JSound

The complete reference

Cezar Andrei <cezar.andrei@oracle.com>
Ghislain Fourny <ghislain.fourny@28msec.com>
Daniela Florescu <dana.florescu@oracle.com>
Edited by Ghislain Fourny

JSound: The complete reference

by Cezar Andrei, Ghislain Fourny, Daniela Florescu, and Ghislain Fourny

Abstract

This document is a description of the JSound, the JSON schema definition language. It describes how to declare constraints on the structure of JSON documents.

Table of Contents

Chapter 1. Introduction

Over the past decade, the need for more flexible and scalable databases has greatly increased. The NoSQL universe brings many new ideas on how to build both scalable data storage and scalable computing infrastructures.

XML and JSON [http://www.json.org/] are probably the most popular two data formats which emerged. While XML reached a level of maturity which gives it an enterprise-ready status, JSON databases are still in their early stages. Scalable data stores (like MongoDB [http://www.mongodb.org/]) are already available. JSONiq [http://www.jsoniq.org/] brings SQL-like query capabilities to JSON. The last missing piece for a full-fledged JSON database is a way to make sure that the data stored is consistent and sound. This is where schemas come into play.

Many lessons can be learnt from 40 years of relational databases history and 15 years of XML. Just like JSONiq is inspired by SQL and XQuery, the goal of this document is to introduce a schema language, JSound, which is greatly inspired from its XML counterpart, XML Schema.

Requirements

The JSound schema definition language is based on the following requirements:

- A schema document is a well-formed JSON document, in the sense that it parses against the JSON grammar.
- A schema document is valid, in the sense that there is a JSound metaschema document against which all schema documents (including itself) are valid.
- The types of the leaf values in a valid JSON document are based on W3C XML Schema builtin atomic types [http://www.w3.org/TR/xmlschema11-2/#built-in-datatypes].
- It must be possible to mark string/value pairs in an object type as optional, and to specify a default value.
- While the schema definition language is greatly inspired from XML Schema, it must avoid its complexity.
- It should be possible, given a JSON document, to turn it into a schema against which it is valid with minimal changes.

Chapter 2. Concepts

Instance

This is a JSON value which may or not be valid against a Schema Type.

JSON values can be objects, arrays, strings, numbers, booleans and nulls.

The simplicity of JSON removes the need of defining a counterpart to the XML infoset for a logical representation of a JSON document. It barely comes down to the following:

- A JSON value is either a JSON object, a JSON array, a JSON string, a JSON number, a JSON boolean or a JSON null.
- A JSON object has an ordered list of JSON string/JSON value pairs.
- A JSON array has an ordered list of JSON values.
- A JSON string, number, boolean or null has a lexical representation which is exactly how it syntactically appears in the document (note: the quotes are not part of the lexical representation of a string).

Schema Document

This is the JSON document which defines Schema Types against which Instances are being validated. A Schema Document must be valid against the Meta Schema Type.

Meta Schema Document

This is the JSON document which defines the Schema Type against which all Schema Documents (which are also JSON documents) are valid. Including itself.

Schema Type

A Schema Document defines Schema Types, which may or may be anonymous. An Instance may or may not be valid against a Schema Type. There are four kind of Schema Types: Atomic Schema Types, Array Schema Types, Object Schema Types and Union Schema Types.

Validation

An Instance can be validated against a Schema Type. The Validation action takes an Instance and a Schema Type (typically, a set of Schema Documents and the name of a Schema Type defined in one of them). It results in a boolean that describes whether the Instance is valid against the Schema type, given its definition in the set of Schema Documents. If the Instance is not valid, a list of errors is provided, which describes how the Instance is not conforming to the Schema Type.

Annotation

Annotation is the action of passing an Instance through a Schema Type (identified with a name in a set of of Schema Documents) and recursively annotating each nested Instance (Object, Array, String, Number, Boolean, Null) in turn with the Schema Type determined by processing its parent. Note that this action is independent of Validation: The two actions can be performed in tandem but they are not required. As the Validation action, the Annotation action takes an Instance, a Schema Type (typically, a set of Schema Documents and the name of a Schema Type defined in one of them) and results in a set of annotations on the Instance that describe the Schema Types which the nested Instances match. This action also works on Instances which validate partially. When a Schema Type cannot be found which matches a given nested Instance, a special annotation is used.

