DATA ACCESS

Die Datenzugriffsberatung mit DR. HOLGER SCHWICHTENBERG

WEITERE NEUERUNGEN IN SYSTEM.TEXT.JSON 9.0

Nachgelegt

Microsoft hat in System.Text.Json 9.0 zwischen dem Release Candidate vom September 2024 und der final veröffentlichten Version noch einige weitere Neuerungen ergänzt.

In der Datenzugriffskolumne der dotnetpro-Ausgabe 12/2024 berichtete ich, dass Microsoft in der Version 9.0 von System.Text.Json erst vier neue Features eingebaut hat:

- Einrückungen anpassen mit *IndentCharacter* und *IndentSize*,
- Einstellungen wie bei Web-APIs mit JsonSerializerOptions.Web,
- Berücksichtigung von Nullable-Kontext und *[DisallowNull]*,
- JSON-Schema exportieren mit JsonSchemaExporter.

Diese Aussagen basierten auf dem Preview-Stand vom 10. September 2024. Bis zur Release-Version am 12. November 2024 hat Microsoft noch einiges nachgelegt.

Berücksichtigung verpflichtender Konstruktorparameter

Neben der Option *RespectNullableAnnotations* = *true* bietet System.Text.Json in Version 9.0 eine weitere optionale Regelverschärfung: *RespectRequiredConstructorParameters* = *true* sorgt dafür, dass beim Deserialisieren alle verpflichtenden Konstruktorparameter befüllt werden. Den Einsatz zeigt Listing 1.

Genau wie die Option *RespectNullableAnnotations* kann man auch dieses Prüf-Feature in der Projektdatei global einschalten:

```
<ItemGroup>
  <RuntimeHostConfigurationOption
    Include="System.Text.Json.Serialization.
    RespectRequiredConstructorParametersDefault"
    Value="true" />
</ItemGroup>
```

Microsoft empfiehlt im Dev-Blog unter [1], diese Option für neue Projekte zu nutzen. Zur Vermeidung eines Breaking Change hat Microsoft an dieser Stelle darauf verzichtet, den Standard zu ändern. ▶

Listing 1: Einsatz von RespectRequiredConstructorParameters

```
public void JSON_RequiredConstructorParameters()
{
  JsonSerializerOptions options = new() {
    RespectRequiredConstructorParameters = true };

  string json = """
    {"Name":"Dr. Holger Schwichtenberg"}
    """; // ID ist Pflicht im Konstruktor, fehlt aber
         // hier im JSON

  CUI.H1("Deserialisieren von einem Objekt mit
    verpflichtenden Konstruktorparametern");
  try
  {
    // Laufzeitfehler: "was missing required
    // properties including: 'ID'."
    Entwickler e = JsonSerializer.Deserialize<
      Entwickler>(json, options);
    Console.WriteLine(e);
  }
  catch (Exception ex)
  {
    CUI.Error(ex);
  }
}
```

Listing 2: Einsatz von [JsonStringEnumMemberName]

```
[JsonConverter(typeof(JsonStringEnumConverter))]
enum Note
{
  [JsonStringEnumMemberName("Sehr gut")] // NEU!
  SehrGut = 1,
  Gut = 2,
  Befriedigend = 3,
  Ausreichend = 4,
  [JsonStringEnumMemberName("Nicht bestanden")]
    // NEU!
  Mangelhaft = 5,
}

public void JSON_EnumNames()
{
  CUI.Demo();

  // Serialisierung
  CUI.H2("Serialisierung");
  string json1 =
    JsonSerializer.Serialize(Note.SehrGut);
  Console.WriteLine(json1); // "Sehr Gut"

  string json2 =
    JsonSerializer.Serialize(Note.Gut);
  Console.WriteLine(json2); // "Gut"

  string json3 =
    JsonSerializer.Serialize(Note.Mangelhaft);
  Console.WriteLine(json3); // "Nicht bestanden"

  // Deserialisierung
  CUI.H2("Deserialisierung");
  Note note1 =
    JsonSerializer.Deserialize<Note>(json1);
  Console.WriteLine(note1.ToString()); // SehrGut

  Note note2 =
    JsonSerializer.Deserialize<Note>(json2);
  Console.WriteLine(note2.ToString()); // Gut

  Note note3 =
    JsonSerializer.Deserialize<Note>(json3);
  Console.WriteLine(note3.ToString()); // Mangelhaft

  try
  {
    string json4 = @"""Mangelhaft""";
      // Das geht nicht mehr, weil
      // [JsonStringEnumMemberName] verwendet wird!
    Note note4 =
      JsonSerializer.Deserialize<Note>(json4);
    Console.WriteLine(note4.ToString());
  }
  catch (Exception ex)
  {
    CUI.Error(ex); // The JSON value could not be
                   // converted to
                   // NET9_Console.FCL90.FCL9_JSON+Note
  }
}
```

```
JSON_EnumNames

Serialisierung
"Sehr gut"
"Gut"
"Nicht bestanden"
Deserialisierung
SehrGut
Gut
Mangelhaft
System.Text.Json.JsonException: The JSON value could not be converted to NET9_Console.FCL90.FCL9_JSON+Note.
```

