Json .Net for Unity 2.0.1

Developer's Guide (rev 1.0.1 – June 24, 2016)

Based on Json .Net Version: 8.0.3

Table of Contents

Introduction

Overview

Json .Net for Unity is a port of the official Json .Net library. It was created to bring the power of Json .Net to Unity as the official library is not compatible with most Unity platforms and is not compatible with any platform using IL2CPP builds. The aim of this document is to provide clear usage examples and technical explanation for Unity developers and should be fairly easy to understand whether you are a novice or seasoned developer. It does not detail every feature of Json .Net. Comprehensive documentation can be found on the official Json .Net help page.

As the official Json .Net does not support Unity, any questions or problems related to this asset should be sent to us.

Changes From Previous Versions

There are some important changes in Json .Net for Unity 2.0. Currently we are no longer supporting Unity versions prior to Unity 5. This was a design decision made keep the asset clean. Prior versions of Unity did not allow mapping of DLLs to different target platforms. This meant that Json .Net for Unity had to be included in the project as full source code as it relies heavily on compiler directives to compile the correct code for each target platform.

Additionally, this approach made supporting new platforms extremely difficult. Json .Net for Unity was previously based on an old (4.x) version of Json .Net and bug fixes and modifications were patched in over time. This made updating the profile extremely difficult and it also meant that in order to support new platforms, complete new versions of Json .Net had to be ported which was the reason for the WinRT folder (it was actually an entirely different version of Json .Net).

As of version 2.0, Json .Net for Unity is making better use of branching strategies. It started with a clean slate, and a brand new clean clone from the official repository. Bug fixes in the official repository can more easily be migrated to Json .Net for Unity, while Unity specific enhancements can continue to be made.

Source code is still provided in an included .zip archive. This code includes a Visual Studio solution which divides a single code base into multiple projects. This will be discussed in more detail later in this document.

Additionally, we have also removed support for WebPlayer as it is being deprecated by Unity and is no longer supported in most modern browsers and tentatively removed support for Windows Phone 8.1 in favor of Windows 10 UWP.

Limitations

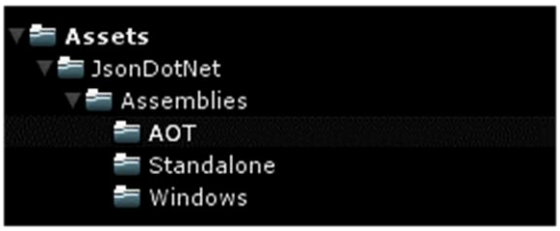
Some built in Unity objects are not directly supported. These include GameObjects and anything that inherits from Monobehavior. There are a few reasons for this. There is limited utility in directly serializing a GameObject because

you can't simply "new" one up. They need to be added to the scene by instiantiating. For this reason you couldn't deserialize one. This is necessary also to maintain the parent / child relationship. Unity also overrides the equality check (== operator) for some objects so they pretend to be null even when they really aren't. These properties (such as rigidbody) have been deprecated, however they are visible via reflection and the serializer will attempt to serialize them which will result in Unity throwing an exception. You can create your own proxy classes to serialize these, however and then repopulate them once they are instantiated.

Installation

As of version 2.0, the structure of Json .Net for Unity has changed. This change provides a cleaner folder structure with less "noise" and make it easier to identify and use. Due to API differences between Unity target platforms, the asset is compiled into multiple DLLs. These DLLs are mapped to various platform targets.

It is always recommended that you delete your existing /JsonDotNet folder before importing a new version. This ensures that you get a clean version. Once the new package is imported you will have the following folder structure:



In addition to the compressed source and assemblies, Json .Net for Unity includes its own link.xml file. This file is used to prevent Unity from stripping built in classes that Json .Net uses. As new stripping issues are discovered, this file will be updated. If you wish to add your own items it is recommended that you do so in your own link.xml file outside of the /JsonDotNet folder to ensure they aren't overwritten when the asset is updated.

Finally, the destination folder for Json .Net is not important. You may move the asset to any folder you desire. Just make sure the link.xml file is retained. If you move the files from outside of Unity you will also need to re-map the DLLs to their appropriate platforms. There is a mapping guide at the end of this document to assist you with mapping the DLLs to the proper target platform.

