JSON and XML Serialization in ASP.NET Web API

05/30/2012 • 9 minutes to read • Contributors 👤👤👤🦖👤 all

In this article

by Mike Wasson

This article describes the JSON and XML formatters in ASP.NET Web API.

In ASP.NET Web API, a *media-type formatter* is an object that can:

- Read CLR objects from an HTTP message body
- Write CLR objects into an HTTP message body

Web API provides media-type formatters for both JSON and XML. The framework inserts these formatters into the pipeline by default. Clients can request either JSON or XML in the Accept header of the HTTP request.

Contents 🔗

- JSON Media-Type Formatter
  - Read-Only Properties
  - Dates
  - Indenting
  - Camel Casing
  - Anonymous and Weakly-Typed Objects

- XML Media-Type Formatter

JSON Media-Type Formatter

JSON formatting is provided by the **JsonMediaTypeFormatter** class. By default, **JsonMediaTypeFormatter** uses the Json.NET library to perform serialization. Json.NET is a third-party open source project.

If you prefer, you can configure the **JsonMediaTypeFormatter** class to use the **DataContractJsonSerializer** instead of Json.NET. To do so, set the **UseDataContractJsonSerializer** property to **true**:

```C#
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.UseDataContractJsonSerializer = true;
```

JSON Serialization

This section describes some specific behaviors of the JSON formatter, using the default Json.NET serializer. This is not meant to be comprehensive documentation of the Json.NET library; for more information, see the Json.NET Documentation.

What Gets Serialized?

By default, all public properties and fields are included in the serialized JSON. To omit a property or field, decorate it with the **JsonIgnore** attribute.

```C#
public class Product
{
```

```csharp
    public string Name { get; set; }
    public decimal Price { get; set; }
    [JsonIgnore]
    public int ProductCode { get; set; } // omitted
}
```

If you prefer an "opt-in" approach, decorate the class with the **DataContract** attribute. If this attribute is present, members are ignored unless they have the **DataMember**. You can also use **DataMember** to serialize private members.

C#   ⧉ Copy

```csharp
[DataContract]
public class Product
{
    [DataMember]
    public string Name { get; set; }
    [DataMember]
    public decimal Price { get; set; }
    public int ProductCode { get; set; }  // omitted by default
}
```

Read-Only Properties

Read-only properties are serialized by default.

Dates

By default, Json.NET writes dates in [ISO 8601](#) format. Dates in UTC (Coordinated Universal Time) are written with a "Z" suffix. Dates in local time include a time-zone offset. For example:

console   ⧉ Copy

```
2012-07-27T18:51:45.53403Z          // UTC
2012-07-27T11:51:45.53403-07:00     // Local
```

By default, Json.NET preserves the time zone. You can override this by setting the DateTimeZoneHandling property:

C#   ⧉ Copy

3/8/2019

JSON and XML Serialization in ASP.NET Web API | Microsoft Docs

```
// Convert all dates to UTC
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.SerializerSettings.DateTimeZoneHandling =
Newtonsoft.Json.DateTimeZoneHandling.Utc;
```

If you prefer to use [Microsoft JSON date format](`"\/Date(ticks)\/"`) instead of ISO 8601, set the **DateFormatHandling** property on the serializer settings:

C#   ⧉ Copy

```
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.SerializerSettings.DateFormatHandling
= Newtonsoft.Json.DateFormatHandling.MicrosoftDateFormat;
```

Indenting

To write indented JSON, set the **Formatting** setting to **Formatting.Indented**:

C#   ⧉ Copy

```
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.SerializerSettings.Formatting = Newtonsoft.Json.Formatting.Indented;
```

Camel Casing

To write JSON property names with camel casing, without changing your data model, set the **CamelCasePropertyNamesContractResolver** on the serializer:

C#   ⧉ Copy

```
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.SerializerSettings.ContractResolver = new
CamelCasePropertyNamesContractResolver();
```

Anonymous and Weakly-Typed Objects

An action method can return an anonymous object and serialize it to JSON. For example:

C#   ⧉ Copy

https://docs.microsoft.com/en-us/aspnet/web-api/overview/formats-and-model-binding/json-and-xml-serialization   4/12

```csharp
public object Get()
{
    return new {
        Name = "Alice",
        Age = 23,
        Pets = new List<string> { "Fido", "Polly", "Spot" }
    };
}
```

The response message body will contain the following JSON:

JSON	⧉ Copy

```json
{"Name":"Alice","Age":23,"Pets":["Fido","Polly","Spot"]}
```

If your web API receives loosely structured JSON objects from clients, you can deserialize the request body to a **Newtonsoft.Json.Linq.JObject** type.

