How to serialize and deserialize JSON in .NET

09/15/2019 • 22 minutes to read •

In this article

This article shows how to use the System.Text.Json namespace to serialize and deserialize to and from JavaScript Object Notation (JSON).

The directions and sample code use the library directly, not through a framework such as ASP.NET Core.

Most of the serialization sample code sets JsonSerializerOptions.WriteIndented to `true` to "pretty-print" the JSON (with indentation and whitespace for human readability). For production use, you would typically accept the default value of `false` for this setting.

Namespaces

The System.Text.Json namespace contains all the entry points and the main types. The System.Text.Json.Serialization namespace contains attributes and APIs for advanced scenarios and customization specific to serialization and deserialization. The code examples shown in this article require `using` directives for one or both of these namespaces:

C# Copy

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

Attributes from the System.Runtime.Serialization namespace aren't currently supported in `System.Text.Json`.

How to write .NET objects to JSON (serialize)

To write JSON to a string or to a file, call the JsonSerializer.Serialize method.

The following example creates JSON as a string:

C# Copy

```
string jsonString;
jsonString = JsonSerializer.Serialize(weatherForecast);
```

The following example uses synchronous code to create a JSON file:

C# Copy

```
jsonString = JsonSerializer.Serialize(weatherForecast);
File.WriteAllText(fileName, jsonString);
```

The following example uses asynchronous code to create a JSON file:

C# Copy

```
using (FileStream fs = File.Create(fileName))
{
    await JsonSerializer.SerializeAsync(fs, weatherForecast);
}
```

The preceding examples use type inference for the type being serialized. An overload of `Serialize()` takes a generic type parameter:

C# Copy

```
jsonString = JsonSerializer.Serialize<WeatherForecastWithPOCOs>
(weatherForecast);
```

Serialization example

Here's an example class that contains collections and a nested class:

C# Copy

```
public class WeatherForecastWithPOCOs
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
    public string SummaryField;
    public IList<DateTimeOffset> DatesAvailable { get; set; }
    public Dictionary<string, HighLowTemps> TemperatureRanges { get; set; }
    public string[] SummaryWords { get; set; }
}

public class HighLowTemps
{
    public int High { get; set; }
    public int Low { get; set; }
}
```

The JSON output from serializing an instance of the preceding type looks like the following example. The JSON output is minified by default:

JSON Copy

```
{"Date":"2019-08-01T00:00:00-07:00","TemperatureCelsius":25,"Summary":"Hot","
DatesAvailable":
```

```
["2019-08-01T00:00:00-07:00","2019-08-02T00:00:00-07:00"],"TemperatureRanges"
:{"Cold":{"High":20,"Low":-10},"Hot":{"High":60,"Low":20}},"SummaryWords":
["Cool","Windy","Humid"]}
```

The following example shows the same JSON, formatted (that is, pretty-printed with whitespace and indentation):

JSON Copy

```json
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
  "DatesAvailable": [
    "2019-08-01T00:00:00-07:00",
    "2019-08-02T00:00:00-07:00"
  ],
  "TemperatureRanges": {
    "Cold": {
      "High": 20,
      "Low": -10
    },
    "Hot": {
      "High": 60,
      "Low": 20
    }
  },
  "SummaryWords": [
    "Cool",
    "Windy",
    "Humid"
  ]
}
```

Serialize to UTF-8

To serialize to UTF-8, call the JsonSerializer.SerializeToUtf8Bytes method:

C# Copy

```csharp
byte[] jsonUtf8Bytes;
var options = new JsonSerializerOptions
{
    WriteIndented = true
};
jsonUtf8Bytes = JsonSerializer.SerializeToUtf8Bytes(weatherForecast, op-
tions);
```

A Serialize overload that takes a Utf8JsonWriter is also available.

Serializing to UTF-8 is about 5-10% faster than using the string-based methods. The difference is because the bytes (as UTF-8) don't need to be converted to strings (UTF-16).

Serialization behavior

- By default, all public properties are serialized. You can specify properties to exclude.
- The default encoder escapes non-ASCII characters, HTML-sensitive characters within the ASCII-range, and characters that must be escaped according to the RFC 8259 JSON spec.
- By default, JSON is minified. You can pretty-print the JSON.
- By default, casing of JSON names matches the .NET names. You can customize JSON name casing.
- Circular references are detected and exceptions thrown. For more information, see issue 38579 on circular references in the dotnet/corefx repository on GitHub.
- Currently, fields are excluded.