Consoles and Other Unmapped Platforms

Consoles and most other unmapped platforms are supported out of the box, however they require access to their respective SDKs and specialized Unity builds in order to map the appropriate DLLs for the platform. For all consoles (and nearly all other cases), these platforms should be mapped to the /JsonDotNet/Assemblies/Aot/Newtonsoft.Json.dll file. In order to map this file, select the DLL in the inspector. Find the proper platform(s) in the platform list and select it. Some platforms require a "placeholder" DLL which is used in

the editor and pre-build. If the Placeholder option is present, click the dropdown and choose the ../Assemblies/Standalone/Newtonsoft.Json.dll assembly. Click "Apply" and you should be ready to build.

When in doubt, please see the Platform DLL Mapping Guide at the end of this document. If you are having trouble with a particular platform, contact us and we would be happy to help.

Unsupported Platforms

Json .Net for Unity does not support legacy Unity platforms such as Blackberry or Flash. As of version 2.0 we also no longer support Webplayer, Windows 8.0 or Windows Phone 8.0. Windows 8.1 and 8.1 Universal are supported for the 8.1 SDK. This version currently does not support Windows Phone 8.1 out of the box. It may be possible to create a build for it if necessary.

Working with the Defaults

Modifying Default Serialization Settings

By default, most people will just perform their serialization without specifying a custom instance of JsonSerializationSettings. When this occurs, Json .Net uses a default configuration. You can get this configuration (or even override the function that generates it) and make changes to how Json .Net behaves by default.

The static JsonConvert class contains a method called DefaultSettings. This returns an instance of JsonSerializerSettings. Changing the settings on the class obtained via this method will change the default behavior of Json .Net and will impact all calls to SerializeObject and DeserializeObject that don't supply custom settings. A list of the most common available options and what they do is provided in the Providing Custom Serialization Settings section later in this document.

Built-in Json Converters

Json .Net itself ships with several built-in Converters such as the *StringEnumConverter* which handles serialization and deserialization of enums as strings rather than their integer value and the *IsoDateTimeConverter* which is used to handle dates to and from the Iso date time format. These converters are optional and we discuss their usage in the Serialization and Deserialization using Converters section.

In addition to the official converters, Json .Net for Unity ships with some Unity specific converters. These include a *VectorConverter*, *Matrix4x4Converter*, *ResolutionConverter*, *HashSetConverter* and *ColorConverter* as these are common items. The VectorConverter and HashSetConverter are enabled by default and are already in the collection of converters when you create a new instance of JsonSerializerSettings (more on this later). They may be removed and replaced by your own if you desire. The HashSetConverter is required to make HashSets deserialize correctly on AOT platforms due to a Unity bug.

The built in converters serve a couple of purposes. They streamline the generated json by only serializing data properties. That is, they do not serialize calculated properties. For example, the Vector converter will only serialize the x, y, z and w properties of Vector2, Vector3 and Vector4 respectively. They also improve performance when serializing and deserializing these common types as they are able to quickly generate json or convert the json to the appropriate object without using extra reflection to resolve properties.

Some of the converters also provide additional customization. The VectorConverter, for instance, was built as a single converter to handle Vector2, Vector3 and Vector4 types. You can replace the VectorConverter in the default settings with a new instance or modify the properties on the existing instance. It contains an overloaded constructor which accepts 3 boolean arguments specifying whether it should be used for Vector2, Vector3 and/or Vector4 respectively.

The signature for Vector converter looks like this:

```
public VectorConverter(bool enableVector2, bool enableVector3, bool enableVector4)
```

It also contains matching *EnableVector2*, *EnableVector3* and *EnableVector4* properties. Setting any of these to false would result in the converter not being used for that type. Keep in mind that some types, such as Vectors, require at least some kind of converter as they have calculated properties that also return Vectors leading to an infinite depth serialization and which will result in an exception.

