**INTRODUCTION**

Introducing the first JSON implementation for .Net to implement a fully object-oriented approach to the JSON structure itself. The concept behind this library is building the various JSON constructs in memory before exporting that structure directly to a string, and vice versa. With this approach, serialization to the string form becomes trivial; it’s the structure that’s important.

**HISTORY**

I was introduced to the JSON format while working for a previous employer where we were designing remote systems for flight simulators and the JSON data format was specified by the client. Those programs were written in C and were quite limited as they had to run on microcontrollers.

In order to test these systems, I had to write a Windows-based client with which they could communicate. I looked at some of the solutions out there (Json.Net, etc.) and was unimpressed (read: “confused”) with the way the various JSON constructs were created. Therefore I conceived the architecture below.

**ARCHITECTURE**

Each of the JSON constructs can be mapped to a .Net construct as follows:

JSON object 🡺 .Net Dictionary

JSON array 🡺 .Net List

JSON boolean 🡺 .Net Boolean

JSON numeric 🡺 .Net Double

JSON string 🡺 .Net String

This JSON structure is implemented in .Net through three classes: JsonValue, JsonObject, and JsonArray.

The JsonValue class is a container class for all of the JSON types. In order for everything to remain strongly typed, the class exposes a property for each JSON type. To implement the JSON null value, the static member JsonValue.Null was created.

The JsonObject class is implemented as Dictionary<string, JsonValue>. As such it can be navigated and manipulated as any other dictionary instance would be.

Similarly to the JsonObject class, the JsonArray class is implemented as List<JsonValue>, which can be navigated and manipulated directly as a list instance.

Each of these three classes override the ToString() method to correctly output in a JSON format.

Implicit casts from JsonObject, JsonArray, String, Double, and Boolean were created in JsonValue to simplify coding and readability. Also, comparison operators were overridden for the JsonValue class.

**USAGE**

JSON constructs can be created directly through the use of the cast operators:

JsonValue jsonBool = false;

JsonValue jsonNum = 42;

JsonValue jsonString = “aString”;

JsonValue jsonObject = new JsonObject{{“aKey”, 42}};

JsonValue jsonArray = new JsonArray{4, true, “aValue”};

The above code creates five JsonValue instances. To access these values, use these properties:

jsonBool.Boolean

jsonNum.Numbe

jsonString.String

jsonObject.Object

jsonArray.Array

If a get accessor is used that does not correspond with the JsonValue’s type, an exception is thrown. The default constructor for JsonValue creates a Null value. The JsonValue’s set accessors automatically change the type of the JsonValue, if required. Thus, the following changes the type to Number and sets its value to 42:

jsonBool.Number = 42;

These declarations can be combined in the same way as when declaring any other object. For example, a moderately complex JsonObject can be built as follows:

var json = new JsonObject

{

{“boolean”, true},

{“number”, 42},

{“string”, “a string”},

{“null”, JsonValue.Null},

{“array”, new JsonArray {6.7, true, “a value”}},

{“object”, new JsonObject {{“aKey”, 42}, {“anotherKey”, false}}}

};

The object’s structure, and ultimately its output, is quite apparent directly from the code that created it.

Since JsonObject and JsonArray are implemented as strongly typed collections, all of the underlying operations (e.g. Add(), AddRange(), etc.) are accessible, including LINQ. As such, the following statements are valid:

json.Add(“newItem”, “a new string”);

var onlyStrings =   
json.Select(jkv => jkv.Value.Type == JsonValueType.String).ToJson();

Here, the ToJson() method is an extension method on the IEnumerable<KeyValuePair<string, JsonValue>> type returned by the LINQ Select() method. It returns a JsonObject. There is also a correlating ToJson() method for the IEnumerable<JsonValue> which returns a JsonArray.

As you can see, creating these constructs is quite easy. As is expected, calling json.ToString() yields:

{“boolean”:True,“number”:42,“string”:“a string”,“null”:Null, “array”:[6.7,True,“a value”],“object”,{“aKey”:42,“anotherKey”:False}}

Furthermore, feeding this output back into the JsonObject constructor yields the original structure (although using new instances).

**SERIALIZATION**

The primary object of every other JSON library that I have found is serialization; that is, converting a .Net object into a JSON-formatted .Net string. The problem with existing implementations is that they gloss over the part about building the JSON structure, which hides the actual structure from the library consumer. This implementation neatly defines and exposes the underlying structure to the consumer. The main benefit of this approach is that the consumer can browse and even edit the JSON structure before serializing it to a string or deserializing it into an explicit object.

The process of serializing an object checks for certain cases. These cases are listed in order of descending priority.

1. Does the object implement IJsonCompatible?
   1. By implementing IJsonCompatible, an object expresses that it has a preferred format for JSON serialization. The serializer respects this preference.
2. Is the object a (JSON) primitive type?
   1. Strings, numeric values, and booleans are considered primitive types for JSON. Serialization of other types requires more information.
   2. Enumerations are serialized to their numeric value, and so are included here.
3. Does a pair of converter methods exist for the object’s type?
   1. For types that do not implement IJsonCompatible, are not primitive, and will not be auto-serialized (see next point) as desired, a public registry exists that manages pairs of methods which convert those types to and from the JSON structure.
4. Attempt to auto-serialize.
   1. If none of the conditions above are met, the serializer will attempt to de/serialize the object based on a best-guess.
   2. If this fails, JsonValue.Null is returned for serialization, and the type’s default value is returned for deserialization.