Supported types include:

- .NET primitives that map to JavaScript primitives, such as numeric types, strings, and Boolean.
- User-defined Plain Old CLR Objects (POCOs).
- One-dimensional and jagged arrays (`ArrayName[][]`).
- `Dictionary<string,TValue>` where `TValue` is `object`, `JsonElement`, or a POCO.
- Collections from the following namespaces. For more information, see the issue on collection support in the dotnet/corefx repository on GitHub.
  - System.Collections
  - System.Collections.Generic
  - System.Collections.Immutable

You can implement custom converters to handle additional types or to provide functionality that isn't supported by the built-in converters.

How to read JSON into .NET objects (deserialize)

To deserialize from a string or a file, call the JsonSerializer.Deserialize method.

The following example reads JSON from a string and creates an instance of the `WeatherForecast` class shown earlier for the serialization example:

C# Copy

```
weatherForecast = JsonSerializer.Deserialize<WeatherForecastWithPOCOs>
(jsonString);
```

To deserialize from a file by using synchronous code, read the file into a string, as shown in the following example:

C# Copy

```
jsonString = File.ReadAllText(fileName);
weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(jsonString);
```

To deserialize from a file by using asynchronous code, call the DeserializeAsync method:

C# Copy

```
using (FileStream fs = File.OpenRead(fileName))
{
    weatherForecast = await
JsonSerializer.DeserializeAsync<WeatherForecast>(fs);
}
```

Deserialize from UTF-8

To deserialize from UTF-8, call a JsonSerializer.Deserialize overload that takes a `Utf8JsonReader` or a `ReadOnlySpan<byte>`, as shown in the following examples. The examples assume the JSON is in a byte array named jsonUtf8Bytes.

C# Copy

```
var readOnlySpan = new ReadOnlySpan<byte>(jsonUtf8Bytes);
weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(readOnlySpan);
```

C# Copy

```
var utf8Reader = new Utf8JsonReader(jsonUtf8Bytes);
weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(ref
utf8Reader);
```

Deserialization behavior

- By default, property name matching is case-sensitive. You can specify case-insensitivity.
- If the JSON contains a value for a read-only property, the value is ignored and no exception is thrown.
- Deserialization to reference types without a parameterless constructor isn't supported.
- Deserialization to immutable objects or read-only properties isn't supported. For more information, see GitHub issue 38569 on immutable object support and issue 38163 on read-only property support in the dotnet/corefx repository on GitHub.
- By default, enums are supported as numbers. You can serialize enum names as strings.
- Fields aren't supported.
- By default, comments or trailing commas in the JSON throw exceptions. You can allow comments and trailing commas.
- The default maximum depth is 64.

You can implement custom converters to provide functionality that isn't supported by the built-in converters.

Serialize to formatted JSON

To pretty-print the JSON output, set JsonSerializerOptions.WriteIndented to `true`:

C# Copy

```
var options = new JsonSerializerOptions
{
    WriteIndented = true,
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

Here's an example type to be serialized and pretty-printed JSON output:

C# Copy

```
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

JSON Copy

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot"
}
```

Customize JSON names and values

By default, property names and dictionary keys are unchanged in the JSON output, including case. Enum values are represented as numbers. This section explains how to:

- Customize individual property names
- Convert all property names to camel case
- Implement a custom property naming policy
- Convert dictionary keys to camel case
- Convert enums to strings and camel case

For other scenarios that require special handling of JSON property names and values, you can implement custom converters.

Customize individual property names

To set the name of individual properties, use the [JsonPropertyName] attribute.

Here's an example type to serialize and resulting JSON:

C# Copy

```
public class WeatherForecastWithPropertyNameAttribute
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
    [JsonPropertyName("Wind")]
    public int WindSpeed { get; set; }
}
```

JSON Copy

```json
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
  "Wind": 35
}
```

The property name set by this attribute:

- Applies in both directions, for serialization and deserialization.
- Takes precedence over property naming policies.

Use camel case for all JSON property names

To use camel case for all JSON property names, set JsonSerializerOptions.PropertyNamingPolicy to `JsonNamingPolicy.CamelCase`, as shown in the following example:

C# Copy