Serializing and Deserializing Data

Basic Serialization and Deserialization

Json .Net for Unity handles most common scenarios out of the box. The heart of this work is performed by the static SerializeObject and DeserializeObject methods of the JsonConvert class. Let's say we have the following enum and class:

```
public enum CharacterType
{
    Oger = 10,
    Human = 20,
    Orc = 30,
    Elf = 40
}

public class PlayerInfo
{
    public string Name { get; set; }
    public int Age { get; set; }
    public CharacterType Race { get; set; }
}
```

We can create a new instance serialize this class by calling JsonConvert.SerializeObject(instance).

```
var mainPlayer = new PlayerInfo()
        {
            Name = "Dustin",
            Age = 36,
            Race = CharacterType.Human
        };

var jsonString = JsonConvert.SerializeObject(mainPlayer);
```

The resulting Json string would be:

```
{
    "Name": "Dustin",
    "Age": 36,
    "Race": 20
}
```

Notice the value of the "Race" property above. Enums are really integers behind the scenes. Enums themselves are a bit of sugar thrown on top of an integer value to give it meaning. When the serializer serializes an enum value, it does so using its integer value. It is possible and very common to use the string value instead. We cover this in the Serialization and Deserialization using Converters section.

You may also control the formatting of the generated json. There are two options available: Indented and None. Using the Indented option producs json that is easily readable and great for debugging.

```
var jsonString = JsonConvert.SerializeObject(mainPlayer, Formatting.None);
```

Deserializing is very similar to serializing, the difference being that DeserializeObject is a generic method for which we supply a type. There are also non-generic overloads that allow you to specify a type parameter and additional settings. We will cover the additional settings in the advanced section.

Let's assume that we have the resulting Json that we serialized earlier.

```
{
    "Name": "Dustin",
    "Age": 36,
    "Race": 20
}
```

Now we want to deserialize this back into the object. The easiest and most common way is to use the generic deserialization method.

```
var newPlayer = JsonConvert.DeserializeObject<PlayerInfo>(jsonString);
```

This will give us a new instance of PlayerInfo with the Name, Age and Race properties populated with the data from our json string. When we make a call to DeserializeObject, Json .Net creates a new instance of PlayerInfo for us and sets the properties from the json. By default, if there are extra properties in the json they are ignored, and if there are properties on PlayerInfo that are not in the json string, they will just be set to their default value (or whatever you initialize them to).

Sometimes you may want to deserialize the json onto an existing object rather than create a new one. This is common in some object pooling scenarios, or where you already have an object in the scene and you just want to load some data and set its properties. This is done using the PopulateObject method. For instance, let's say we have the following PlayerInfo class created:

```
var newPlayer = new PlayerInfo
  {
     Name = "Frank",
     Age = 54,
     Race = CharacterType.Orc
  };
```

Now we want to replace that with the version that we serialized earlier. We can do so with the following code:

```
JsonConvert.PopulateObject(jsonString, newPlayer);
```

Now our newPlayer object will have the properties from our json string instead of the original properties we specified.

Serialization and Deserialization using Converters

Sometimes you may want to perform serialization and deserialization by supplying custom JsonConverters to use rather than adding them to the global settings. Let's say for instance that we want to serialize and deserialize a Resolution object. We can create and maintain an array of JsonConverters and pass them to the SerializeObject and DeserializeObject calls as follows:

```
var res = new Resolution
{
    height = 500,
    width = 400
};

var converter = new ResolutionConverter();
var jsonString = JsonConvert.SerializeObject(res, converter);

var res2 = JsonConvert.DeserializeObject<Resolution>(jsonString, converter);
```

For this overload (and the others that accept converters), the argument supplied is a params array of type JsonConverter. This means that you can pass a single converter, or a JsonConverter array that contains instances of multiple converters you wish to apply.

One important thing to note is that by using this method, only the converters you've supplied will be used. This means that the default converters such as HashSetConverter and VectorConverter will not be enabled. In order to use these you will want to include them in your Converters array as well.

An alternative is to construct a new instance of JsonSerializerSettings instead which will already include the default converters. Then you can simply add your ResolutionConverter to the Converters collection. Here is the example from above but using a JsonSerializerSettings instance instead:

```
var res = new Resolution
{
    height = 500,
    width = 400
};

var settings = new JsonSerializerSettings();
settings.Converters.Add(new ResolutionConverter());

var jsonString = JsonConvert.SerializeObject(res, settings);

var res2 = JsonConvert.DeserializeObject<Resolution>(jsonString, settings);
```

Custom Object Creation with CustomCreationConverter

Sometimes we may need to manually handle the way an object is created. This is helpful if we need to sometimes use an alternate constructor for an object or return a different class type that inherits a base. The CustomCreationConverter<T> is an abstract class that can be inherited to create your own converters. It inherits JsonConverter so can be used just as any standard JsonConverter above. To implement a CustomCreationConverter, you only need to override the Create method which returns type T.

