Design considerations for a schema language for JavaScript Object Notation

S. Jeylatha
Department of Computer Science and Engineering
BITS – PILANI
Dubai, U.A.E
jeylatha@yahoo.com

Munawwar Firoz
Department of Computer Science and Engineering
BITS – PILANI
Dubai, U.A.E
munawwarfiroz@hotmail.com

***Abstract*—This paper deals with a schema format for JavaScript Object Notation based on the current IETF JavaScript object notation schema specification. JavaScript object notation is a text-based data interchange format. The data transferred between two users or modules may have an application specific structure. The aim of this paper is to help application developers ease their task of data validation, documentation and to provide other structural information about data, by defining the structure of the data through the proposed JavaScript object notation schema.**

Keywords—*JavaScript Object Notation, JSON, Text-based data interchange, JSON Schema.*

I. Introduction

JavaScript Object Notation (JSON) [2][3] is a text-based data interchange format that is a well known alternative to XML. Its popularity is mainly due to its readability and simplicity in quickly implementing a general script to exchange data between a server and client. JSON isn't specific to the JavaScript language and can be used with any programming language. There are many implementations that aid in encoding and decoding JSON data through various programming languages, like the Yahoo! User Interface JSON utility and Dojo Toolkit for JavaScript, and the built-in JSON library for PHP.

JSON is a relatively new format as compared to XML, and does not enjoy all the technologies that XML has got, like XML schema. However various efforts are being made to make such comparable technologies, and in fact are already being used by many, even though these efforts are not standardized (currently). This paper is aimed at bringing a modified JSON schema representation based on such an existing work.

This paper is organized into six sections followed by references. Section II briefly states the objectives and motivation behind the paper. Section III A lists all acronyms used in this paper. Section III B and the rest describe the details of the schema specification.

II. Objectives and Motivation

The current IETF schema specification [1] has brought an effort to bring a schema format for JSON. However, there is a strong requirement to extend the JSON schema with a named conceptual entity defining the structure of the data represented and organized into "schema classes". This way defining a schema would have conceptual similarities to defining a context free grammar. This also makes the schema easily definable and brings a quick learning curve. Another requirement is that there needs to be a mechanism for defining ordered arrays.

With an objective to incorporate these needs, this paper aims to specify a modified schema format for JSON based on the current IETF schema specification [1].

III. JavaScript Object Notation Schemas and Schema Class Design

The schema specification mentioned here is a modified version of the JavaScript Object Notation Schema working draft at IETF [1]. The specification is sub-divided into two broad categories, namely core schema and hyper schema. In the most general terms, core schema aims at data validation and hyper schema provides additional information about the data (meta-data) and relations between data.

A. Abbreviations and Acronyms

CSS	: Cascaded Style Sheets.
HTTP	: Hypertext Transfer Protocol.
IETF	: Internet Engineering Task Force.
ISO	: International standard Organization.
JSON	: JavaScript Object Notation.
MIME	: Multipurpose Internet Mail Extensions.
PHP	: Hypertext Preprocessing.
URI	: Uniform Resource Identifier.
XML	: eXtensible Markup Language.

B. Terminology

Primitive data types such as object, array, boolean, number and integer have the same definition as defined in the JSON schema specification [1].

Schema document: A schema definition contained in a single file. Like any other JSON object, it begins with a '{' and ends with '}'.

Instance: JSON data consists of name/value pairs. The phrase 'instance value' denotes a value that the schema describes. The phrase 'instance name' denotes the name (part) of the pair. "Instance" can also be described for a particular type. For example, "Instance object" would mean that the value is described to be of an *object* data type.

Attribute: JSON schema consists of name/value pairs. The phrase 'attribute value' denotes the value (part) of the pair and 'attribute name' denotes the name (part) of the pair.

Schema class: A named object (also known as the 'ID' of the class), containing attributes that defines the structure of the data entity.

C. General structure of a JSON schema definition

A JSON schema document must contain schema class definitions and a way to answer the question "Which part of the JSON data follows which schema class?". Hence the schema document body contains an attribute named "classes" (which is an object that contains schema class definitions) and an attribute named "apply" (which is an array of object, with each object indicating the class references to portions of the JSON data).

The "to" attribute references a location in the JSON data and can either use dot notation (myProducts.product1) or square bracket notation (myProducts["product1"]). "class" references a class name/ID.