```csharp
var serializeOptions = new JsonSerializerOptions
{
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, serializeOptions);
```

Here's an example class to serialize and JSON output:

C# Copy

```csharp
public class WeatherForecastWithPropertyNameAttribute
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
    [JsonPropertyName("Wind")]
    public int WindSpeed { get; set; }
}
```

JSON Copy

```json
{
  "date": "2019-08-01T00:00:00-07:00",
  "temperatureCelsius": 25,
  "summary": "Hot",
  "Wind": 35
}
```

The camel case property naming policy:

- Applies to serialization and deserialization.
- Is overridden by `[JsonPropertyName]` attributes. This is why the JSON property name `Wind` in the example is not camel case.

Use a custom JSON property naming policy

To use a custom JSON property naming policy, create a class that derives from JsonNamingPolicy and override the ConvertName method, as shown in the following example:

C# Copy

```csharp
using System.Text.Json;

namespace SystemTextJsonSamples
{
    class UpperCaseNamingPolicy : JsonNamingPolicy
    {
        public override string ConvertName(string name) =>
            name.ToUpper();
    }
}
```

Then set the JsonSerializerOptions.PropertyNamingPolicy property to an instance of your naming policy class:

C# Copy

```
var options = new JsonSerializerOptions
{
    PropertyNamingPolicy = new UpperCaseNamingPolicy(),
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

Here's an example class to serialize and JSON output:

C# Copy

```
public class WeatherForecastWithPropertyNameAttribute
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
    [JsonPropertyName("Wind")]
    public int WindSpeed { get; set; }
}
```

JSON Copy

```
{
  "DATE": "2019-08-01T00:00:00-07:00",
  "TEMPERATURECELSIUS": 25,
  "SUMMARY": "Hot",
  "Wind": 35
}
```

The JSON property naming policy:

- Applies to serialization and deserialization.
- Is overridden by `[JsonPropertyName]` attributes. This is why the JSON property name `Wind` in the example is not upper case.

Camel case dictionary keys

If a property of an object to be serialized is of type `Dictionary<string,TValue>`, the `string` keys can be converted to camel case. To do that, set DictionaryKeyPolicy to `JsonNamingPolicy.CamelCase`, as shown in the following example:

C# Copy

```
var options = new JsonSerializerOptions
{
    DictionaryKeyPolicy = JsonNamingPolicy.CamelCase,
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

Serializing an object with a dictionary named `TemperatureRanges` that has key-value pairs `"ColdMinTemp", 20` and `"HotMinTemp", 40` would result in JSON output like the following example:

JSON Copy

```json
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
  "TemperatureRanges": {
    "coldMinTemp": 20,
    "hotMinTemp": 40
  }
}
```

The camel case naming policy for dictionary keys applies to serialization only. If you deserialize a dictionary, the keys will match the JSON file even if you specify `JsonNamingPolicy.CamelCase` for the `DictionaryKeyPolicy`.

Enums as strings

By default, enums are serialized as numbers. To serialize enum names as strings, use the JsonStringEnumConverter.

For example, suppose you need to serialize the following class that has an enum:

C# Copy

```csharp
public class WeatherForecastWithEnum
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public Summary Summary { get; set; }
}
```

```
public enum Summary
{
    Cold, Cool, Warm, Hot
}
```

If the Summary is `Hot`, by default the serialized JSON has the numeric value 3:

JSON Copy

```json
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": 3
}
```

The following sample code serializes the enum names instead of the numeric values, and converts the names to camel case:

C# Copy

```csharp
options = new JsonSerializerOptions();
options.Converters.Add(new
JsonStringEnumConverter(JsonNamingPolicy.CamelCase));
options.WriteIndented = true;
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

The resulting JSON looks like the following example:

JSON Copy

```json
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "hot"
}
```

Enum string names can be deserialized as well, as shown in the following example:

C# Copy

```csharp
options = new JsonSerializerOptions();
options.Converters.Add(new
JsonStringEnumConverter(JsonNamingPolicy.CamelCase));
weatherForecast = JsonSerializer.Deserialize<WeatherForecastWithEnum>
```

```
(jsonString, options);
```

Exclude properties from serialization

By default, all public properties are serialized. If you don't want some of them to appear in the JSON output, you have several options. This section explains how to exclude:

- Individual properties
- All read-only properties
- All null-value properties

Exclude individual properties

To ignore individual properties, use the [JsonIgnore] attribute.