Ausgabe von Listing 2 (Bild 1)

Listing 3: Einsatz von AllowOutOfOrderMetadataProperties=true

```
public void JSON_OutOfOrderMetadataProperties()
{
  CUI.Demo();

  // Erstelle Objekt
  var consultant = new Consultant()
  {
    ID = 42,
    FullName = "Holger Schwichtenberg",
    Salutation = "Dr.",
    PersonalWebsite = "www.dotnet-doktor.de"
  };
  consultant.Languages = ["C#", "Visual Basic .NET",
    "JavaScript", "TypeScript"];

  // Serialisierungsoptionen festlegen
  var options = new JsonSerializerOptions
  {
    // ReferenceHandler = ReferenceHandler.Preserve,
    WriteIndented = true,
    IndentCharacter = ' ', // NEU! --> ACHTUNG: Als
      // IndentCharacter werden aber nur ein Leerzeichen
      // ' ' oder ein Tabulator '\t' unterstützt!!!
    IndentSize = 4,
  };

  // Serialisieren in JSON
  CUI.H2("Serialisieren --> $type am Anfang");
  var json = JsonSerializer.Serialize<Person>(
    consultant, options);
  Console.WriteLine(json);

  // Ausgabe:
  // {
  //   "$type": "Consultant",
  //   "Languages": [
  //     "C#",
  //     "Visual Basic .NET",
  //     "JavaScript",
  //     "TypeScript"
  //   ],
  //   "PersonalWebsite": "www.dotnet-doktor.de",
  //   "ID": 42,
  //   "FullName": "Holger Schwichtenberg",
  //   "Salutation": "Dr.",
  //   "Address": null
  // }

  // Deserialisieren aus JSON
  CUI.H2("Deserialisieren mit $type am Anfang");
  var obj = JsonSerializer.Deserialize<Person>(json);
  Console.WriteLine(obj); // Consultant!

  string json2 = """
  {
    "Languages": [
```

```
JSON_OutOfOrderMetadataProperties

Serialisieren --> $type am Anfang
{
    "$type": "Consultant",
    "Languages": [
        "C#",
        "Visual Basic .NET",
        "JavaScript",
        "TypeScript"
    ],
    "PersonalWebsite": "www.dotnet-doktor.de",
    "ID": 42,
    "FullName": "Holger Schwichtenberg",
    "Salutation": "Dr.",
    "Address": {
        "Country": null,
        "City": "Essen"
    }
}
Deserialisieren mit $type am Anfang
Consultant 42: Dr. Holger Schwichtenberg wohnt in Essen (www.dotnet-doktor.de)
-> Sprachen: C#+Visual Basic .NET+JavaScript+TypeScript
Deserialisieren mit $type am Ende
Consultant 42: Dr. Holger Schwichtenberg wohnt in Essen (www.dotnet-doktor.de)
-> Sprachen: C#+Visual Basic .NET+JavaScript+TypeScript
```

Ausgabe von Listing 3 (Bild 2)

```
      "C#",
      "Visual Basic .NET",
      "JavaScript",
      "TypeScript"
    ],
    "PersonalWebsite": "www.dotnet-doktor.de",
    "ID": 42,
    "FullName": "Holger Schwichtenberg",
    "Salutation": "Dr.",
    "Address": null,
    "$type": "Consultant"
  }
  """;

  // Deserialisieren aus JSON
  // Führt ohne AllowOutOfOrderMetadataProperties=
  // true zum Fehler: 'The metadata property is
  // either not supported by the type or is not
  // the first property in the deserialized JSON
  // object
  CUI.H2("Deserialisieren mit $type am Ende");
  JsonSerializerOptions options2 = new() {
    AllowOutOfOrderMetadataProperties = true };
  var obj2 = JsonSerializer.Deserialize<Person>(
    json2, options2);
  Console.WriteLine(obj2); // Consultant!
}
```

Anpassung der Serialisierung von Enumerationsmitgliedsnamen

System.Text.Json kann man ab Version 9.0 mit der neuen Annotation *[JsonStringEnumMemberName]* vermitteln, dass bei der Serialisierung von Namen aus Aufzählungstypen (Enumerationen) nicht der im Programmcode vergebene Name, sondern ein anderer Text genutzt werden soll. Auch bei der Deserialisierung werden diese Namen verwendet.