The general structure of a schema document is shown in Figure 1. At the top is the JSON data and the bottom is schema definition for the data. With this schema it is understood that myProducts.product1 (from the JSON data) must follow the schema class named "Product". This has been specified in the "apply" attribute.

```
For the following JSON data...
{
  "myProducts": {
    "product1": {
      "id": 4874,
      "name": "Nike - footwear",
      "price": 130
    }
  }
}

…a sample schema definition
{
  "classes": {
    "Product": {
      < Here is where all the attributes and
      properties (like id, name and price) of
      the class will be defined >
    }
  },
  "apply": [
    {
      "class": "Product",
      "to": "myProducts.product1"
    }
  ]
}
```

Figure 1 – General structure of a schema document

The following sections define the core and hyper schema, which are the attributes that can be used to define a schema class.

D. Core Schema

type: A string or array - This attribute defines what primitive type the schema instance value must contain. The allowed types are *string*, *number*, *integer*, *boolean*, *object*, *array*, *null*, *any* and a special type known as '*union type*'. If type isn't specified then the default is *any*.

Union type is an array of strings, where each string denotes a primitive type that indicates that the instance value must be either one of the types mentioned in the array.

{"type":["string","integer"]}. This indicates that the type should be either string or integer.

required: boolean - If true, indicates that the instance is mandatory. Default: false

description: string - Provides description of the instance property or schema class.

enum: Provides an enumeration of all the values that are valid for the instance property. This value must be an array which represents possible values for the instance value.

default: Defines the default value for the instance when the instance is undefined.

base: string or object - This attribute points to another schema class, which the current schema shall inherit and may extend. This attribute can be an object that has two sub-attributes, namely *class* and *schemaURI.*

class: string – Name of the class which the current schema class will extend from.

schemaURI: A URI [8] to an external JSON schema document. The schema class referenced with the *class* attribute must be defined in the mentioned schema document. If this attribute value is a special value named “$this”, then it indicates that the class is defined within the current schema document. This attribute is optional and it defaults to “$this” when not specified.

If the attribute is a string, then the string should be the name of a schema class in the current document.

format: This attribute defines the meaning of a string, integer or numeric instance value. The following formats are predefined:

- date-time : This should be a date in ISO 8601[5] format of YYYY-MM-DDThh:mm:ssZ in UTC time. This is the recommended form of date/timestamp.
- date : This should be a date in the format of YYYY-MM-DD. It is recommended that you use the "date-time" format instead of "date" unless you need to transfer only the date part.
- time : This should be a time in the format of hh:mm:ss. It is recommended that you use the "date-time" format instead of "time" unless you need to transfer only the time part.
- utc-millisec : This should be the difference, measured in milliseconds, between the specified time and midnight, 00:00 of January 1, 1970 UTC. The value should be a number (integer or float).
- color : This is a CSS color (like "#FF0000" or "red"), based on CSS 2.1 [6].
- phone : This should be a phone number (format may follow E.123 [7]).
- uri : This value should be a URI [8].

The following attributes are only significant when the instance is of the specified type (mentioned in the sub headers):

- *For objects:*

  properties: An object that defines the values (properties) that are valid within the schema instance *object*. For a property definition, all attributes that can come within a schema class can also come as attributes within a property unless otherwise stated.

  additionalProperties: boolean- “Can the instance object have additional properties which are not specified in the *properties* attribute?”. If yes then this attribute should be *true*, else *false*. Default: *true*.
- *For arrays*
  items: object or array of objects - Defines the allowed items in an instance *array*. (More explanation at section III F: Ordered Arrays)
  uniqueItems: Boolean that indicated whether the items in the array are unique.
  Two instance are consider equal if they are both of the same type and
  are null;

  or are boolean/numbers/strings and have the same value;

  or are arrays, containing the same number of items, and each item in the array is equal to the corresponding item in the other array;

  or are objects, containing the same property names, and each property in the object is equal to the corresponding property in the other object.

- *For numbers*
  minimum: number - Minimum value of an instance number.
  maximum: number - Maximum value of an instance number.
  exclusiveMinimum: boolean – If true, the value of the instance number cannot be equal to the number defined by the *minimum* attribute. Default: false
  exclusiveMaximum: boolean – If true, the value of the instance number cannot be equal to the number defined by the *maximum* attribute. Default: false
- For strings
  pattern: string- A regular expression string defined by the ECMAScript 5 standard [4]. The instance must confirm to the regular expression.