Here's an example type to serialize and JSON output:

C# Copy

```csharp
public class WeatherForecastWithIgnoreAttribute
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    [JsonIgnore]
    public string Summary { get; set; }
}
```

JSON Copy

```json
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
}
```

Exclude all read-only properties

A property is read-only if it contains a public getter but not a public setter. To exclude all read-only properties, set the JsonSerializerOptions.IgnoreReadOnlyProperties to `true`, as shown in the following example:

C# Copy

```
var options = new JsonSerializerOptions
{
    IgnoreReadOnlyProperties = true,
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

Here's an example type to serialize and JSON output:

C# Copy

```
public class WeatherForecastWithROProperty
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
    public int WindSpeedReadOnly { get; private set; } = 35;
}
```

JSON Copy

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
}
```

This option applies only to serialization. During deserialization, read-only properties are ignored by default.

Exclude all null value properties

To exclude all null value properties, set the IgnoreNullValues property to `true`, as shown in the following example:

C# Copy

```
var options = new JsonSerializerOptions
{
    IgnoreNullValues = true,
    WriteIndented = true
};
```

```
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

Here's an example object to serialize and JSON output:

Property	Value
Date	8/1/2019 12:00:00 AM -07:00
TemperatureCelsius	25
Summary	null

JSON Copy

```json
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25
}
```

This setting applies to serialization and deserialization. For information about its effect on deserialization, see Ignore null when deserializing.

Customize character encoding

By default, the serializer escapes all non-ASCII characters. That is, it replaces them with `\uxxxx` where `xxxx` is the Unicode code of the character. For example, if the `Summary` property is set to Cyrillic жарко, the `WeatherForecast` object is serialized as shown in this example:

JSON Copy

```json
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "\u0436\u0430\u0440\u043A\u043E"
}
```

Serialize language character sets

To serialize the character set(s) of one or more languages without escaping, specify

Unicode range(s) when creating an instance of System.Text.Encodings.Web.JavaScriptEncoder, as shown in the following example:

C# Copy

```
using System;
using System.Text.Encodings.Web;
using System.Text.Json;
using System.Text.Unicode;
```

C# Copy

```
options = new JsonSerializerOptions
{
    Encoder = JavaScriptEncoder.Create(UnicodeRanges.Cyrillic,
UnicodeRanges.GreekExtended),
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

This code doesn't escape Cyrillic or Greek characters. If the `Summary` property is set to Cyrillic жарко, the `WeatherForecast` object is serialized as shown in this example:

JSON Copy

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "жарко"
}
```

To serialize all language sets without escaping, use UnicodeRanges.All.

Serialize specific characters

An alternative is to specify individual characters that you want to allow through without being escaped. The following example serializes only the first two characters of жарко:

C# Copy

```
using System;
using System.Text.Encodings.Web;
using System.Text.Json;
```

```csharp
using System.Text.Unicode;
```

C# Copy

```csharp
var encoderSettings = new TextEncoderSettings();
encoderSettings.AllowCharacters('\u0436', '\u0430');
options = new JsonSerializerOptions
{
    Encoder = JavaScriptEncoder.Create(encoderSettings),
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

Here's an example of JSON produced by the preceding code:

JSON Copy

```json
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "жа\u0440\u043A\u043E"
}
```

Serialize all characters

To minimize escaping you can use JavaScriptEncoder.UnsafeRelaxedJsonEscaping, as shown in the following example:

C# Copy

```csharp
using System;
using System.Text.Encodings.Web;
using System.Text.Json;
using System.Text.Unicode;
```

C# Copy

```csharp
options = new JsonSerializerOptions
{
    Encoder = JavaScriptEncoder.UnsafeRelaxedJsonEscaping,
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

Caution

Compared to the default encoder, the `UnsafeRelaxedJsonEscaping` encoder is more permissive about allowing characters to pass through unescaped:

- It doesn't escape HTML-sensitive characters such as `<`, `>`, `&`, and `'`.
- It doesn't offer any additional defense-in-depth protections against XSS or information disclosure attacks, such as those which might result from the client and server disagreeing on the *charset*.

Use the unsafe encoder only when it's known that the client will be interpreting the resulting payload as UTF-8 encoded JSON. For example, you can use it if the server is sending the response header `Content-Type: application/json; charset=utf-8`. Never allow the raw `UnsafeRelaxedJsonEscaping` output to be emitted into an HTML page or a `<script>` element.

Serialize properties of derived classes

Polymorphic serialization isn't supported when you specify at compile time the type to be serialized. For example, suppose you have a `WeatherForecast` class and a derived class `WeatherForecastWithWind`:

C# Copy