In dem folgenden Beispiel soll es in dem JSON-Dokument etwas abweichende Texte für zwei Noten geben: Beim Aufzählungsmitglied *SehrGut* soll es im Dokument die korrekte Schreibweise „Sehr gut“ geben, die im Programmcode nicht erlaubt wäre. Beim Aufzählungsmitglied *Mangelhaft* soll im Dokument der Text „Nicht bestanden“ stehen.

Wenn Namen mit *[JsonStringEnumMemberName]* geändert werden, werden die ursprünglichen Enumerationsmitgliedsnamen nicht mehr bei der Deserialisierung erkannt, siehe den vergeblichen Versuch in Listing 2, den Text „Mangelhaft“ zu deserialisieren, und die zugehörige Ausgabe in Bild 1.

Anpassen der Position von Typ-Metadaten

In System.Text.Json Version 7.0 hatte Microsoft die Unterstützung für Polymorphismus eingeführt, indem man beim Serialisieren einen Typnamen mit *$type* angeben kann, der dann beim Deserialisieren zur Instanzierung des korrekten Typs führt. System.Text.Json setzt *$type* immer vor die erste ▶

Listing 4: Einsatz von DeepEquals

```
public void JSON_DeepEquals()
{
  // Erstelle Objekt
  var consultant = new Consultant()
  {
    ID = 42,
    FullName = "Holger Schwichtenberg",
    Salutation = "Dr.",
    PersonalWebsite = "www.dotnet-doktor.de"
  };
  consultant.Languages = ["C#",
    "Visual Basic .NET", "JavaScript",
    "TypeScript"];

  // Serialisieren in JSON
  CUI.H2("Objekt serialisieren");
  var json =
    JsonSerializer.Serialize<Person>(consultant);
  Console.WriteLine(json);

  // Lade in JSONElement
  JsonDocument doc1 = JsonDocument.Parse(json);
  JsonElement root1 = doc1.RootElement;

  // Lade in noch ein JSONElement
  JsonDocument doc2 = JsonDocument.Parse(json);
  JsonElement root2 = doc1.RootElement;

  bool deepEqual1 =
    JsonElement.DeepEquals(root1, root2);
  Console.WriteLine(deepEqual1); // true

  CUI.H2("Geändertes Objekt serialisieren");
  consultant.Address =
    new Address() { City = "Essen" };
  var json3 =
    JsonSerializer.Serialize<Person>(consultant);
  Console.WriteLine(json3);

  // Lade geändertes Objekt in ein JSONElement
  JsonDocument doc3 = JsonDocument.Parse(json3);
  JsonElement root3 = doc3.RootElement;

  bool deepEqual2 =
    JsonElement.DeepEquals(root1, root3);
  Console.WriteLine(deepEqual2); // false

}
```

```
JSON_MultiDocReader
Number 42
StartObject
PropertyName ID
Number 123
PropertyName FullName
String Holger Schwichtenberg
PropertyName Languages
StartArray
String C#
String Visual Basic .NET
String JavaScript
String TypeScript
EndArray
EndObject
StartArray
Number 1
Number 2
Number 3
EndArray
Null
StartObject
EndObject
StartArray
EndArray
```

Ausgabe von Listing 6 (Bild 3)

Listing 5: Beispiel für ein JSON-Dokument ohne eindeutigen Wurzelknoten

```
42
{ "ID": "123", "FullName": "Holger Schwichtenberg",
  "Languages": [
    "C#",
    "Visual Basic .NET",
    "JavaScript",
    "TypeScript"
]
}
[1,2,3]
null {} []
```

Property und erwartete bisher dies auch beim Deserialisieren genau in dieser Reihenfolge. Mit der neuen Option *AllowOutOfOrderMetadataProperties = true* kann System.Text.Json auch JSON-Dokumente deserialisieren, in denen die Position von *$type* eine andere ist, weil andere JSON-Bibliotheken