C#	⧉ Copy

```csharp
public void Post(JObject person)
{
    string name = person["Name"].ToString();
    int age = person["Age"].ToObject<int>();
}
```

However, it is usually better to use strongly typed data objects. Then you don't need to parse the data yourself, and you get the benefits of model validation.

The XML serializer does not support anonymous types or **JObject** instances. If you use these features for your JSON data, you should remove the XML formatter from the pipeline, as described later in this article.

XML Media-Type Formatter

XML formatting is provided by the **XmlMediaTypeFormatter** class. By default, **XmlMediaTypeFormatter** uses the **DataContractSerializer** class to perform serialization.

If you prefer, you can configure the **XmlMediaTypeFormatter** to use the **XmlSerializer** instead of the **DataContractSerializer**. To do so, set the **UseXmlSerializer** property to **true**:

C#	⧉ Copy

```
var xml = GlobalConfiguration.Configuration.Formatters.XmlFormatter;
xml.UseXmlSerializer = true;
```

The **XmlSerializer** class supports a narrower set of types than **DataContractSerializer**, but gives more control over the resulting XML. Consider using **XmlSerializer** if you need to match an existing XML schema.

XML Serialization

This section describes some specific behaviors of the XML formatter, using the default **DataContractSerializer**.

By default, the DataContractSerializer behaves as follows:

- All public read/write properties and fields are serialized. To omit a property or field, decorate it with the **IgnoreDataMember** attribute.
- Private and protected members are not serialized.
- Read-only properties are not serialized. (However, the contents of a read-only collection property are serialized.)
- Class and member names are written in the XML exactly as they appear in the class declaration.
- A default XML namespace is used.

If you need more control over the serialization, you can decorate the class with the **DataContract** attribute. When this attribute is present, the class is serialized as follows:

- "Opt in" approach: Properties and fields are not serialized by default. To serialize a property or field, decorate it with the **DataMember** attribute.
- To serialize a private or protected member, decorate it with the **DataMember** attribute.
- Read-only properties are not serialized.
- To change how the class name appears in the XML, set the *Name* parameter in the **DataContract** attribute.
- To change how a member name appears in the XML, set the *Name* parameter in the **DataMember** attribute.
- To change the XML namespace, set the *Namespace* parameter in the **DataContract** class.

Read-Only Properties

Read-only properties are not serialized. If a read-only property has a backing private field, you can mark the private field with the **DataMember** attribute. This approach requires the **DataContract** attribute on the class.

C#   ⧉ Copy

```csharp
[DataContract]
public class Product
{
    [DataMember]
    private int pcode;  // serialized

    // Not serialized (read-only)
    public int ProductCode { get { return pcode; } }
}
```

Dates

Dates are written in ISO 8601 format. For example, "2012-05-23T20:21:37.9116538Z".

Indenting

To write indented XML, set the **Indent** property to **true**:

C#   ⧉ Copy

```csharp
var xml = GlobalConfiguration.Configuration.Formatters.XmlFormatter;
xml.Indent = true;
```

Setting Per-Type XML Serializers

You can set different XML serializers for different CLR types. For example, you might have a particular data object that requires **XmlSerializer** for backward compatibility. You can use **XmlSerializer** for this object and continue to use **DataContractSerializer** for other types.

To set an XML serializer for a particular type, call **SetSerializer**.

C#   ⧉ Copy

```csharp
var xml = GlobalConfiguration.Configuration.Formatters.XmlFormatter;
// Use XmlSerializer for instances of type "Product":
xml.SetSerializer<Product>(new XmlSerializer(typeof(Product)));
```

You can specify an **XmlSerializer** or any object that derives from **XmlObjectSerializer**.

Removing the JSON or XML Formatter

You can remove the JSON formatter or the XML formatter from the list of formatters, if you do not want to use them. The main reasons to do this are:

- To restrict your web API responses to a particular media type. For example, you might decide to support only JSON responses, and remove the XML formatter.
- To replace the default formatter with a custom formatter. For example, you could replace the JSON formatter with your own custom implementation of a JSON formatter.

The following code shows how to remove the default formatters. Call this from your **Application_Start** method, defined in Global.asax.

C#	⧉ Copy

```csharp
void ConfigureApi(HttpConfiguration config)
{
    // Remove the JSON formatter
    config.Formatters.Remove(config.Formatters.JsonFormatter);

    // or

    // Remove the XML formatter
    config.Formatters.Remove(config.Formatters.XmlFormatter);
}
```

Handling Circular Object References

By default, the JSON and XML formatters write all objects as values. If two properties refer to the same object, or if the same object appears twice in a collection, the formatter will serialize the object twice. This is a particular problem if your object graph contains cycles, because the serializer will throw an exception when it detects a loop in the graph.