```csharp
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

C# Copy

```csharp
public class WeatherForecastDerived : WeatherForecast
{
    public int WindSpeed { get; set; }
}
```

And suppose the type argument of the `Serialize` method at compile time is `WeatherForecast`:

C# Copy

```
var serializeOptions = new JsonSerializerOptions
{
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize<WeatherForecast>(weatherForecast,
serializeOptions);
```

In this scenario, the `WindSpeed` property is not serialized even if the `weatherForecast` object is actually a `WeatherForecastWithWind` object. Only the base class properties are serialized:

JSON Copy

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot"
}
```

This behavior is intended to help prevent accidental exposure of data in a derived runtime-created type.

To serialize the properties of the derived type, use one of the following approaches:

- Call an overload of Serialize that lets you specify the type at runtime:

  C# Copy

  ```
  serializeOptions = new JsonSerializerOptions
  {
      WriteIndented = true
  };
  jsonString = JsonSerializer.Serialize(weatherForecast,
  weatherForecast.GetType(), serializeOptions);
  ```

- Declare the object to be serialized as `object`.

  C# Copy

```
serializeOptions = new JsonSerializerOptions
{
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize<object>(weatherForecast,
serializeOptions);
```

In the preceding example scenario, both approaches cause the `WindSpeed` property to be included in the JSON output:

JSON Copy

```json
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
  "WindSpeed": 35
}
```

For information about polymorphic deserialization, see Support polymorphic deserialization.

Allow comments and trailing commas

By default, comments and trailing commas are not allowed in JSON. To allow comments in the JSON, set the JsonSerializerOptions.ReadCommentHandling property to `JsonCommentHandling.Skip`. And to allow trailing commas, set the JsonSerializerOptions.AllowTrailingCommas property to `true`. The following example shows how to allow both:

C# Copy

```csharp
var options = new JsonSerializerOptions
{
    ReadCommentHandling = JsonCommentHandling.Skip,
    AllowTrailingCommas = true,
};
var weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(jsonString,
options);
```

Here's example JSON with comments and a trailing comma:

JSON Copy

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25, // Fahrenheit 77
  "Summary": "Hot", /* Zharko */
}
```

Case-insensitive property matching

By default, deserialization looks for case-sensitive property name matches between JSON and the target object properties. To change that behavior, set JsonSerializerOptions.PropertyNameCaseInsensitive to `true`:

C# Copy

```
var options = new JsonSerializerOptions
{
    PropertyNameCaseInsensitive = true,
};
var weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(jsonString,
options);
```

Here's example JSON with camel case property names. It can be deserialized into the following type that has Pascal case property names.

JSON Copy

```
{
  "date": "2019-08-01T00:00:00-07:00",
  "temperatureCelsius": 25,
  "summary": "Hot",
}
```

C# Copy

```
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

Handle overflow JSON

While deserializing, you might receive data in the JSON that is not represented by properties of the target type. For example, suppose your target type is this:

C# Copy

```csharp
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

And the JSON to be deserialized is this:

JSON Copy

```json
{
  "Date": "2019-08-01T00:00:00-07:00",
  "temperatureCelsius": 25,
  "Summary": "Hot",
  "DatesAvailable": [
    "2019-08-01T00:00:00-07:00",
    "2019-08-02T00:00:00-07:00"
  ],
  "SummaryWords": [
    "Cool",
    "Windy",
    "Humid"
  ]
}
```

If you deserialize the JSON shown into the type shown, the `DatesAvailable` and `SummaryWords` properties have nowhere to go and are lost. To capture extra data such as these properties, apply the JsonExtensionData attribute to a property of type `Dictionary<string,object>` or `Dictionary<string,JsonElement>`:

C# Copy

```csharp
public class WeatherForecastWithExtensionData
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
    [JsonExtensionData]
    public Dictionary<string, object> ExtensionData { get; set; }
}
```

When you deserialize the JSON shown earlier into this sample type, the extra data becomes key-value pairs of the `ExtensionData` property:

