Create a custom JSON Serialization Binder to resolve Derived Types

By Andrew Hinkle

Serialization and deserialization of derived types is unfortunately not straightforward regardless of the serializer you use. A solution to resolve derived types correctly for [JsonSerializer](https://www.newtonsoft.com/json/help/html/T_Newtonsoft_Json_JsonSerializer.htm) is to use [TypeNameHandling](https://www.newtonsoft.com/json/help/html/T_Newtonsoft_Json_TypeNameHandling.htm) and add a custom JSON serialization Binder that derives from the [DefaultSerializationBinder](https://www.newtonsoft.com/json/help/html/T_Newtonsoft_Json_Serialization_DefaultSerializationBinder.htm) to override the BindToName and BindToType.

The following is some example code from a C# .NET Core WebApi using the JSON serializer. Check out <https://github.com/penblade/Tips/tree/master/Tips.JsonSerializer> for the entire solution. First, let's create the Derived Types of class "Product1" and class "Nested.Product2" that implements abstract class "Product".

# Models.Product

namespace Tips.JsonSerializer.Models

{

public abstract class Product

{

public int Id { get; set; }

}

}

# Models.Product1

namespace Tips.JsonSerializer.Models

{

public class Product1 : Product

{

public string UniqueProperty1 { get; set; }

}

}

# Models.Nested.Product2

namespace Tips.JsonSerializer.Models.Nested

{

public class Product2 : Product

{

public string UniqueProperty2 { get; set; }

}

}

Next, we'll create a "ProductRequest" class that implements the Derived Types, the abstract class, and a List variation of each.

# Models.ProductRequest

using System.Collections.Generic;

using Tips.JsonSerializer.Models.Nested;

namespace Tips.JsonSerializer.Models

{

public class ProductRequest

{

public Product1 A { get; set; }

public Product2 B { get; set; }

public Product C { get; set; }

public Product D { get; set; }

public List<Product1> AList { get; set; }

public List<Product2> BList { get; set; }

public List<Product> CList { get; set; }

public List<Product> DList { get; set; }

}

}

In the ProductsController we'll add the simple Get Action that accepts a ProductRequest.

# Controllers.ProductsController

using Microsoft.AspNetCore.Mvc;

using Tips.JsonSerializer.Models;

namespace Tips.JsonSerializer.Controllers

{

[ApiController]

public class ProductsController : ControllerBase

{

[Route("api/products")]

[HttpGet]

public ActionResult Get(ProductRequest request)

{

// Return the request formatted nicely to quickly see

// if the request deserialized and serialized correctly.

return Ok(request);

}

}

}

With the Derived Types and basic controller setup, let's add a CustomJsonSerializationBinder that implement the Newtonsoft.JSON DefaultSerializationBinder. In this scenario I don't want to enforce the assembly name. I also want to inject into the serialization binder the namespace name up to the folder where the Models are located, so I can just reference the {relative path}\{model name} of the models in the JSON $type property.

# Configuration.CustomJsonSerializationBinder

using System;

using Newtonsoft.Json.Serialization;

namespace Tips.JsonSerializer.Configuration

{

// Inspiration

// https://stackoverflow.com/questions/8039910/how-do-i-omit-the-assembly-name-from-the-type-name-while-serializing-and-deseria

public class CustomJsonSerializationBinder : DefaultSerializationBinder

{

private readonly string \_namespaceToTypes;

public CustomJsonSerializationBinder(string namespaceToTypes)

{

\_namespaceToTypes = namespaceToTypes;

}

public override void BindToName(

Type serializedType, out string assemblyName, out string typeName)

{

assemblyName = null;

typeName = serializedType.FullName.Replace(\_namespaceToTypes, string.Empty).Trim('.');

}

public override Type BindToType(string assemblyName, string typeName)

{

var typeNameWithNamespace = $"{\_namespaceToTypes}.{typeName}";

return Type.GetType(typeNameWithNamespace);

}

}

}

Cool, in the Startup.ConfigureServices we set the JSON SerializationBinder to CustomJsonSerializationBinder. We also set the TypeNameHandling to TypeNameHandling.Auto. This will add the $type property to JSON when serializing and deserializing, but only for Derived Types.