Consider the following object models and controller.

C#	⧉ Copy

```csharp
public class Employee
{
```

```csharp
    public string Name { get; set; }
    public Department Department { get; set; }
}

public class Department
{
    public string Name { get; set; }
    public Employee Manager { get; set; }
}

public class DepartmentsController : ApiController
{
    public Department Get(int id)
    {
        Department sales = new Department() { Name = "Sales" };
        Employee alice = new Employee() { Name = "Alice", Department = sales };
        sales.Manager = alice;
        return sales;
    }
}
```

Invoking this action will cause the formatter to thrown an exception, which translates to a status code 500 (Internal Server Error) response to the client.

To preserve object references in JSON, add the following code to **Application_Start** method in the Global.asax file:

C#   ⧉ Copy

```csharp
var json = GlobalConfiguration.Configuration.Formatters.JsonFormatter;
json.SerializerSettings.PreserveReferencesHandling =
    Newtonsoft.Json.PreserveReferencesHandling.All;
```

Now the controller action will return JSON that looks like this:

JSON   ⧉ Copy

```json
{"$id":"1","Name":"Sales","Manager":{"$id":"2","Name":"Alice","Department":
{"$ref":"1"}}}
```

Notice that the serializer adds an "$id" property to both objects. Also, it detects that the Employee.Department property creates a loop, so it replaces the value with an object reference: {"$ref":"1"}.

ⓘ **Note**

> Object references are not standard in JSON. Before using this feature, consider
> whether your clients will be able to parse the results. It might be better simply to
> remove cycles from the graph. For example, the link from Employee back to
> Department is not really needed in this example.

To preserve object references in XML, you have two options. The simpler option is to add
`[DataContract(IsReference=true)]` to your model class. The *IsReference* parameter enables
object references. Remember that **DataContract** makes serialization opt-in, so you will also
need to add **DataMember** attributes to the properties:

C#   ⧉ Copy

```csharp
[DataContract(IsReference=true)]
public class Department
{
    [DataMember]
    public string Name { get; set; }
    [DataMember]
    public Employee Manager { get; set; }
}
```

Now the formatter will produce XML similar to following:

XML   ⧉ Copy

```xml
<Department xmlns:i="http://www.w3.org/2001/XMLSchema-instance" z:Id="i1"
            xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/"
            xmlns="http://schemas.datacontract.org/2004/07/Models">
  <Manager>
    <Department z:Ref="i1" />
    <Name>Alice</Name>
  </Manager>
  <Name>Sales</Name>
</Department>
```

If you want to avoid attributes on your model class, there is another option: Create a new
type-specific **DataContractSerializer** instance and set *preserveObjectReferences* to **true** in
the constructor. Then set this instance as a per-type serializer on the XML media-type
formatter. The following code show how to do this:

C#   ⧉ Copy

```csharp
var xml = GlobalConfiguration.Configuration.Formatters.XmlFormatter;
var dcs = new DataContractSerializer(typeof(Department), null, int.MaxValue,
```

```csharp
            false, /* preserveObjectReferences: */ true, null);
    xml.SetSerializer<Department>(dcs);
```

Testing Object Serialization

As you design your web API, it is useful to test how your data objects will be serialized. You can do this without creating a controller or invoking a controller action.

C#   ⧉ Copy

```csharp
string Serialize<T>(MediaTypeFormatter formatter, T value)
{
    // Create a dummy HTTP Content.
    Stream stream = new MemoryStream();
    var content = new StreamContent(stream);
    /// Serialize the object.
    formatter.WriteToStreamAsync(typeof(T), value, stream, content,
null).Wait();
    // Read the serialized string.
    stream.Position = 0;
    return content.ReadAsStringAsync().Result;
}

T Deserialize<T>(MediaTypeFormatter formatter, string str) where T : class
{
    // Write the serialized string to a memory stream.
    Stream stream = new MemoryStream();
    StreamWriter writer = new StreamWriter(stream);
    writer.Write(str);
    writer.Flush();
    stream.Position = 0;
    // Deserialize to an object of type T
    return formatter.ReadFromStreamAsync(typeof(T), stream, null, null).Result
as T;
}

// Example of use
void TestSerialization()
{
    var value = new Person() { Name = "Alice", Age = 23 };

    var xml = new XmlMediaTypeFormatter();
    string str = Serialize(xml, value);

    var json = new JsonMediaTypeFormatter();
    str = Serialize(json, value);

    // Round trip
```

```
        Person person2 = Deserialize<Person>(json, str);
}
```