JSON-ND Format Specification V1.0

JSON-ND or ***JSON with Named Datatypes*** is a very simple extension to the JSON format.

The specification describes a way to define ***specific data-types*** for JSON elements and also allows for ***complex-data types to be defined*** as interfaces. This is far simpler, (but also less complete) than the JSON Schema approach because

- the data and definition are not separated
- and no prior knowledge is required by a service to use the definition.

JSON-ND encoding is always valid JSON and all JSON is valid JSON-ND.

Inspired in part by TypeScript notation, all that is required is that (optionally) ***the data-type be appended to the end of a JSON string***.

Using JSON-ND

To encode data using JSON ND when using Name/Value pairs, just add a colon character (":") and the data-type after **Element Name** within the JSON string.

{"Name:*data-type*" : value}

For **Value elements** contained within a JSON Array, a JSON string is used. The data element is encoded as a JSON string with any literal colons escaped (as per the JSON specification using UNICODE character \u003A), followed by the colon character (":") and the data-type.

["Value:*data-type*"]

The term **"data-type"** is completely open and it should be interpreted to mean:

Text that describes how the associated value should be interpreted by the intended consumer.

A data-type MAY BE a literal value.

It is expected that *more specific* definitions for specific purposes (eg "JSON-ND for C Languages") will be defined in the future as extensions of this specification.

Use of Colons in Data-types

Implementations of JSON-ND should only consider the first occurrence of the colon (":") character. Subsequent colons MAY or MAY NOT be escaped as desired and MUST BE considered as data, and NOT a delimiter.

The purpose of interpreting subsequent colons as data is to facilitate data-types that may include the colon character for its own purpose. For example C++ uses as double colon ("::") as a namespace qualifier. This approach also allows the data-type to describe a method, its parameters and return type for the purposes of RPC calls.

While using characters such as colons, braces and brackets in the name is valid JSON, the practice makes the object name inaccessible after a JSON.parse() call in Javascript. This is fortunate and convenient as it forces the script to validate the received object before any qualified values are (easily) accessible to javascript code. A very basic validator implementation is included as an example below. demo

Data-type structure and style

This specification **DOES NOT** define any specific data types except for the **Interface** data-type. Any programming language style can be used to specify the data type.

In order to ensure the parsing system knows how to deal with the data-types, the following qualifying element **CAN BE** prefixed to the JSON message:

"JsonND" :{"version": 1.0,"style": "***language***"}

where **language** refers to a specific language or specification used within the JSON document.

The ***default language style*** is defined as "***Typescript***", and when no style is specified, TypeScript MUST BE assumed. Again the exact specification for "JSON-ND for Typescript" is out of scope and may be defined elsewhere.

There are strong arguments for selecting **TypeScript** to be the default language. However the limitation of this is that **Number** data type is often the reason the type qualification is required.

Defining Complex Types - the *Interface* data-type

A complex type can be defined using an array of values or objects. The **Interface** data-type reserved for this purpose. It is defined in the following way:

{ "data-type name: **interface**" : "Name:data-type"}

OR

{"data-type name: **interface**" : [

"elementName1:data-type" **OR** {"elementName1:data-type":"value" **OR** [] },

"elementName2:data-type" **OR** {"elementName2:data-type":"value" **OR** [] },

...

"elementName'n':data-type" **OR** {"elementName'n':data-type":"value" **OR** [] }

] }

The Element Name MAY BE empty.

JSON-ND Mime Type

The following MIME Type (to be used in html headers) is defined to be ***application/json+nd*** and the preferred

file extension is ***.jsonnd***.

The use of this mime type is preferred however, as JSON-ND is always valid JSON, the standard JSON mime type (*application/json*) and the *.json* file extension are acceptable.

The HTTP *content-type* and *accept* Headers

For HTTP protocol responses, as per rfc1341, the mime type to use as the JSON-ND mimetype as defined above

```
application/json+nd
```

Consideration should also be given to using the ***Parameter*** option with the "style" attribute to define the language style.

Request Headers: `accept: application/json+nd[; style=`***language***`]`

Response Header: `content-type: application/json+nd[; style=`***language***`]`

JSON-ND Examples:

Element Names

```
{ "useFactor" : 1 }
```

Becomes:

```
{ "useFactor:boolean" : 1 }
```

Value Elements in JSON Array

```
[
"T520",
"H555",
"To be: or not to be",
1,
true
]
```

Becomes:

```
[
"T520:char[4]",
"T520:char[4]",
"To be\u003A or not to be:string",
"1:single",
"true:boolean"
]
```

When intent is not clear use data types, otherwise data type is optional

```
[
"id" : 0,
"Name" : string;
"isLoggedIn:boolean" : 0
]
```

Programming Language Styles (suggestion only)

