczone żadnym z atrybutów Xml są akceptowane do serializacji. |
| AcceptSimple­PropertiesAsAttributes | bool | false | Czy właściwości proste nieoznaczone żadnym z atrybutów Xml są akceptowane do serializacji jako atrybuty. |
| AttributeNameCase | Serialization­Case | Unchanged | Czy i jak mają się zmienić nazwy atrybutów XML podczas serializacji. |
| ­ElementNameCase | Serialization­Case | Unchanged | Czy i jak mają się zmienić nazwy elementów XML podczas serializacji. |
| ­EnumNameCase | Serialization­Case | Unchanged | Czy i jak mają się zmienić nazwy wartości wyliczanych podczas serializacji. |
| IgnoreCaseOnEnum | bool | false | Czy ignorować wielkość liter w nazwach wartości wyliczanych.  W wartościach typu Boolean zawsze są ignorowane. |
| PrecedePropertyName­WithElementName | bool | false | Czy nazwa właściwości serializowana jako element XML ma być poprzedzona nazwą serializowanej klasy. Znak kropki ('.') jest używany jako separator. |
| ItemTag | string? | null | Ciąg znaków do oznaczania elementów w kolekcjach. Jeśli nie określono, elementy są serializowane bezpośrednio. |
| Culture | CultureInfo | Invariant­Culture | Kultura używana w liczbach i datach podczas serializacji/deserializacji. Domyślną jest InvariantCulture. |
| IgnoreUnknownElements | bool | false | Określa, że gdy deserializator znajdzie nieznany element XML, przechodzi do elementu zamykającego (lub ignoruje, jeśli jest to pusty element). |
| UseNilValue | bool | false | Określa, że gdy atrybut ma wartość null, to trzeba wypisywać specjalną wartość xsi:nil. |
| FalseString | string | "false" | Łańcuch wypisywany jako wartość False podczas serializacji. |
| TrueString | string | "true" | Łańcuch wypisywany jako wartość True podczas serializacji. |
| DateTimeFormat | string | "yyyy-MM-ddTHH:mm:sszzz" | Format zapisu daty i czasu podczas serializacji. Przy deserializacji wiele formatów jest akceptowalnych. |
| ShouldSerializeMethodPrefix | string | "ShouldSerialize" | Początek nazwy metody, która określa, czy właściwość ma być serializowana. |

## Metody pomocnicze

Metody pomocnicze zostały zdefiniowane jako statyczne, aby mogły być wywoływane zarówno przez sam serializator, jak i pozostałe komponenty.

Tab. . Metody pomocnicze zadeklarowane w części QXmlSerializer.Helper

|  |  |
| --- | --- |
| Metoda | Znaczenie |
| string ChangeCase (string str, SerializationCase nameCase) | Zmiana wielkości liter nazwy w zależności od opcji. |
| string FirstLetterToLower (string text) | Zmiana pierwszej litery łańcucha na małą. |
| string FirstLetterToUpper (string text) | Zmiana pierwszej litery łańcucha na wielką. |
| bool IsFirstLetterLower (string text) | Czy pierwsza litera łańcucha jest mała? |
| bool IsFirstLetterUpper (string text) | Czy pierwsza litera łańcucha jest wielka? |

## Mapowanie typów

Mapowanie typów polega na utworzeniu złożonego słownika, który z jednej strony dla określonego typu przypisze informację wykorzystywaną przy serializacji, a z drugiej strony dla określonej nazwy elementu XML przypisze informację wykorzystywaną przy deserializacji.

Słownik ten jest reprezentowany przez klasę KnownTypesDictionary wypełnianą przez obiekt klasy XmlSerializationInfoMapper. Klasa KnownTypesDictionary dla każdego typu napotkanego przy analizie przypisuje element klasy SerializationTypeInfo.

Tab. 8. Właściwości klasy SerializationTypeInfo

|  |  |  |
| --- | --- | --- |
| Właściwość | Typ | Znaczenie |
| Type | Type | Typ, którego ta informacja dotyczy. |
| QualifiedName | QualifiedName | Nazwa kwalifikowana przestrzenią nazw. Jeśli typ zawiera atrybut [XmlRoot], to nazwą jest wartość tego atrybutu, a jak nie, to nazwa samego typu. |
| KnownConstructor | ConstructorInfo | Informacja o konstruktorze bezparametrowym (z refleksji typu). |

jest pochodna od generycznej klasy TypesDictionary<ItemType>, która w istocie składa się z dwóch słowników:

* TypeIndexedItems: Dictionary<Type, ItemType>,
* NameIndexedItems: SortedDictionary<QualifiedName, ItemType>.

Klasa TypesDictionary implementuje trzy interfejsy:

* IEnumerable<ItemType>,
* IDictionary<QualifiedName, ItemType>,
* IDictionary<Type, ItemType>.

Pierwszy słownik rejestruje informacje o serializowanych typach wg samych typów, drugi porządkuje tę samą informację wg nazw kwalifikowanych. Nazwa kwalifikowana, QualifiedName, jest klasą składającą się z dwóch pól:

* Namespace: string,
* Name: string.

gdzie Namespace reprezentuje przestrzeń nazw (języka C#), do której należy typ, a Name reprezentuje nazwę typu. Klasa QualifiedName została zadeklarowana osobno jako implementująca interfejs IComparable, aby można jej było użyć przy sortowaniu kluczy w klasie SortedDictionary. Należy ją odróżnić od standardowej klasy XmlQualifiedName, w której właściwość Namespace reprezentuje przestrzeń nazw Xml.