Here is an example of a class with two constructors. The first is the default parameterless constructor, and the second is a constructor that takes a Vector3 location.

```
public class SpawnedItem
{
    public Vector3 SpawnLocation { get; set; }

    public SpawnedItem()
    {
        SpawnLocation = Vector3.zero;
    }

    public SpawnedItem(Vector3 location)
    {
        SpawnLocation = location;
    }
}
```

Given the example above, let us assume that in some cases we want to use the second constructor and default the "location" to a Vector3 at x:0, y:10, z:0. The following CustomCreationConverter would allow us to do that:

```
public class SpawnedItemCreationConverter : CustomCreationConverter<SpawnedItem>
{
    public override SpawnedItem Create(Type objectType)
    {
        return new SpawnedItem(new Vector3(0, 10, 0));
    }
}
```

To use this creation converter, we simply add it to the converters collection as we do with any standard JsonConverter. It's also possible to use via attributes which we will demonstrate in the Using Json Attributes section. Another important thing to note is that the type supplied to the CustomCreationConverter does not have to the the exact type of the item you're creating. The created item could in fact be a sub-type or implementation of the type specified in T.

For example, let's say that SpawnedItem implements an interface called ISpawnable. You can specify your converter as inheriting CustomCreationConverter<ISpawnable> and use it to default the creation of any type that implements that interface. If you've used JsonSerializerSettings to specify TypeNameHandling.Auto, Object or All, then the actual concrete type that's expected will be passed in as the objectType parameter, letting you handle creation of different objects separately.

Another common example of this would be a factory of some kind. Depending on the type supplied in the objectType parameter, you may want to integrate with an object pooling solution to retrieve an object from a pool rather than "new" one up directly.

Customizing Serialization Settings

The default serialization settings can be modified as discussed in the Modifying Default Serialization Settings section, but you can also create and provide your own custom serialization settings. This allows you the flexibility to provide different settings for different serialization and deserialization scenarios by using a new instance of JsonSerializerSettings. In the context of Unity, it is recommended that you create an instance for each scenario you need and cache it to avoid adding allocations to the heap.

Providing Custom Serialization Settings

You can customize the way the serializer works by modifying the default settings or by providing a new instance of JsonSerializerSettings to the serializer. Below are the common properties available on JsonSerializerSettings. Those not mentioned here are more advanced concepts and you can learn more about them on the official Json .Net documentation.

ObjectCreationHandling – Controls how objects are created. The default is Auto which will either create new or update existing objects. If a class initializes a member by default, Json .Net will populate the properties of that existing object rather than replacing it with a new instance if set to Auto. Setting it to Replace will result in objects always being replaced by new instances, and setting it to Reuse will mean that only existing instances are populated and new ones will never be created. In most instances, this should be left to the default value of Auto.

CheckAdditionalContent – By default, Json .Net will throw an exception if it encounters extra json (except for comments) in your json string after it reaches the end of an object. If you'd like the serializer to ignore extra content found at the end of your json string, set this property to false.

ConstructorHandling – Json .Net expects your objects to have a default public parameterless constructor. You can override this behavior for specific objects by using a custom creation converter if you need to use non-standard constructors, but you can also tell Json .Net to look for a private parameterless constructor. By default, ConstructorHandling is set to Default. You can change this setting to AllowNonPublicDefaultConstructor which allows this feature.

Converters – The Converters property holds a collection of JsonConverter objects. These converters are responsible for providing customized serialization and deserialization of certain object types. Whenever you create a new instance of JsonSerializerSettings it will be pre-populated with the default converter types, at which point you can add additional converters and/or remove the defaults. The Creating a Custom JsonConverter section describes how to create your own converter.

Culture – Defines the localization culture that is used when Json is read. By default this is set to CultureInfo.InvariantCulture. If you need to read internationalized Json, you may need to modify this. In most cases, the default won't need to be changed.

DateFormatHandling – This setting defines how dates are serialized and written. By default, dates are written using the IsoDateFormat setting which serialize dates in the Iso format. If necessary, you can change this to MicrosoftDateFormat to use Microsoft date formatting, though in nearly all cases, IsoDateFormat is most appropriate.

DateFormatString – This property allows you to customize the format used for the date, time and timezone offset using a format string. However, beware that the serializer will also then expect dates to be in this format when reading dates and times as well. Refer to the .NET 3.5 documentation for valid format strings.