Typescript style:

```
  "count:BigInteger" : 1,
  "age:number" : 27.3,
  "arrivalTime:Date" : "15:23:02",
  "dollarAmount:number" : 200.33,
  "arrayOfInt:number[]" : [1,2,3,4,5,6]
  "twoDArray:any[2][2]" : [ ["0,0","0,1"], ["1,0","1,1"] ]
}
```

C++ style:

```
{
  ["JsonND" :{"version": 1.0,"style": "C++"},
  "count:short int" : 1,
  "age:float" : 27.3,
  "arrivalTime:std::tm" : "15:23:02",
  "dollarAmount:long double" : 200.33,
  "arrayOfInt:int[]" : [1,2,3,4,5,6]
  "twoDArray:string[2][2]" : [ ["0,0","0,1"], ["1,0","1,1"] ]
}
```

C# style:

```
{
  ["JsonND" :{"version": 1.0,"style": "C#"},
  "count:int" : 1,
  "age:Single;f" : 27.3,
  "arrivalTime:DateTime" : "15:23:02",
  "dollarAmount:Decimal;m" : 200.33,
  "arrayOfInt:int[]" : [1,2,3,4,5,6]
  "twoDArray:string[2,2]" : [ ["0,0","0,1"], ["1,0","1,1"] ]
}
```

Pascal style:

```
{
  ["JsonND" :{"version": 1.0,"style": "C++"},
  "count:integer" : 1,
  "age:single" : 27.3,
  "arrivalTime:datetime" : "15:23:02",
  "dollarAmount:currency" : 200.33,
  "arrayOfInt:array of integer" : [1,2,3,4,5,6]
  "twoDArray:array[0..1][0..1] of string" : [ ["0,0","0,1"], ["1,0","1,1"] ]
}
```

Methods

Given a method **GetUser** with parameters "id" (integer) and "options" (UserEnum) which returns a "User" object, the data COULD BE encoded as follows:

```
{ "GetUser:function(int id, UserEnum options):User" : "https://userserver/api/getuser" }
```

Remember that this specification is not specific about the programming language style. The example above is in a C style, but could also be encoded in a pascal style

```
{ "GetUser:function(Id:integer;Options:UserEnum):User" : "https://userserver/api/getuser" }
```

Interface Examples

Single Value (type aliasing)

C Style

```
{"PInteger:Interface": ":*int"}
```

Pascal Style

```
{"PInteger:Interface": ":^integer"}
```

```
{"HResult:Interface": ":Int32"}
```

Enumerated types.

```
{
  "RoleType:enum" :[
    "admin:1",
    "accounts",
    "sales",
    "service"
  ]
}
```

Simple Objects

```
{
"User:Interface": [
"name:string",
"id:int",
"roles:RoleTypes[0,]"
]
}
```

Class Data Types

If follows that a class type COULD BE defined using the interface above.

The approach for defining data classes is not in the scope of this document. The recommended approach for defining classes using JSON-ND MAY BE described elsewhere. The example below gives the concept some

consideration, but should not be taken as a part of the specification.

A class might include:

- Private, Public, Protected and Published Members
- Methods (and Class Methods)

First define a Class interface:

```
{ "Class:Interface"
  [
  "private:any[0,]",
  "protected:any[0,]",
  "public:any[0,]",
  "published:any[0,]"
]
}
```

Now define the User Class (using the User type and GetUser method type examples)

```
{
  "UserClass:Class" : [
    "private" : [
              "_user:User",
              "getUser:GetUser"
    ],
    "public" : [
              {"user:User as Property": ["get:getUser","set:null"],
               "HasRole:function(role:RoleType):boolean"
    ]
  ]
}
```

Javascript JSON-ND Validation

Below is a very simple JSON-ND parser/validator. [demo]

```
<html>
<head>
<script>
  function JSON_ND_Parse(ndObj)
  {
    var obj = JSON.parse(ndObj);
    for (var propertyName in obj) {
        var p = propertyName.indexOf(":");
        var v = obj[propertyName];
        if (p > 0) {
            if (propertyName.substr(p+1,3)==='int' && !parseInt(v))
            {
                v=NaN;
            }; // add more validation.
            obj[propertyName.substr(0, p)] = v;
            delete(obj[propertyName]);
        }
    };
    return obj;
  }
</script>
</head>
<body>
```

```
  <script>
    var nd = '{"abc:string":"abc contains a string", "def:integer" : "not a number"}';
    document.write("<h3>Original 'nd' object:<pre>",nd,"<pre></h3>");
    document.write("See that 'nd.abc' has no value: abc=<b>" ,nd.abc,"</b>");
    document.write("<h3>Then validate with JSON_ND_Parse:<pre>",JSON.stringify(JSON_ND_Parse(
    document.write("<p style='color: red'>Note that 'def' failed validation</p>");
  </script>
</body>
</html>
```