Meta-keys

Schema Documents mix keys that are describing actual data fields (actual keys) and keys that define the Schema Types (meta keys). To make the distinction between the two, we use the $ (dollar sign) to represent the meta keys. In order to use $ in actual keys one should use the escaped version by doubling the $ character. This is consistent with other JSON meta languages.

Chapter 3. Atomic Types

Scope

Atomic Schema Types match JSON Strings, Numbers, Booleans and Null (JSON leaf values).

Atomic Schema Types have a lexical space (set of literals denoting the values), a value space (set of actual values), and a lexical mapping which maps the former into the latter.

An Atomic Schema Type can be either the topmost *atomic*, or a *primitive* builtin type, or a builtin type *derived* from a primitive type, or a user-defined type *derived* from any other Atomic Schema Type (except *atomic*).

Examples

Against the Atomic Schema Type represented by the following JSON document:

```
{
  "$atomic" : "string",
  "$enumeration" : [ "foo", "bar" ]
}
```

the JSON Strings "foo" and "bar" are valid. The JSON String "foobar" and the JSON array ["foo", "bar"] are not.

Against the Atomic Schema Type represented by the following JSON document:

```
{
  "$name" : "digits",
  "$atomic" : "integer",
  "$minInclusive" : 1,
  "$maxExclusive" : 10
}
```

the JSON Numbers 2 and 7 are valid. The JSON String "2", the JSON Number 0 and the JSON array ["foo", "bar"] are not.

Since it is a named Atomic Schema Type, one can also say that 2 is valid against "#digits".

Builtin Atomic Schema Types

A number of builtin Atomic Schema Types are predefined. Most of them have counterparts in XML Schema 1.1. In particular, they have the same value space, the same lexical space, the same lexical mapping and (for primitive types) the same associated set of facets.

Some of these builtin types are primitive and marked as such below. Others are derived from another builtin type.

- Builtin Atomic Schema Types for JSON Strings: string [http://www.w3.org/TR/xmlschema11-2/#string] (primitive), anyURI [http://www.w3.org/TR/xmlschema11-2/#anyURI] (primitive), date [http://www.w3.org/TR/xmlschema11-2/#date] (primitive), dateTime [http://www.w3.org/TR/xmlschema11-2/#dateTime] (primitive), time [http://www.w3.org/TR/

xmlschema11-2/#time] (primitive), dateTimeStamp [http://www.w3.org/TR/xmlschema11-2/#dateTimeStamp] (primitive), gDay [http://www.w3.org/TR/xmlschema11-2/#gDay] (primitive), gMonth [http://www.w3.org/TR/xmlschema11-2/#gMonth] (primitive), gMonthDay [http://www.w3.org/TR/xmlschema11-2/#gMonthDay] (primitive), gYear [http://www.w3.org/TR/xmlschema11-2/#gYear] (primitive), gYearMonth [http://www.w3.org/TR/xmlschema11-2/#gYearMonth] (primitive), duration [http://www.w3.org/TR/xmlschema11-2/#duration] (primitive), dayTimeDuration [http://www.w3.org/TR/xmlschema11-2/#dayTimeDuration] (derived from duration), yearMonthDuration [http://www.w3.org/TR/xmlschema11-2/#yearMonthDuration] (derived from duration), base64Binary [http://www.w3.org/TR/xmlschema11-2/#base64Binary] (primitive), hexBinary [http://www.w3.org/TR/xmlschema11-2/#hexBinary] (primitive).