Listing 6: Einsatz von AllowMultipleValues beim Utf8JsonReader

```
public void JSON_MultiDocReader()
{
  CUI.Demo();

  ReadOnlySpan<byte> utf8Json1 =
    """
    42
    { "ID": 123, "FullName": "Holger Schwichtenberg",
      "Languages": [
        "C#",
        "Visual Basic .NET",
        "JavaScript",
        "TypeScript"
    ]
    }
    [1,2,3]
    null {} []
    """u8;

  JsonReaderOptions options = new() {
    AllowMultipleValues = true };
    // NEU für Multi-Dokumente!
  Utf8JsonReader reader =
    new(utf8Json1, options);

  while (reader.Read())
  {
    Console.Write(reader.TokenType);

    switch (reader.TokenType)
    {
      case JsonTokenType.PropertyName:
      case JsonTokenType.String:
      {
        string? text = reader.GetString();
        Console.Write(" ");
        Console.Write("\e[32m");
        Console.Write(text);
        break;
      }

      case JsonTokenType.Number:
      {
        int intValue = reader.GetInt32();
        Console.Write(" ");
        Console.Write("\e[32m");
        Console.Write(intValue);
        break;
      }

      // ... (weitere Token-Typen)

    }

    Console.Write("\e[0m");
    Console.WriteLine();
  }
}
```

Listing 7: Einsatz von DeserializeAsyncEnumerable() mit topLevelValues = true

```
public async Task JSON_MultiDocDeserialisierung()
{
  CUI.Demo();

  CUI.H2("Liste von Zahlen");
  ReadOnlySpan<byte> utf8Json1 = """ 1 2 3"""u8;
  using var stream1 =
    new MemoryStream(utf8Json1.ToArray());

  await foreach (int item in
      JsonSerializer.DeserializeAsyncEnumerable<int>(
      stream1, topLevelValues: true))
  {
    Console.WriteLine(item);
  }

  CUI.H2("Liste von Arrays");
  ReadOnlySpan<byte> utf8Json2 =
    """[42] [42,43] [42,43,44] [42,42,42] []"""u8;
  using var stream2 =
    new MemoryStream(utf8Json2.ToArray());

  await foreach (int[] array in
      JsonSerializer.DeserializeAsyncEnumerable<int[]>(
      stream2, topLevelValues: true))
  {
    Console.WriteLine(
      "Array mit " + array.Length + " Element(en):");
    foreach (var num in array)
    {
      CUI.LI(num);
    }
  }

  CUI.H2("Liste von Objekten");
  ReadOnlySpan<byte> utf8Json3 =
    """
    { "ID": 1, "FullName": "Holger Schwichtenberg",
      "Languages": [
        "C#",
        "Visual Basic .NET",
        "JavaScript",
        "TypeScript" ],
      "Address": { "City" : "Essen" }
    }

    { "ID": 2, "FullName": "Rainer Stropek",
      "Languages": [
        "C#",
        "Rust",
        "Go",
        "JavaScript",
        "TypeScript" ],
      "Address": { "City" : "Linz" }
    }
    """u8;

  using var stream3 =
    new MemoryStream(utf8Json3.ToArray());

  await foreach (Consultant c in
      JsonSerializer.DeserializeAsyncEnumerable<
      Consultant>(stream3, topLevelValues: true))
  {
    Console.WriteLine(c);
    foreach (var l in c.Languages)
    {
      CUI.LI(l);
    }
  }
}
```

das Dokument erzeugt haben, siehe Listing 3 und die zugehörige Ausgabe in Bild 2.

Microsoft warnt davor, dass die Aktivierung von *AllowOutOfOrderMetadataProperties* zu Leistungseinbußen führen kann, da der Serialisierer mehr puffern muss, denn er muss den Typ ja kennen, bevor er das Objekt instanzieren kann.

Falls man auch den Namen *$type* ändern will oder muss: Das geht schon seit Version 7.0 mit:

```
[JsonPolymorphic(TypeDiscriminatorPropertyName = "$xyz")]
```

Neue Methode DeepEquals() in der Klasse JsonElement

Die Klasse *JsonElement* bietet eine neue Methode *DeepEquals()*, mit der man zwei Elemente und alle darunterliegenden Elemente (tiefer Vergleich) vergleichen kann. Den Einsatz dieser Methode zeigt Listing 4. Analog gab es zuvor schon *DeepEquals()* in der Klasse *JsonNode*.