Property	Value	Notes
Date	8/1/2019 12:00:00 AM -07:00	
TemperatureCelsius	0	Case-sensitive mismatch (`temperatureCelsius` in the JSON), so the property isn't set.
Summary	Hot	
ExtensionData	temperatureCelsius: 25	Since the case didn't match, this JSON property is an extra and becomes a key-value pair in the dictionary.
	DatesAvailable: 8/1/2019 12:00:00 AM -07:00 8/2/2019 12:00:00 AM -07:00	Extra property from the JSON becomes a key-value pair, with an array as the value object.
	SummaryWords: Cool Windy Humid	Extra property from the JSON becomes a key-value pair, with an array as the value object.

When the target object is serialized, the extension data key value pairs become JSON properties just as they were in the incoming JSON:

JSON Copy

```json
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 0,
  "Summary": "Hot",
  "temperatureCelsius": 25,
  "DatesAvailable": [
    "2019-08-01T00:00:00-07:00",
    "2019-08-02T00:00:00-07:00"
  ],
  "SummaryWords": [
    "Cool",
    "Windy",
    "Humid"
  ]
}
```

Notice that the `ExtensionData` property name doesn't appear in the JSON. This behavior lets the JSON make a round trip without losing any extra data that otherwise wouldn't be deserialized.

Ignore null when deserializing

By default, if a property in JSON is null, the corresponding property in the target object is set to null. In some scenarios, the target property might have a default value, and you don't want a null value to override the default.

For example, suppose the following code represents your target object:

C# Copy

```csharp
public class WeatherForecastWithDefault
{
    public WeatherForecastWithDefault()
    {
        Summary = "No summary";
    }
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

And suppose the following JSON is deserialized:

JSON Copy

```json
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": null
}
```

After deserialization, the `Summary` property of the `WeatherForecastWithDefault` object is null.

To change this behavior, set JsonSerializerOptions.IgnoreNullValues to `true`, as shown in the following example:

C# Copy

```
var options = new JsonSerializerOptions
{
    IgnoreNullValues = true
};
weatherForecast = JsonSerializer.Deserialize<WeatherForecastWithDefault>
(jsonString, options);
```

With this option, the `Summary` property of the `WeatherForecastWithDefault` object is the default value "No summary" after deserialization.

Null values in the JSON are ignored only if they are valid. Null values for non-nullable value types cause exceptions. For more information, see issue 40922 on non-nullable value types in the dotnet/corefx repository on GitHub.

Utf8JsonReader, Utf8JsonWriter, and JsonDocument

System.Text.Json.Utf8JsonReader is a high-performance, low allocation, forward-only reader for UTF-8 encoded JSON text, read from a `ReadOnlySpan<byte>`. The `Utf8JsonReader` is a low-level type that can be used to build custom parsers and deserializers. The JsonSerializer.Deserialize method uses `Utf8JsonReader` under the covers.

System.Text.Json.Utf8JsonWriter is a high-performance way to write UTF-8 encoded JSON text from common .NET types like `String`, `Int32`, and `DateTime`. The writer is a low-level type that can be used to build custom serializers. The JsonSerializer.Serialize method uses `Utf8JsonWriter` under the covers.

System.Text.Json.JsonDocument provides the ability to build a read-only Document

Object Model (DOM) by using `Utf8JsonReader`. The DOM provides random access to data in a JSON payload. The JSON elements that compose the payload can be accessed via the JsonElement type. The `JsonElement` type provides array and object enumerators along with APIs to convert JSON text to common .NET types. `JsonDocument` exposes a RootElement property.

The following sections show how to use these tools for reading and writing JSON.

Use JsonDocument for access to data

The following example shows how to use the JsonDocument class for random access to data in a JSON string:

C# Copy

```
double sum = 0;
int count = 0;

using (JsonDocument document = JsonDocument.Parse(jsonString))
{
    JsonElement root = document.RootElement;
    JsonElement studentsElement = root.GetProperty("Students");
    foreach (JsonElement student in studentsElement.EnumerateArray())
    {
        if (student.TryGetProperty("Grade", out JsonElement gradeElement))
        {
            sum += gradeElement.GetDouble();
        }
        else
        {
            sum += 70;
        }
        count++;
    }
}

double average = sum / count;
Console.WriteLine($"Average grade : {average}");
```

The preceding code:

- Assumes the JSON to analyze is in a string named `jsonString`.
- Calculates an average grade for objects in a `Students` array that have a `Grade`

property.

- Assigns a default grade of 70 for students who don't have a grade.
- Counts students by incrementing a `count` variable with each iteration. An alternative is to call GetArrayLength, as shown in the following example:

C# Copy