- Builtin Atomic Schema Types for JSON Numbers: decimal (primitive), integer [http://www.w3.org/TR/xmlschema11-2/#integer] (derived from decimal), long [http://www.w3.org/TR/xmlschema11-2/#long] (derived from integer), int [http://www.w3.org/TR/xmlschema11-2/#int] (derived from long), short [http://www.w3.org/TR/xmlschema11-2/#short] (derived from int), byte [http://www.w3.org/TR/xmlschema11-2/#byte] (derived from short), double [http://www.w3.org/TR/xmlschema11-2/#double] (primitive), float [http://www.w3.org/TR/xmlschema11-2/#float] (primitive).
- Builtin Atomic Schema Types for JSON Booleans: boolean [http://www.w3.org/TR/xmlschema11-2/#boolean] (primitive).
- Builtin Atomic Schema Types for JSON Null: null (primitive), which has a singleton value space containing the JSON null value with the lexical representation "null".

There is also a special builtin type *atomic*, which is a supertype of all primitive types and, by transition, of all atomic types.

Derived Atomic Schema Types

It is possible to define new Atomic Schema Types by constraining the value (and lexical) space of another Atomic Schema Type. Such Atomic Schema Types are called Derived Atomic Schema Types.

Derived Atomic Schema Types may or may not have a name.

Restriction is done using the facets that are available for the base type.

JSound supports the following facets, which are defined in XML Schema 1.1. For convenience, the summary from the XML Schema 1.1 specification is provided below. Which primitive type has which facets is defined in XML Schema 1.1 as well.

- $length [http://www.w3.org/TR/xmlschema11-2/#rf-length] (integer): Constraining a value space to values with a specific number of units of length, where units of length varies depending on the base type.
- $minLength [http://www.w3.org/TR/xmlschema11-2/#rf-minLength] (integer): Constraining a value space to values with at least a specific number of units of length, where units of length varies depending on the base type.
- $maxLength [http://www.w3.org/TR/xmlschema11-2/#rf-maxLength] (integer): Constraining a value space to values with at most a specific number of units of length, where units of length varies depending on the base type.
- $pattern [http://www.w3.org/TR/xmlschema11-2/#rf-pattern] (string): Constraining a value space to values that are denoted by literals which match each of a set of regular expressions.
- $enumeration [http://www.w3.org/TR/xmlschema11-2/#rf-enumeration] (array with atomics): Constraining a value space to a specified set of values.

- $maxInclusive [http://www.w3.org/TR/xmlschema11-2/#rf-maxInclusive] (atomic): Constraining a value space to values with a specific inclusive upper bound.
- $maxExclusive [http://www.w3.org/TR/xmlschema11-2/#rf-maxExclusive] (atomic): Constraining a value space to values with a specific exclusive upper bound.
- $minExclusive [http://www.w3.org/TR/xmlschema11-2/#rf-minExclusive] (atomic): Constraining a value space to values with a specific exclusive lower bound.
- $minInclusive [http://www.w3.org/TR/xmlschema11-2/#rf-minInclusive] (atomic): Constraining a value space to values with a specific inclusive lower bound.
- $totalDigits [http://www.w3.org/TR/xmlschema11-2/#rf-totalDigits] (integer): Restricting the magnitude and arithmetic precision of values in the value spaces of decimal and datatypes derived from it.
- $fractionDigits [http://www.w3.org/TR/xmlschema11-2/#rf-fractionDigits] (integer): Placing an upper limit on the arithmetic precision of decimal values.
- $explicitTimezone [http://www.w3.org/TR/xmlschema11-2/#rf-explicitTimezone] ("required", "prohibited" or "optional"): Requiring or prohibiting the time zone offset in date/time datatypes.
- $constraints (array of strings): Constraining a value space to the values for which a set of JSONiq queries evaluates to true. In these JSONiq queries, the context item is bound to the Instance being validated.

Atomic Schema Type properties

A User-defined Atomic Schema Type is fully defined with the following properties.

- {name} (string): a string containing an EQName of the form Q{http://www.example.com/}<name> where the URL in braces is the namespace of the defining Schema, and "<name>" is its name.
- {base type definition} (string): the name of an Atomic Schema Type (the name of a builtin type, or an EQName).
- {facets} (object with the pairs correspond to the facets mentioned above): the atomic facets which constrain the value space.

There are the following constraints on these properties:

- {name} is optional.
- {base type definition} is required.
- {facets} is required, but may be empty.