# Startup

…

namespace Tips.JsonSerializer

{

public class Startup

{

…

public void ConfigureServices(IServiceCollection services)

{

var namespaceToTypes = typeof(Product).Namespace;

services

.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1)

.AddJsonOptions(options =>

{

// Indented to make it easier to read during this demo.

options.SerializerSettings.Formatting = Formatting.Indented;

options.SerializerSettings.TypeNameHandling = TypeNameHandling.Auto;

options.SerializerSettings.ContractResolver = new CamelCasePropertyNamesContractResolver();

options.SerializerSettings.Converters.Add(new StringEnumConverter());

options.SerializerSettings.SerializationBinder = new CustomJsonSerializationBinder(namespaceToTypes);

});

// The AddTransient is not necessary for the JsonSerializer.

//services.AddTransient<Product, Product1>();

//services.AddTransient<Product, Product2>();

}

…

}

I've included a unit test that I used to create the example JSON for the integration test using [Postman](https://www.getpostman.com/) ([SoapUI](https://www.soapui.org/) and [Fiddler](https://www.telerik.com/fiddler) work just as well).

# SerializerTests

using System.Collections.Generic;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Newtonsoft.Json;

using Newtonsoft.Json.Converters;

using Newtonsoft.Json.Serialization;

using Tips.JsonSerializer.Configuration;

using Tips.JsonSerializer.Models;

using Tips.JsonSerializer.Models.Nested;

namespace Tips.JsonSerializer.Tests

{

[TestClass]

public class SerializerTests

{

[TestMethod]

public void CreateSerializedDataForPostmanTest()

{

var request = new ProductRequest

{

A = new Product1

{

Id = 1,

UniqueProperty1 = "value 1"

},

B = new Product2

{

Id = 2,

UniqueProperty2 = "value 2"

},

C = new Product1

{

Id = 3,

UniqueProperty1 = "value 3"

},

D = new Product2

{

Id = 4,

UniqueProperty2 = "value 4"

},

AList = new List<Product1>

{

new Product1

{

Id = 1001,

UniqueProperty1 = "value 1001"

},

new Product1

{

Id = 1002,

UniqueProperty1 = "value 1002"

},

new Product1

{

Id = 1003,

UniqueProperty1 = "value 1003"

}

},

BList = new List<Product2>

{

new Product2

{

Id = 2001,

UniqueProperty2 = "value 2001"

},

new Product2

{

Id = 2002,

UniqueProperty2 = "value 2002"

},

new Product2

{

Id = 2003,

UniqueProperty2 = "value 2003"

}

},

CList = new List<Product>

{

new Product1

{

Id = 3001,

UniqueProperty1 = "value 3001"

},

new Product2

{

Id = 3002,

UniqueProperty2 = "value 3002"

},

new Product1

{

Id = 3003,

UniqueProperty1 = "value 3003"

}

},

DList = new List<Product>

{

new Product2

{

Id = 4001,

UniqueProperty2 = "value 4001"

},

new Product1

{

Id = 4002,

UniqueProperty1 = "value 4002"

},

new Product2

{

Id = 4003,

UniqueProperty2 = "value 4003"

}

}

};

var namespaceToTypes = typeof(Product).Namespace;

var settings = new JsonSerializerSettings {

TypeNameHandling = TypeNameHandling.Auto,

Formatting = Formatting.Indented,

ContractResolver = new CamelCasePropertyNamesContractResolver(),

Converters = new List<JsonConverter> { new StringEnumConverter() },

SerializationBinder = new CustomJsonSerializationBinder(namespaceToTypes)

};

var serialized = JsonConvert.SerializeObject(request, settings);

// Create a new request in Postman

// using the url for this app's localhost.

// Copy the text in the serialized variable.

// Paste into Postman Body.