```csharp
double sum = 0;
int count = 0;

using (JsonDocument document = JsonDocument.Parse(jsonString))
{
    JsonElement root = document.RootElement;
    JsonElement studentsElement = root.GetProperty("Students");

    count = studentsElement.GetArrayLength();

    foreach (JsonElement student in studentsElement.EnumerateArray())
    {
        if (student.TryGetProperty("Grade", out JsonElement
gradeElement))
        {
            sum += gradeElement.GetDouble();
        }
        else
        {
            sum += 70;
        }
    }
}

double average = sum / count;
Console.WriteLine($"Average grade : {average}");
```

Here's an example of the JSON that this code processes:

JSON Copy

```json
{
  "Class Name": "Science",
  "Teacher\u0027s Name": "Jane",
  "Semester": "2019-01-01",
  "Students": [
    {
      "Name": "John",
      "Grade": 94.3
    },
    {
      "Name": "James",
      "Grade": 81.0
    },
    {
      "Name": "Julia",
      "Grade": 91.9
    },
    {
      "Name": "Jessica",
      "Grade": 72.4
    },
    {
      "Name": "Johnathan"
    }
  ],
  "Final": true
}
```

Use JsonDocument to write JSON

The following example shows how to write JSON from a JsonDocument:

C# Copy

```
string jsonString = File.ReadAllText(inputFileName);

var writerOptions = new JsonWriterOptions
{
    Indented = true
};

var documentOptions = new JsonDocumentOptions
{
    CommentHandling = JsonCommentHandling.Skip
};

using (FileStream fs = File.Create(outputFileName))
using (var writer = new Utf8JsonWriter(fs, options: writerOptions))
using (JsonDocument document = JsonDocument.Parse(jsonString,
documentOptions))
{
    JsonElement root = document.RootElement;

    if (root.ValueKind == JsonValueKind.Object)
    {
        writer.WriteStartObject();
    }
    else
    {
        return;
    }

    foreach (JsonProperty property in root.EnumerateObject())
    {
        property.WriteTo(writer);
    }

    writer.WriteEndObject();

    writer.Flush();
```

The preceding code:

- Reads a JSON file, loads the data into a `JsonDocument`, and writes formatted (pretty-printed) JSON to a file.
- Uses JsonDocumentOptions to specify that comments in the input JSON are allowed but ignored.
- When finished, calls Flush on the writer. An alternative is to let the writer autoflush when it's disposed.

Here's an example of JSON input to be processed by the example code:

JSON Copy

```json
{"Class Name": "Science","Teacher's Name": "Jane","Semester":
"2019-01-01","Students": [{"Name": "John","Grade": 94.3},{"Name":
"James","Grade": 81.0},{"Name": "Julia","Grade": 91.9},{"Name":
"Jessica","Grade": 72.4},{"Name": "Johnathan"}],"Final": true}
```

The result is the following pretty-printed JSON output:

JSON Copy

```json
{
  "Class Name": "Science",
  "Teacher\u0027s Name": "Jane",
  "Semester": "2019-01-01",
  "Students": [
    {
      "Name": "John",
      "Grade": 94.3
    },
    {
      "Name": "James",
      "Grade": 81.0
    },
    {
      "Name": "Julia",
      "Grade": 91.9
    },
    {
      "Name": "Jessica",
      "Grade": 72.4
    },
    {
      "Name": "Johnathan"
    }
  ],
  "Final": true
}
```

Use Utf8JsonWriter

The following example shows how to use the Utf8JsonWriter class:

C# Copy

```
var options = new JsonWriterOptions
{
    Indented = true
};

using (var stream = new MemoryStream())
{
    using (var writer = new Utf8JsonWriter(stream, options))
    {
        writer.WriteStartObject();
        writer.WriteString("date", DateTimeOffset.UtcNow);
        writer.WriteNumber("temp", 42);
        writer.WriteEndObject();
    }

    string json = Encoding.UTF8.GetString(stream.ToArray());
    Console.WriteLine(json);
}
```

Use Utf8JsonReader

The following example shows how to use the Utf8JsonReader class:

C# Copy