JSON representation of a User-defined Atomic Schema Type

An Atomic Schema Type is represented with a JSON object. Two of the pairs define the name and base type definition, all other pairs correspond to facets. If the representation is $type, then the associated Atomic Schema Type is:

- {name} : $type."$name" prefixed with Q{<namespace of the Schema which defines this type>}.
- {base type definition} : $type."$atomic".
- {facets} : $type entirely, but without the $name and $atomic pairs.

Validity of an instance against an Atomic Schema Type.

Objects and arrays are not valid against Atomic Schema Type.

An JSON leaf value V is valid against an Atomic Schema Type A, which is (directly or indirectly, via {base type definition}) derived from the primitive type P, if:

- The JSON category of P (see list of builtin types above) must match that of V, for example if P is for matching JSON strings, then V must be a JSON string.

  The lexical representation of V is in the lexical space of P.

  It is facet-valid against each of the lexical {facets} of A.

  The value V corresponding to S given the lexical mapping of P is facet-valid against each of the value-based {facets} of A and all its base types.

JSON representation of the Object Schema Type against which the JSON representations of all Atomic Schema Types are valid.

```
{
  "$name" : "atomic-schema-type",
  "$object" : {
    "$$atomic" : { "$type" : "string" },
    "$$name" : { "$type" : "string",
                 "$optional" : true },
    "$$length" : { "$type" : "integer",
                   "$optional" : true },
    "$$minLength" : { "$type" : "integer",
                      "$optional" : true },
    "$$maxLength" : { "$type" : "integer",
                      "$optional" : true },
    "$$totalDigits" : { "$type" : "integer",
                        "$optional" : true },
    "$$fractionDigits" : { "$type" : "integer",
                           "$optional" : true },
    "$$pattern" : { "$type" : "string",
                    "$optional" : true },
    "$$enumeration" : { "$type" : { "$array" : [ "atomic" ] },
                        "$optional" : true },
    "$$whitespace" : { "$type" : { "$atomic" : "string",
                                   "$enum" : [ "collapse", "preserve", "replace
                       "$optional" : true },
    "$$maxInclusie" : { "$type" : "atomic",
                        "$optional" : true },
    "$$maxExclusie" : { "$type" : "atomic",
                        "$optional" : true },
    "$$minExclusie" : { "$type" : "atomic",
                        "$optional" : true },
    "$$minInclusie" : { "$type" : "atomic",
                        "$optional" : true },
    "$$explicitTimezone" : { "$type" : { "$atomic" : "string",
                             "$enum" : [ "required", "prohibited", "optional" ]
                             "$optional" : true },
    "$$constraints" : { "$type" : { "$array" : [ "string" ],
                        "$optional" : true },
  }
}
```

Chapter 4. Object Types

Scope

Object Schema Types match JSON objects.

There are no builtin Object Schema Types.

An Object Schema Type can either be defined by specifying its layout, or by restricting from another Object Schema Type.

In all cases, a restriction can be made with a JSONiq query that must evaluate to true.

Examples

Against the following Object Schema Type:

```
{
"$object" : {
            "foo" : {
              "$value" : "string",
              "$optional" : true
            }
          } (: see pair descriptor representation :)
"$name" : "my-object-type"
}
```

The objects {} and { "foo" : "bar" } are valid. { "bar" : true } and { "foo" : "bar", "bar" : true } are not.

Against the following Object Schema Type:

```
{
"$object" : "#my-object-type"
"$constraints" : "$$.foo eq \"bar\""
}
```

The object { "foo" : "bar" } is valid. {} and { "foo" : "bar" } are not.

Builtin Object Schema Type

There is one topmost, builtin Object Schema Type named *object*, against which all objects are valid.

Derived Object Schema Type

It is possible to define new Object Schema Types by constraining the value space of another Object Schema Type. Such Object Schema Types are called Derived Object Schema Types.

Derived Object Schema Types may or may not have a name.

Restriction is done using the object facets.

JSound supports the following object facets.

- $ordered (boolean): Forces or not a valid Instance to have its pair appear in the exact same order as defined in the Object Schema Type.
- $layout (object): the layout definition, each pair in {layout} is a pair descriptor.

  The value in a Pair Descriptor has the following required properties.