Figure 2 builds on to define the “Product” class from Figure 1. Attributes from the core schema such as *type* and *required* are used here.

```
{
 "classes": {
  "Product": {
   "type": "object",
   "properties": {
    "id": {
     "type": "integer",
     "required": true
    },
    "name": {
     "type": "string"
    },
    "price": {
     "type": "number",
     "required": true
    }
   }
  }
 },
 "apply": [
  {
   "class": "Product",
   "to": "myProducts.product1"
  }
 ]
}
```

Figure 2 - Complete schema based on JSON data from Figure 1

E. Hyper Schema

meta: An object or array where each attribute/item gives additional information about the class attributes and possible relations. This attribute can only be defined in a class (and cannot be defined anywhere within the *properties* attribute).

Attributes that come within *meta* attribute can be application specific and are open to other standards, except for the following attributes:

href: URI [8] to external resource.

method: POST or GET HTTP method

encodingType: The encoding type of the request that might be required for a POST HTTP message.

rel: A reference to a property in the *properties* object. Path to sub-properties can also be specified, which are separated from parent properties by a dot (That is, in the format property.subProperty). The path isn't a real path to the attribute, but only indicates which property is being referred to. For example, from Snippet 2 'name' is a property, but name.type isn't a property.

contentType: For string instance -The content's MIME [9] type can be specified by this attribute. For binary data encoded as a string, this attribute indicates what type of content is contained within the string, example image/jpeg or image/gif.

contentEncoding: For string instance - For binary data encoded as a string, an encoding scheme (MIME [9]) can be specified, such as application/base64.

Schema validators will pass this *meta* information to the application during validation. Figure 3 demonstrates a class with meta information. Here *href* indicates that the web link to the product is a pattern based on the product ID.

```
"Product": {
  "type": "object",
  "properties": {
              "id": {
                "type": "integer",
                "required": true
              },
              "name": {
                "type": "string"
              },
              "price": {
                "type": "number",
                "required": true
              }
  },
  "meta": {
              "ProductURL": {
                "href": "http://www.some.com/?id={id}",
                "method": "GET"
              }
  }
}
```

Figure 3 – A class with meta information giving a link to a web page based on the product ID.

F. Ordered Arrays

For instance arrays, it is possible to define the order at which the data must be arranged. This can be defined at the *items* attribute of a class.

The *items* attribute is an array of *rules*. A *rule* is either an object or a string where an object has three sub-attributes, namely *item*, *schemaURI* (optional) and *quantifier* (optional). *item* indicates a primitive type or class name. *schemaURI* has the same definition as described in Section III D - under *base* attribute.

quantifier specifies a symbol that indicates the number of occurrences of the item. The following symbols are recognized:

*: 0 or more occurrence. This is the default when no quantifier is defined.

+: At least 1 or more occurrence

?: Either 1 or no occurrence

{X}: Exactly X occurrences

{X,}: At least X occurrences

{X,Y}: Between X and Y occurrences (inclusive)

If the rule is a string, then the string should specify a primitive type or a class name. A string rule is equivalent to an object with *quantifier* as ”*” and *schemaURI* as “$this”.

Union rules can also be specified, which is an array of rules, which indicates that at least one of the rules in the array should be satisfied.

G. *JSON Schema Case Study*

Figure 4, demonstrates on how to transfer JPEG images through encoded strings. Here “rel” points to the property that contains the string. And *contentType* and *contentEncoding* helps the application to determine the content and encoding format.

```
A schema definition...
{
  "classes": {
    "Product": {
      "type": "object",
      "properties": {
        "id": {
          "type": "integer",
          "required": true
        },
        "name": {
          "type": "string"
        },
        "price": {
          "type": "number",
          "required": true
        },
        "productImage": {
          "type": "string"
        }
      },
      "meta": {
        "ProductURL": {
          "href": "http://www.some.com/?id={id}",
          "method": "GET"
        },
        "imageInfo": {
          "rel": "productImage"
          "contentType": "image/jpeg",
          "contentEncoding": "application/base64"
        }
      }
    },
    "ProductList": {
      "type": "array",
      "items": ["Product"]
    }
  },
  "apply": [
    {
      "class": "ProductList",
      "to": "myProducts"
    }
  ]
}
```

```
...for the following JSON data
{
  "myProducts": [
    {
      "id": 4874,
      "name": "Nike - footwear",
      "price": 130,
      "productImage":
"VGhpcyBpcyBhbiBpbWFnZS5JbWFnZSAx"
    },
    {
      "id": 4259,
      "name": "Reebok - footwear",
      "price": 110,
      "productImage":
"VGhpcyBpcyBhbiBpbWFnZS5JbWFnZSAy"
    },
    {
      "id": 4936,
      "name": "Addidas - footwear",
      "price": 135,
      "productImage":
"VGhpcyBpcyBhbiBpbWFnZSBJbWFnZSAz"
    }
  ]
}
```

Figure 4: An example demonstrating the use of meta data.