```csharp
var options = new JsonReaderOptions
{
    AllowTrailingCommas = true,
    CommentHandling = JsonCommentHandling.Skip
};
Utf8JsonReader reader = new Utf8JsonReader(jsonUtf8Bytes, 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(text);
                break;
            }

        case JsonTokenType.Number:
            {
                int value = reader.GetInt32();
                Console.Write(" ");
                Console.Write(value);
                break;
            }

            // Other token types elided for brevity
    }
    Console.WriteLine();
}
```

The preceding code assumes that the `jsonUtf8` variable is a byte array that contains valid JSON, encoded as UTF-8.

Filter data using Utf8JsonReader

The following example shows how to read a file synchronously and search for a value:

C# Copy

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

namespace SystemTextJsonSamples
{
    class Utf8ReaderFromFile
    {
        private static readonly byte[] s_nameUtf8 =
Encoding.UTF8.GetBytes("name");
        private static readonly byte[] s_universityUtf8 =
Encoding.UTF8.GetBytes("University");

        public static void Run()
        {
            // Read as UTF-16 and transcode to UTF-8 to convert to a
Span<byte>
            string fileName = "Universities.json";
            string jsonString = File.ReadAllText(fileName);
            ReadOnlySpan<byte> jsonReadOnlySpan =
Encoding.UTF8.GetBytes(jsonString);

            // Or ReadAllBytes if the file encoding is UTF-8:
            //string fileName = "UniversitiesUtf8.json";
            //ReadOnlySpan<byte> jsonReadOnlySpan =
File.ReadAllBytes(fileName);

            int count = 0;
            int total = 0;

            var json = new Utf8JsonReader(jsonReadOnlySpan, isFinalBlock:
true, state: default);

            while (json.Read())
            {
                JsonTokenType tokenType = json.TokenType;

                switch (tokenType)
                {
                    case JsonTokenType.StartObject:
                        total++;
                        break;
                    case JsonTokenType.PropertyName:
                        if (json.ValueSpan.SequenceEqual(s_nameUtf8))
                        {
                            // Assume valid JSON, known schema
                            json.Read();
                            if (json.ValueSpan.EndsWith(s_universityUtf8))
                            {
                                count++;
                            }
                        }
                        break;
                }
            }
            Console.WriteLine($"{count} out of {total} have names that end
with 'University'");
        }
    }
}
```

The preceding code:

- Assumes the file is encoded as UTF-16 and transcodes it into UTF-8. A file encoded as UTF-8 can be read directly into a `ReadOnlySpan<byte>`, by using the following code:

  C# Copy

  ```
  ReadOnlySpan<byte> jsonReadOnlySpan = File.ReadAllBytes(fileName);
  ```

- Assumes the JSON contains an array of objects and each object may contain a "name" property of type string.
- Counts objects and `name` property values that end with "University".

Here's a JSON sample that the preceding code can read. The resulting summary message is "2 out of 4 have names that end with 'University'":

JSON Copy

```
[
  {
    "web_pages": [ "https://contoso.edu/" ],
    "alpha_two_code": "US",
    "state-province": null,
    "country": "United States",
    "domains": [ "contoso.edu" ],
    "name": "Contoso Community College"
  },
```

```
  {
    "web_pages": [ "http://fabrikam.edu/" ],
    "alpha_two_code": "US",
    "state-province": null,
    "country": "United States",
    "domains": [ "fabrikam.edu" ],
    "name": "Fabrikam Community College"
  },
  {
    "web_pages": [ "http://www.contosouniversity.edu/" ],
    "alpha_two_code": "US",
    "state-province": null,
    "country": "United States",
    "domains": [ "contosouniversity.edu" ],
    "name": "Contoso University"
  },
  {
    "web_pages": [ "http://www.fabrikamuniversity.edu/" ],
    "alpha_two_code": "US",
    "state-province": null,
    "country": "United States",
    "domains": [ "fabrikamuniversity.edu" ],
    "name": "Fabrikam University"
  }
]
```

Additional resources

- System.Text.Json overview
- System.Text.Json API reference
- Write custom converters for System.Text.Json
- DateTime and DateTimeOffset support in System.Text.Json
- GitHub issues in the dotnet/corefx repository labeled json-functionality-doc

Is this page helpful?

Yes No