  - {type} (object): a Schema Type.
  - {default} (item): indicates that the pair is optional and that, if it is missing, the specified value (which must be valid against {type}) is the default value.

  An instance is facet-valid against $layout if each pair $key : $value is such that $key is in keys($layout) and $value is valid against $layout.$key."$type", or it is valid against $layout."$any"."$type" if $layout."$any" exists.
- $constraints (array of strings): Constraining a value space to the values for which a set of JSONiq queries evaluates to true. In these JSONiq queries, the context item is bound to the Instance being validated.

Object Schema Type properties

An Object Schema Type is fully defined with the following properties.

- {name} (string): an EQName of the form Q{http://www.example.com/}<name> where the URL in braces is the namespace of the defining Schema, and "<name>" is its name.
- {base type definition} (string) : the name of an Object Type (EQName or object).
- {facets} (object): a set of object facets which constrain the value space.

There are the following constraints on these properties.

- {name} is optional.
- {base type definition} is optional.
- {facets} is required, must at least contain $layout.

JSON representation of an Object Schema Type

An Object Schema Type is represented by a JSON object. One of the pairs is named "$object" and specifies either a base type definition or an object layout. The other pairs specify facets. If this object representation is $type, the represented Object Schema Type is as follows:

- {name} : $type."$name" prefixed with Q{<namespace of the Schema which defines this type>}.
- {base type definition} : $type."$object".
- {facets} : $type entirely, but without the $name and $object pairs.

Validity of an instance against an Object Schema Type.

JSON Arrays and leaf values are not valid against any Object Schema Type.

A JSON object O is valid against an Object Schema Type T if all of the following assertions are fulfilled:

- O is facet-valid against all {facets}

JSON representation of the Object Schema Type against which the JSON representations of all Object Schema Types are valid.

```
{
  "$name" : "object-schema-type",
  {
    "$object" : {
      "$$object" : {
        "$type" : "string",
        "$default" : "empty-object"
      },
      "$$layout" : {
        "$type" : "#object-layout"
      },
      "$$ordered" : {
        "$type" : "boolean",
        "$default" : false
      },
      "$$constraints" : {
        "$type" : { "$array" : [ "string" ] },
        "$default" : [] },
    }
  }
}

{
  "$name" : "object-layout"
  "$object" : {
    "$$any" : {
      "$type : {
        "$object" : {
          "$$type" : { "$type" : "#schema-type" },
          "$$default" : { "$type" : { { "$union" : "atomic", "$default" : "$non
        }
      }
    }
    "$any" : {
      "$type : {
        "$object" : {
          "$$type" : { "$type" : "#schema-type" },
          "$$optional" : { "$type" : "boolean",
                           "$default" : false },
          "$$default" : { "$type" : "atomic" }
        }
      }
    }
  "$ordered" : false
  }
}
```

Chapter 5. Array Types

Scope

Array Schema Types match JSON arrays.

There are no builtin Array Schema Types.

An Array Schema Type can either be defined by specifying the type of its members, or by restricting from another Array Schema Type.

In all cases, a restriction can be made with a JSONiq query that must evaluate to true.

Array Schema Type properties

An Array Schema Type is fully defined with the following properties.

- {name} : a string containing an EQName of the form Q{http://www.example.com/}<name> where the URL in braces is the namespace of the defining Schema, and "<name>" is its name.
- {base type definition} : an an Array Schema Type Definition component. Must be absent if and only if {member type definition} is present.
- {member type definition} : a (Atomic, Object, Array or Union) Schema Type Definition component. Must be absent if and only if {base type definition} is present.
- {facets} : a set of Constraining Facet components. $minLength (integer) and $maxLength (integer).
- {constraints} : a set of JSONiq queries (possibly empty).

JSON representation of an Array Type Schema Component

An Array Type Schema Component can be built from a JSON object $type as follows:

- {name} : the EQName with the name $type."$name" and the namespace is that of the Schema which defines this type.
- {target namespace} : the target namespace of the Schema in which this Schema Type is defined.
- {base type definition} : if $type."$array" is a string, the type that has the name specified by this string.
- {member type definition} : if $type."$array" is a array, the Schema Type component that is associated with the unique member of that array.
- {facets} : the Constraining Facet components, built from the ("$xxxLength", integer) pairs, which are overlayed with the {facets} of the {base type definition}.
- {constraints} : the set containing all {constraints} of the {base type definition}, augmented with the JSONiq queries serialized in the strings $type."$constraint"().