DateTimeZoneHandling – This setting specifies how dates are treated when they are deserialized. DateTime objects in .NET can have a DateTimeKind and hold timezone information. By default, calls to new DateTime() create a DateTime object with the Kind set to "Unspecified". The DateTimeZoneHandling setting allows you to control how Json .Net handles DateTime creation and conversion from your json data. *Local* means that if datetime strings are UTC formatted, they are converted to a Local datetime. *RoundTripKind* means that time and timezone information are preserved when serialized or deserialized. *Unspecified* means the default .NET DateTime creation handling will be used. *Utc* means the value will be converted to Utc time and specify Utc as the DateTimeKind.

DefaultValueHandling – By default, all public fields and properties on an object are written to the resulting json, even if they are null (or the default value such as 0 for an int). If the properties exist on the class but not in the json string, they are ignored. This setting modifies this behavior as follows: *Ignore* means that properties that are set to their default value (null for objects, false for bools, 0 for ints, etc) are not included in the output json. For cases where you only need the values if they are non-default, this can result in more compact json. *Include* is the default setting and writes members to the resulting json even if they have their default values. *IgnoreAndPopulate* does the same as Ignore when serializing, but when deserializing sets properties to their default values when they are missing from the Json. *Populate* handles serialization the same way as Include, but if members are missing from the json string it sets them to their default values on the object.

Error – This is an event handler that allows you to specify handlers that get executed in the event of an error in serialization or deserialization.

Formatting – Specifies how the resulting Json string is formatted. Indented provides 'pretty' json that is easy to read and debug. 'None' creates compact json with no indents / spacing.

MissingMemberHandling – By default this property is set to Ignore which means that if the Json has a property that doesn't exist on the target object it just gets ignored. Setting this to Error will cause an exception to be thrown if the json contains members that don't exist on the target object type.

NullValueHandling – Options are Include and Ignore. This option specifies whether null values should be included when serializing and deserializing data. The default is Include.

PreserveReferencesHandling – By default this value is set to None. The other options are All, Arrays and Objects. This setting instructs the serializer to keep references in mind when serializing. The resulting Json will include *$ref* properties which identify the objects. These objects will only be deserialized once and references between objects will be restored.

ReferenceLoopHandling – Sometimes the serializer encounters circular or looping references. ReferenceLoopHandling instructs the serializer of what to do in these cases where it may not already be handled by a custom converter. The options are Error, Ignore and Serialize. Error throws an exception with a reference loop is detected. Ignore will ignore the reference loop and not serialize the property. Serialize will serialize the loop. If the loop never ends an error will be thrown. The default setting is Error.

TypeNameHandling – Specifies whether type names are included in the resulting json via a $type property. This is necessary for polymorphic serialization and deserializing objects where an Interface is defined as the target type. The default is None which includes no type information. All includes all type information. Arrays includes type information when serializing Arrays. Objects includes type information when serializing objects. Auto specifies type information when the type being serialized is not the same as the declared type (for example, is a class that inherits the property type or implements it if it's an interface).

Using Json Attributes

JsonConverter Attribute

The JsonConverter attribute is one more commonly used attributes. You can use this attribute to specify converters to be used when you define your classes or properties. For example, if we had a class with a Uri property and we wanted to use the included UriConverter, we could use the JsonConverter attribute to specify it as follows:

```
using Newtonsoft.Json.Converters;

public class GamerProfile
{
    public string GamerTag { get; set; }

    [JsonConverter(typeof(UriConverter))]
    public Uri ProfileLink { get; set; }
}
```

This tells the Serializer and Deserializer to use the UriConverter for the ProfileLink property any time this class is serialized or deserialized. Any other instance of Uri or class with a Uri property would not use this converter unless it was also also being passed in to the serializer or included in the JsonSerializerSettings.Converters collection.

JsonProperty Attribute

The JsonProperty attribute gives us more fine grained control over how a specific property is serialized and deserialized. The two most common uses for JsonProperty are to specify a different name to be used in the json string or if private property getters or setters need to be used. When the JsonProperty attribute is included, the serializer will not only serialize public properties, but will also use internal and private getters and setters.