Figure 5, demonstrates the use of defining ordered arrays, in a case where vertices of a polygon needs to be transferred to or from a user. For a quadrilateral, we specify four pairs of vertices, which are sequentially arranged in an array. To define this structure, here we define three classes, namely vertex - which is an array of x and y co-ordinates, quadrilateral – which is an array of four “vertex” classes and quadrilateralList – which is an array of “quadrilateral” classes.

```
For the following JSON data
{
  "polygons": {
    "quadrilaterals": [
      {
        "id": 1,
        "vertices": [[10,10],[10,-10],[-10,-10],[-10,10]]
      },
      {
        "id": 2,
        "vertices": [[10,9],[10,-9],[-10,-10],[-10,10]]
      }
    ]
  }
}
```

```
A schema definition for the above data
{
 "classes": {
  "vertex": {
    "type": "array",
    "items": [{"item": "number", "quantifier": "{2}"}],
  },
  "quadrilateral": {
    "type": "object",
    "properties": {
     "id": {
       "type": "integer",
       "required": true
     },
     "vertices": {
       "type": "array",
       "items": [{"item": "vertex", "quantifier": "{4}"}],
       "required": true
     }
    }
  },
  "polygon": {
    "type": "array",
    "items": ["quadrilateral"]
  }
 },
 "apply": [
  {
    "class": "polygon",
    "to": "polygons.quadrilaterals"
  }
 ]
}
```

Figure 5: An example demonstrating the use of ordered arrays.

H. Conclusion and Direction for Future Work

This paper dealt with JavaScript Object Notation Schema. A modified schema structure based on an existing work has been proposed for use after a formal trial. It aims at bringing a higher level structure to JSON schema and it introduces a concept of schema classes. It also introduces a way to define ordered arrays. This specification can be extended by bringing additional attributes underneath the *meta* attribute. Also a mechanism to define inner classes would also be of convenience.

Schemas can help application developers to easily validate data with help of a schema and a schema validator. This way validation code need not be tightly integrated into the application. Furthermore, it also encourages reuse of code. Hence, future work will also focus on bringing a strong implementation of a JSON schema validator in various programming languages.

ACKNOWLEDGMENT

The authors are thankful to Dr. B. Vijayakumar, Associate Professor – Computer Science, BITS-Pilani for his valuable suggestions while revising this paper.

REFERENCES

[1] Gary Court, "A JSON Media Type for Describing the Structure and Meaning of JSON Documents", 2010 Nov, Internet: http://tools.ietf.org/html/draft-zyp-json-schema-03

[2] D. Crockford, "JavaScript Object Notation", ,Internet: http://tools.ietf.org/id/draft-crockford-jsonorg-json-04.txt, 2006 Feb

[3] D. Crockford, The application/json Media Type for JavaScript Object Notation, Network Working Group, The Internet Society, July 2006, Report No: RFC-4627, Sponsored by International Engineering Task Force

[4] Brendan Eich inventor, "ECMAScript Language Specification", ECMA-262 standard, ECMA international 2009, Internet: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[5] International Standard Organization, "Data elements and interchange formats — Information interchange — Representation of dates and times", Switzerland, ISO Standard, Reference: ISO 8601:2004(E), 2004 Dec 01

[6] Lie H. W., Bos B., World Wide Web Consortium," Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification", W3C Candidate Recommendation,, Section 4.3.6 - Colors, Internet: http://www.w3.org/TR/2007/CR-CSS21-20070719/syndata.html#color-units

[7] Recommendation E.123**,** **"**Notation for national and international telephone numbers, e-mail addresses and web addresses", International Telecommunication union (ITU), Internet: http://www.itu.int/rec/T-REC-E.123-200102-I/en

[8] T. Berners-Lee, R. Fielding, L. Masinter, *Uniform Resource Identifier (URI): Generic Syntax*, The Internet Society, Network Working Group, 2005, Report No: RFC- 3986, Sponsored by Internet Engineering Task Force

[9] Borenstein N., Freed N., "MIME format, Mechanisms for Specifying and Describing the Format of Internet Message Bodies", Reference: RFC 1521.Internet: http://www.faqs.org/rfcs/rfc1521.html , 1993 Sep