// Change the ContentType to JSON (application/json)

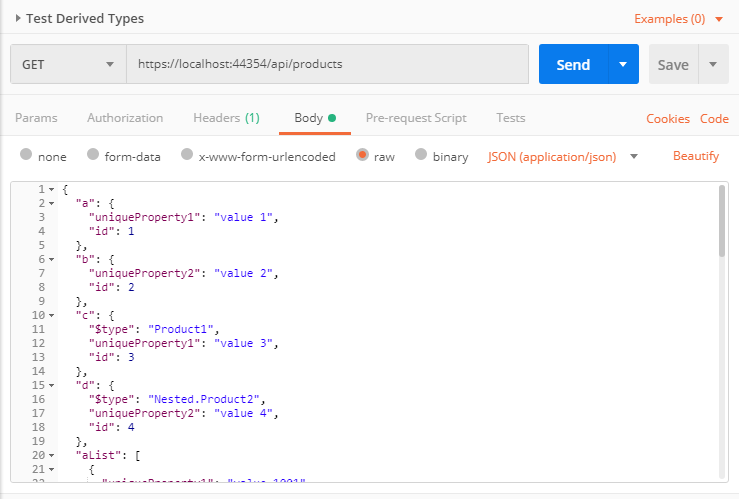
// Click Send

}

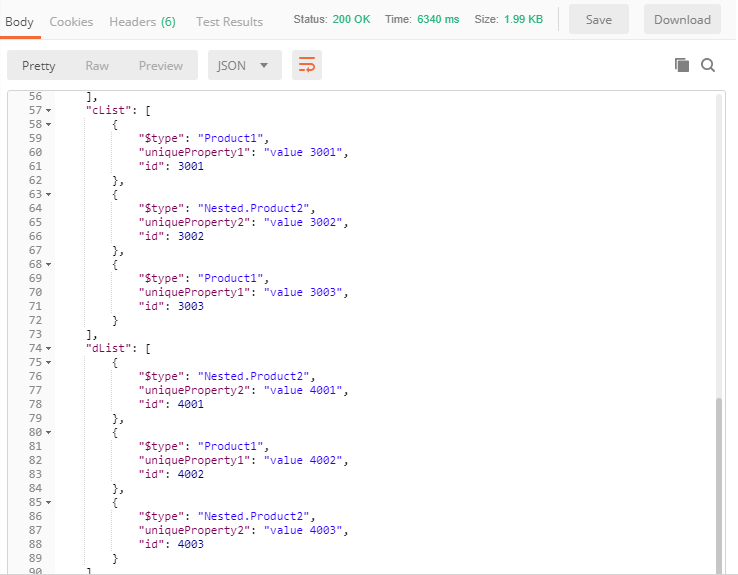
}

}

# Postman (Request) screenshot



# Postman (Response) screenshot



# Postman (Request) JSON

{

"a": {

"uniqueProperty1": "value 1",

"id": 1

},

"b": {

"uniqueProperty2": "value 2",

"id": 2

},

"c": {

"$type": "Product1",

"uniqueProperty1": "value 3",

"id": 3

},

"d": {

"$type": "Nested.Product2",

"uniqueProperty2": "value 4",

"id": 4

},

"aList": [

{

"uniqueProperty1": "value 1001",

"id": 1001

},

{

"uniqueProperty1": "value 1002",

"id": 1002

},

{

"uniqueProperty1": "value 1003",

"id": 1003

}

],

"bList": [

{

"uniqueProperty2": "value 2001",

"id": 2001

},

{

"uniqueProperty2": "value 2002",

"id": 2002

},

{

"uniqueProperty2": "value 2003",

"id": 2003

}

],

"cList": [

{

"$type": "Product1",

"uniqueProperty1": "value 3001",

"id": 3001

},

{

"$type": "Nested.Product2",

"uniqueProperty2": "value 3002",

"id": 3002

},

{

"$type": "Product1",

"uniqueProperty1": "value 3003",

"id": 3003

}

],

"dList": [

{

"$type": "Nested.Product2",

"uniqueProperty2": "value 4001",

"id": 4001

},

{

"$type": "Product1",

"uniqueProperty1": "value 4002",

"id": 4002

},

{

"$type": "Nested.Product2",

"uniqueProperty2": "value 4003",

"id": 4003

}

]

}

# Conclusion

Serialization can be a pain, especially when working with Derived Types. Hopefully, this snippet will give you the inspiration needed to create your own serialization binder or come up with an even better technique. Do you use JsonSerializer or do you prefer XmlSerializer, DataContractSerializer, or rolling your own custom serializer? How do you handle serialization? Do you support Derived Types or have you been working with other workarounds? Do you have another solution to support Derived Types that you like better?