As an example, let us assume that we have a Player class with a property called Name, but we're fetching the json string from a web service that sends it back as player_name. The json would look like this:

```
{
        "player_name": "Lucius",
        "rank": 23456
}
```

We can "map" the Name property of our player class to the "player_name" moniker by using the JsonProperty attribute. This will cuase the "player_name" value in the Json string to be deserialized to the Name property, and when the Player class is serialized, it will serialize the Name property back as "player_name".

Our Player class would look like this:

```
public class Player
{
    [JsonProperty("player_name")]
    public string Name { get; set; }
    public int Rank { get; set; }
}
```

Additionally, the JsonProperty can be used to set a variety of other options and even to override those on the serializer settings such as TypeNameHandling and DefaultValueHandling. The full reference for the JsonProperty attribute can be found here:
http://www.newtonsoft.com/json/help/html/T_Newtonsoft_Json_JsonPropertyAttribute.htm

JsonIgnore Attribute

The JsonIgnore attribute can be used to ignore properties. In the following example the "Foo" property is ignored entirely but the "Bar" property is serialized.

```
public class Demo
{
    [JsonIgnore]
    public string Foo { get; set; }

    private string Bar { get; set; }
}
```

Additional Attributes

There are many more attributes. As they are not as commonly used they will be added to this document over time. For now, you can see the reference for all of the attributes and types here:
http://www.newtonsoft.com/json/help/html/N_Newtonsoft_Json.htm

Advanced Serialization and Deserialization

Json Without a Matching Class

Sometimes you may want to work with raw json without having a class that matches your json structure. This can be accomplished by using the JObject class which deserializes your data into a key/value collection. For a json array, you can do the same with the JArray class.

The JObject and JArray classes are in the Newtonsoft.Json.Linq namespace. You can use these objects to both serialize and deserialize json. A simple example of using a JObject to create a quick json string is as follows:

```
var jo = new JObject();

jo.Add("PlayerName", "Frank");
jo.Add("Age", 36);

var json = jo.ToString();
```

This method works with primitive values such as int, bool and string because there are implicit converters between them and JToken. JObject inherits JToken, so you can always add a JObject as a property to another JObject, or you can use JToken's FromObject method. For example, if you had a class called PlayerStats and wanted to add it to the above example, it would look like this:

```
var jo = new JObject();

jo.Add("PlayerName", "Frank");
jo.Add("Age", 36);
jo.Add("Stats", JToken.FromObject(stats));

var json = jo.ToString();
```

You can also create a new JObject from a json string. For this example we're going to assume we have the following json string which represents a complex object and contains an array.

```
{
	"PlayerId": 5345,
	"Name": "L33tN00b",
	"Likes": [
		{
			"Type": "Mammal",
			"Name": "Kitten"
		},
		{
			"Type": "Bird",
			"Name": "Eagle"
		}
	]
}
```

It's not important to fully understand the json above for this example, but if you want to know exactly what's going on you can jump ahead to the Appendix B: JSON Structure Primer. For now just know that it prepresents an object with PlayerId, Name and Likes properties. The Likes property is an array of objects each with a Type and Name property. To parse this json string you simply use JObject.Parse:

```
var jo = JObject.Parse(jsonString);
```

Using the JObject we've created above, we can get the properties we want by key name as JObject also functions as a Dictionary<string, JToken>. Since each value is a JToken, there are a number of ways we can get that data back. JTokens have a Value<T> method which allows you to get back types that have implicit converts, and a ToObject<T> method which allows you to deserialize it to a concrete object. You may also use ToString() which will return the json for that specific JToken or, if it's a complex object you may enumerate its properties or directly access its keys.

Given the above statement, let's say we want to access the second "Like" object and find out what its Name is. This can be accomplished as follows:

```
var likeName = jo["Likes"][1]["Name"].Value<string>();
```

Sometimes you may need to deserialize the entire property back to a concrete class. In this case .Value<T> won't cut it as there won't be an implicit conversion between JToken and your type. In this case you'll want to use the ToObject<T> method which will deserialize the JToken to a concrete class. For example, if you were to have a class called LikeItem that has a Type property and a Name property (both of type string), you could deserialize it as per this example:

```
var likeObj = jo["Likes"][1].ToObject<LikeItem>();
```