Examples

```
{
```

```
"$array" : [ "string" ]
"$name" : "my-array"
}
```

```
{
  "$array" : "#my-array-type"
  "$constraints" : "$$(1) eq \"bar\""
}
```

Validity of an instance against an Array Schema Type.

Atomics and objects are not valid against any Array Schema Type.

An array A is valid against an Array Schema Type T if all of the following assertions are fulfilled:

- If {base type definition} is present, A is valid against it.

  If {member type definition} is present, each member of A matches the {member type definition} Schema Type component.

  All queries in {constraints} evaluate to true when the context item is bound to A.

JSON representation of the Object Schema Type against which the JSON representations of all Array Schema Types are valid.

```
{
  "$name" : "array-schema-type",
  "$union" : [
    "string",
    {
      "$object" : {
        "$$array" : { $array : [ "#schema-type" ], "$minLength" : 1, "$maxLengt
        "$$minLength" : { "$type" : "integer",
                          "$optional" : true },
        "$$maxLength" : { "$type" : "integer",
                          "$optional" : true },
        "$$constraints" : { "$type" : { "$array" : [ "string" ],
                            "$optional" : true },
      }
    }
  ]
}
```

Chapter 6. Union Types

Scope

The value space of a Union Schema Type is the union of the value spaces of all its member types.

Union Schema Type Definition component

An Union Schema Type is fully defined with the following properties.

- {name} : a string containing an EQName of the form Q{http://www.example.com/}<name> where the URL in braces is the namespace of the defining Schema, and "<name>" is its name.
- {target namespace} : a string containing a URI. Optional.
- {member type definitions} : a set of (Atomic, Object, Array or Union) Schema Type Definition components (possibly empty).
- {constraints} : a set of JSONiq queries (possibly empty).

JSON representation of a Union Schema Type component

A Schema Type component can be built using either a JSON object representation (to define a new Schema Type) or a string representation (to refer to an existing type).

A Union Type Schema Component can be built from a JSON object $type as follows:

- {name} : the EQName with the name $type."$name" and the namespace is that of the Schema which defines this type.
- {target namespace} : the target namespace of the Schema in which this Schema Type is defined.
- {member type definitions} : the set of the Schema Type Definition components represented by the members of the array $type."$union"
- {constraints} : the set containing all {constraints} of the {base type definition}, augmented with the JSONiq queries serialized in the strings $type."$constraint"().

Examples

```
{
"$union" : [ "string", { "$array" : [ "integer" ] } ]
}
```

Validity of an instance against a Union Schema Type.

An instance I is valid against a Union Schema Type (corresponding to the component T) if all of the following assertions are fulfilled:

- I is valid against at least one of the Schema Types corresponding to the components in {member type definitions}.

All queries in {constraints} evaluate to true when the context item is bound to A.

JSON representation of the Object Schema Type against which the JSON representations of all Union Schema Types are valid.

```
{
  "$name" : "union-schema-type",
  "$union" : [
    "string",
    {
      "$object" : {
        "$$union" : { $array : [ "#schema-type" ] },
        "$$constraints" : { "$type" : { "$array" : [ "string" ],
                            "$optional" : true },
      }
    }
  ]
}
```

```
{
  "$name" : "schema-type",
  "$union" : [
    "#atomic-schema-type",
    "#object-schema-type",
    "#array-schema-type",
    "#union-schema-type"
  ]
}
```

Topmost type

There is a topmost type named item. It is the union of atomic, object and array.

```
{
  "$name" : "item",
  "$union" : [ "object", "array", "atomic" ]
}
```

Appendix A. Revision History

Revision History			
Revision 0-0	Thu May 23 2013	Dude	McPants
		`<Dude.McPants@example.com>`	
Initial creation of book by publican			

Index

18