Lesen von Multi-JSON-Dokumenten

Ein JSON-Dokument benötigt normalerweise einen eindeutigen Wurzelknoten. Aber was tun, wenn es diesen nicht gibt, sondern man ein JSON-Dokument bekommt, das technisch gesehen aus mehreren JSON-Dokumenten besteht? Ein Beispiel ist in Listing 5 gezeigt.

System.Text.Json bietet seit Version 9.0 eine Lösung an, auch solche „Multi-JSON-Dokumente“ zu lesen. Das geht zum einen im Utf8JsonReader mit Aktivierung der neuen Option *AllowMultipleValues* = *true*, siehe Listing 6 und dessen Ausgabe in Bild 3. ▶

```
JSON_MultiDocDeserialisierung

Liste von Zahlen
1
2
3
Liste von Arrays
Array mit 1 Element(en):
- 42
Array mit 2 Element(en):
- 42
- 43
Array mit 3 Element(en):
- 42
- 43
- 44
Array mit 3 Element(en):
- 42
- 42
- 42
Array mit 0 Element(en):
Liste von Objekten
Consultant 1: Holger Schwichtenberg wohnt in Essen -> Sprachen: C#+Visual Basic .NET+JavaScript+TypeScript
- C#
- Visual Basic .NET
- JavaScript
- TypeScript
Consultant 2: Rainer Stropek wohnt in Linz -> Sprachen: C#+Rust+Go+JavaScript+TypeScript
- C#
- Rust
- Go
- JavaScript
- TypeScript
```

Ausgabe von Listing 7 (Bild 4)

Method	Runtime	_value	Mean	Ratio	Allocated	Alloc Ratio
Serialize	.NET 8.0	Default	38.67 ns	1.00	24 B	1.00
Serialize	.NET 9.0	Default	27.23 ns	0.70	–	0.00
Deserialize	.NET 8.0	Default	73.86 ns	1.00	–	NA
Deserialize	.NET 9.0	Default	70.48 ns	0.95	–	NA
Serialize	.NET 8.0	Instance, NonPublic	37.60 ns	1.00	24 B	1.00
Serialize	.NET 9.0	Instance, NonPublic	26.82 ns	0.71	–	0.00
Deserialize	.NET 8.0	Instance, NonPublic	97.54 ns	1.00	–	NA
Deserialize	.NET 9.0	Instance, NonPublic	70.72 ns	0.73	–	NA

Quelle: Microsoft [2]

Leistungsverbesserungen in System.Text.Json 9.0 (Bild 5)

Zum anderen gibt es auch bei der Klasse *JsonSerializer* eine neue Überladung von *DeserializeAsyncEnumerable()*, bei der man das Flag *topLevelValues* auf *true* setzen kann. Das Listing 7 zeigt die Deserialisierung von Dokumenten mit Listen von Zahlen, Listen von Arrays von Zahlen und Listen von Objekten; die zugehörige Ausgabe ist in Bild 4 zu sehen.

Für das Schreiben von Multi-JSON-Dokumenten gibt es noch keine spezielle Lösung in System.Text.Json. Sie können einfach mehrere Objekte einzeln serialisieren und die entstehenden JSON-Dokumente dann zusammensetzen (zum Beispiel bei *StringBuilder* oder *Stream*).

Leistungsverbesserungen in System.Text.Json 9.0

Microsoft hat die Leistung von System.Text.Json beim Serialisieren und Deserialisieren verbessert. Die Leistungsverbesserungen der Version 9.0 von System.Text.Json sind im Rahmen eines sehr langen Dokuments unter [2] erwähnt, das alle Leistungsverbesserungen in .NET 9.0 auflistet (Bild 5). ■

[1] Microsoft Dev Blogs, Eirik Tsarpalis, What's new in System.Text.Json in .NET 9, www.dotnetpro.de/SL2503DataAccess1

[2] Microsoft Dev Blogs, Stephen Toub, Performance Improvements in .NET 9 / System.Text.Json 9.0, www.dotnetpro.de/SL2503DataAccess2

Dr. Holger Schwichtenberg
alias der „Dotnet-Doktor" ist Leiter des Expertennetzwerks www.IT-Visions.de, das mit 53 renommierten Experten mittlere und große Unternehmen durch Beratungen und Schulungen sowie bei der Softwareentwicklung unterstützt. Durch seine Fachbücher und Vorträge auf Fachkonferenzen gehört er zu den bekanntesten Experten für .NET und Webtechniken in Deutschland.
www.dotnet-doktor.de

dnpCode	A2503DataAccess