Serializing With Interfaces or Base / Inherited Classes

Sometimes it is desirable to do serialization and deserialization that involves polymorphism (treating an inherited class as a base class) or serialize and deserialize objects whose properties and defined by interfaces. Let's take the following example. Here we have an IShootable interface and a WeaponBase base class. We also have a Rifle class that inherits WeaponBase and implements IShootable.

```
public interface IShootable
{
    int Ammunition { get; set; }
    void Fire();
}
```

```
public abstract class WeaponBase
{
    public abstract void Equip();
}

public class Rifle : WeaponBase, IShootable
{
    public int Ammunition { get; set; } //IShootable.Ammunition

    public override void Equip() //WeaponBase.Equip
    {
        //Equip the weapon
    }

    public void Fire() //IShootable.Fire
    {
        //Fire the weapon
    }
}
```

Now there are three main ways we can reference this type. The class below has three "weapon" slots, each that references it differently.

```
public class WeaponDemo
{
    public WeaponBase Weapon1 { get; set; }

    public IShootable Weapon2 { get; set; }

    public Rifle Weapon3 { get; set; }
}
```

In the example above, an instance of the Rifle class could be assigned to Weapon1, Weapon2 and Weapon3 because Rifle both inherits WeaponBase and implements IShootable. An instance of the WeaponDemo class would serialize fine, however we would not be able to deserialize it.

This is because the deserializer would not know what the concrete type was by looking at the json. The object you're deserializing to says that it should be WeaponBase and IShootable, however you can't create a new instance of an abstract class or an interface. So how do we work around this?

Json .NET has the ability to store type information along with your json. If you remember from the Customizing Serialization Settings section, we saw a TypeNameHandling property of the JsonSerializerSettings object. In this case setting it to Auto would be most appropriate as it would create an extra variable in your json called $type which would store the concrete type information. It would only do this for Weapon1 and Weapon2 when using Auto because Weapon3 is already of type Rifle so there is no need to store the type.

Important Note: You need to use those settings when *both* serializing and deserializing so it knows to store the $type values and to read them when recreating your objects.

Creating a Custom JsonConverter

It is possible to completely customize the way objects are serialized and deserialized by writing custom converters. These are special classes that inherit JsonConverter. There built in converters we've already talked about are examples of these, such as VectorConverter, HashSetConverter and ColorConverter.

JsonConverters can be complex. In a future update we will included more documentation and examples on creating JsonConverters. For now you can learn more by looking at the official Examples and the API Documentation for JsonConverter.

http://www.newtonsoft.com/json/help/html/CustomJsonConverter.htm

Binary Serialization with BSON

While data serialized to the json format is great for it's simplicity and ease of use, sometimes you may want something that is serialized in binary format. This makes things such as save data harder to tamper with as they are not stored in a plain text format. It also makes the final payloads smaller and serialization and deserialization to and from binary is faster than json.

Json .NET supports a format called "BSON" which is a binary json notation. Serialization and deserialization with bson is a bit different. You'll need to use a BsonWriter which writes the serialized data to a stream (or reader which reads from a stream). That stream can be a file stream, a memory stream, a network request stream etc.

The following example creates a file called PlayerInfo.dat in your persistent data path. It serializes an instance of PlayerInfo and writes it to the file.

```
var pi = new PlayerInfo
{
    Name = "Nemo",
    SkillLevel = 60,
    Health = 74
};

var filePath = Path.Combine(Application.persistentDataPath, "PlayerInfo.dat");
using (var fs = File.Open(filePath, FileMode.Create))
{
    using (var writer = new BsonWriter(fs))
    {
        var serializer = new JsonSerializer();
        serializer.Serialize(writer, pi);
    }
}
```

Here the same serializer is used that is used for Json serialization, however instead of writing the data out to a json string with a JsonWriter, we pass a BsonWriter to the serializer which has a reference to a stream (in this case a FileStream). The BsonWriter writes the binary results to the stream.

To deserialize the contents of the file, we just reverse the process. Here we use a BsonReader to read the contents of the FileStream and then deserialize it back to a PlayerInfo instance.

```
PlayerInfo pi;

using (var fs = File.OpenRead(filePath))
{
    using (var reader = new BsonReader(fs))
    {
        var serializer = new JsonSerializer();
        pi = serializer.Deserialize<PlayerInfo>(reader);
    }
}
```

Reading of bson changes a bit when when the data represents a collection (an array or list for example). The BsonReader expects to read the first binary character as an object, so if you had previously serialized a List<PlayerInfo> or an array (PlayerInfo[]), then you need to tell the reader when deserializing that it's an array.