Furthermore, to reduce the size of the serialized data, any object whose value is the default for its type will be serialized to JsonValue.Null. The serializer will omit these values from the output structure. (I plan to implement a JsonSerializerOptions object that will allow configuration for this sort of behavior.)

**IJsonCompatible**

This interface has been created to allow objects to set their own serialization requirements. In this way, non-public members may be serialized and deserialized. These requirements will be respected when the object is serialized as a member of another object. The interface exposes two methods:

* JsonValue ToJson()
  + Returns a JSON representation of the object.
* void FromJson(JsonValue json)
  + Given a JSON representation, this method assigns values to the pertinent data members.

At minimum, the FromJson() method should correctly deserialize the output from the ToJson() method. The serializer cannot prevent default values from appearing in the JSON structure when these methods are used.

**JsonSerializationTypeRegistry**

This class has been implemented as a work-around for those objects which will not be auto-serialized properly or as desired. Many explicit types, such as DateTime, and generic types, such as Nullable<T>, are examples of these.

Some common explicit types are automatically registered. These include:

* System.DateTime
* System.TimeSpan
* (more to come)

To register explicit types such as these, first two methods must be implemented: one that converts to a JsonValue, and one that converts back from a JsonValue to an instance of the type. Then a call is made to the RegisterType<T>() method. For example, registering the System.Drawing.Point object could be implemented as follows:

public static class Program()

{

JsonSerializer \_serializer = new JsonSerializer();

public void Main()

{

JsonSerializationTypeRegistry.RegisterType(PointToJson, JsonToPoint);

var json = \_serializer.Serialize(new Point(5, 6));

}

static JsonValue PointToJson(Point p)

{

return new JsonObject{{“x”, p.x},{“y”, p.y}};

}

static Point JsonToPoint(JsonValue p)

{

return new Point(json.Object[“x”].Number, json.Object[“y”].Number);

}

}

At any time, the serialization methods for a type can be changed by calling the RegisterType<T>() method with a new method pair. This means, for example, that if the default encoding for DateTime is not preferred for the application, the behavior can be overridden by registering new methods. In this way, serialization is completely customizable.

While no generic types are automatically registered, several methods have been exposed to easily register a generic type. These include:

* RegisterNullableType<T>()
* RegisterListType<T>()
* RegisterDictionaryType<TKey, TValue>()
* RegisterQueueType<T>()
* RegisterStackType<T>()
* (more to come)

To register an explicitly-typed generic class, one of these methods should be used. (How to register a generic type not listed here will be covered later.) For example, the following registers the List<int> type:

public static class Program()

{

JsonSerializer \_serializer = new JsonSerializer();

public void Main()

{

JsonSerializationTypeRegistry.RegisterListType<int>();

var json = \_serializer.Serialize(new List<int>{1, 2, 3, 4});

}

}

The methods for serializing the object (List<int> in this case) will be automatically generated.

It should be noted that the default behavior for serializing a Dictionary<TKey, TValue> yields a JsonArray of JsonObjects, each containing “Key” and “Value” items. This is done because the JSON object structure only supports strings as keys, and TKey may not be representable as a string.

For generic types that are not listed above, simply call RegisterType<T>() on the explicitly-defined type. For instance, if an generic class MyGenericClass<T> exists, and a you need to register a MyGenericClass<int>, create the required conversion methods and call RegisterType<MyGenericClass<int>>().

To unregister a type, simply call RegisterType<T>() and pass nulls for both methods. If only one method is null, an exception is thrown. This works for both explicit types and explicitly-defined generic types.

* RegisterType<int>(null, null);
* RegisterType<List<int>>(null, null);

This approach can be likened to implementing IJsonCompatible on third-party types. The ideal solution would be to create one or more static classes which contain these conversion methods, then make a single call to a static method to register them all. Once registered, they remain in memory throughout the life of the application or until they are unregistered.

**Auto-serialization**

The serializer can automatically serialize most objects. There are some notes to consider, however:

* All of the properties to serialize must be implemented with public getters and setters,
  + For instance, the DateTime object cannot be automatically serialized because it exposes no public properties that have both public getters and public setters.
* Properties of interface or abstract class types can be serialized, but the value’s type will be explicitly listed in the serialization.
  + Serialization of properties of these types will result in a JsonObject with two keys:
    - “#Type” – Used to indicate the serialized value’s assembly-qualified type name.
    - “#Value” – Used to hold the serialized object.
* Any property marked with the JsonIgnore attribute will not be serialized.

Any type that does not implement IJsonCompatible and cannot be properly auto-serialized will need to be registered, or else it will be ignored by the serializer.

**Type Serialization**

In addition to serializing objects, the serializer can also serialize the public static properties of types. To do this, simply call the SerializeType<T>() method with the type to be serialized.

No type information is encoded in the JSON structure for type serializations. It remains the responsibility of the consumer to track the various JSON structures that are created to know which types they represent.

**Deserialization**

To deserialize an object, call the Deserialize<T>() method with the appropriate type argument and pass in the JSON representation of the object. The deserializer will parse out which properties are set.

To deserialize a type, call the DeserializeType<T>() method with the appropriate type argument.

If the JSON structure contains keys that are not properties of the object or type, the associated values are ignored. (This also is a behavior that will be configurable with the upcoming JsonSerializerOptions object.) Any properties which are not explicitly set in the JSON structure will remain the default value for the type of that property.

**ONE LAST THING**

I bet some of you are wondering why my base namespace is Manatee. I feel sorry for the creatures; they look so forlorn. I therefore dedicate this JSON library to the Disappointed Manatee.