To tell the BsonReader that you're using an array, you set the ReadRootValueAsArray property of your constructed reader. The following example shows how you would deserialize your PlayerInfo.dat file if it contained a collection of PlayerInfo objects:

```
List<PlayerInfo> players;

using (var fs = File.OpenRead(filePath))
{
    using (var reader = new BsonReader(fs))
    {
        reader.ReadRootValueAsArray = true;
        var serializer = new JsonSerializer();
        players = serializer.Deserialize<List<PlayerInfo>>(reader);
    }
}
```

Appendix A: Platform DLL Mapping

The below table shows which platforms map to each DLL. Over time, Unity also adds additional platforms that are not pre-mapped. In nearly all cases, these platforms should be mapped to the /Assemblies/AOT/Newtonsoft.Json.dll. If in doubt or a platform is not working for you, please send us an email: dustin@parentelement.com.

If you're unfamiliar with assembly mapping, this is a feature that was added in Unity 5. To map an assembly to specific platforms, select the DLL in the project explorer. In the inspector for that file you can specify which platforms and scripting backends you want to use. There are some limitations to the assembly mapping process. For example, it's not possible to map the same DLL to Windows SDK8.1 and UWP without specifying the Any SDK option. This means that I could specify another DLL that only works with SDK 8.0 (or Windows Phone 8).

Editor Standalone	/JsonDotNet/Assemblies/Standalone/Newtonsoft.Json.dll
iOS Android WebGL WSAPlayer (IL2CPP Backend) Tizen SamsungTV tvOS (AppleTV) Xbox360 * XboxOne * WiiU * PS3 * PS4 *	/JsonDotNet/Assemblies/AOT/Newtonsoft.Json.dll
WSAPlayer (.NET Backend) **	/JsonDotNet/Assemblies/Windows/Newtonsoft.Json.dll ***

* Note: Consoles are not currently mapped as I don't have access to the SDK. I can rely on others to test for me, however I can't map the final package before submitting to the asset store. For this reason, if you're developing on a console you will need to map the DLL yourself using the guide above.

** Note: The WSAPlayer DLL was built to target .NET 4.5. Because of this, the mapping for Windows also gives you the option to choose a Placeholder DLL which will be used in the editor. Please use the /JsonDotNet/Assemblies/Standalone/Newtonsoft.Json.dll as the placeholder for WSAPlayer mappings.

*** Note: Even if a placeholder is mapped, the Unity Editor still attempts to read the DLL to check for script upgrades when you first load the package. If you are not developing on Windows 10, you may see an error in the editor indicating that it cannot read the DLL. You can safely ignore this error for now. I will be reporting it as a bug so Unity may fix this in a future version.

Appendix B: JSON Structure Primer

Json is a simple data structure. Property names are surrounded by quotes. Property names and values are separated by a colon. String values are surrounded by quotes. Numeric values are unquoted. Objects are surrounded by curly braces. Arrays are surrounded by square brackets. Commas separate properties and also array elements.

Here are some basic examples:

```
"Lorum Ipsum"
```

A simple string serialized to json.

```
{
  "Name": "Dustin",
  "Age": 36
}
```

An Object with 'Name' and 'Age' properties. 'Name' is a string and 'Age' is a number.

```
[2, 11, 85, 79, 41]
```

An array of numbers.

```
{
  "Name": "Dustin",
  "LuckyNumbers": [2, 11, 85, 79, 41]
}
```

An object with a Name property and LuckyNumbers property. Name is a string and LuckyNumbers is an array of numbers.

```
{
  "Name": "Dustin",
  "Skills": [
            { "Id":1, "Level": 30 },
            { "Id":12, "Level": 8 }
  ]
}
```

An object just like the previous one but in this case Skills is an array of objects.

```
{
  "Name": "Dustin",
  "Attributes": {
            "Age": 36,
            "HeightInches": 69,
            "WeightPounds": 189
  }
}
```

An object with Name and Attributes properties. In this case Attributes is also an Object which would be represented by a separate class. Since all three properties of Attributes are integers, Attributes could also be of type Dictionary<string, int>.

Developer Support Contact

Developer: Dustin Horne
Email: dustin@parentelement.com
Twitter: @dustinhorne
Company Website: http://www.parentelement.com