JSound 0.1

JSound

The complete reference

Cezar Andrei

Ghislain Fourny

Daniela Florescu

JSound 0.1 JSound
The complete reference
Edition 0.1.3

Author	Cezar Andrei	*cezar.andrei@oracle.com*
Author	Ghislain Fourny	*ghislain.fourny@28msec.com*
Author	Daniela Florescu	*dana.florescu@oracle.com*
Editor	Ghislain Fourny	*ghislain.fourny@28msec.com*

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.

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*[1] are probably the most popular two data formats that emerged. While XML reached a level of maturity that gives it an enterprise-ready status, JSON databases are still in their early stages. Scalable data stores (like *MongoDB*[2]) are already available. *JSONiq*[3] 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 learned from 40 years of relational databases history and 15 years of XML. The goal of this document is to introduce a schema language, JSound, which is much simpler than XML Schema, just like JSON syntax is much simpler than XML syntax.

1.1. Requirements

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

- A schema document must be a well-formed JSON document in the sense that it parses against the JSON grammar.
- A schema document must be valid in the sense that there is a JSound metaschema document against which all schema documents (including itself) are valid.
- While the schema definition language is greatly inspired from XML Schema, it must avoid its complexity. It must be simpler and more readable.
- JSound must support most of the *XML Schema primitive atomic types*[4], as many of them are very useful and completely orthogonal to XML.
- JSound must avoid the XML Schema model of restriction/extension of structured types. Instead, it must support a different model of subtyping based on classical object-oriented inheritance. In JSound, a subtype's value space must always be a subset of its base type's value space.
- It must be possible to mark string/value pairs in an object type as optional or to specify a default value in case of absence.
- It should be possible, given a JSON document, to turn it into a schema against which it is valid with minimal changes.

[1] http://www.json.org/
[2] http://www.mongodb.org/
[3] http://www.jsoniq.org/
[4] http://www.w3.org/TR/xmlschema11-2/#built-in-datatypes

Concepts

2.1. Candidate Instance

This is a JDM (JSONiq Data Model) value. A Candidate Instance may or not be valid against a Schema Type.

In the JSONiq Data Model, instances can be objects, arrays, or atomics.

- An object has an unordered list of string/value pairs. A top-level object is also referred to as a "JSON document".
- An array has an ordered list of values.
- An atomic has a value annotated with an atomic type.

Typically, the Candidate Instance will have been freshly parsed and will only have atomics of type string, integer, decimal, double, boolean and null. Integer, decimal and double values correspond to parsed JSON numbers, depending on the presence of dots and scientific notation.

2.2. Annotated Instance

An Annotated Instance is a "Post-validation JDM Instance", i.e., it corresponds to a Candidate Instance that has been recursively annotated after going through the Annotation process against a Type.

2.3. Schema Document

A Schema Document is a JSON document which defines Types against which Candidate Instances are being validated. A Schema Document is also a Candidate Instance and must be valid against the Meta Schema Type.

2.4. Meta Schema Document

A Meta Schema Document is a JSON document that defines the Type against which all Schema Documents are valid including itself.

2.5. Type

A Schema Document defines Types, which may or may not be anonymous. A Candidate Instance may or may not be valid against a Type. A Candidate Instance can be annotated against a Type, which results in a Data Model Instance. There are four kinds of Types: Atomic, Array, Object and Union. Types are represented with objects that are nested in a Schema Document. Named Types can also be referred to with Qualified Names.

2.6. Namespace

A JSound Schema Document is associated with a Namespace, and all types that it defines live in that Namespace. However, Namespaces in JSound are much closer to C++ namespaces or Java packages than to XML namespaces. Prefixes may still, but must not be, used as convenient shorcuts to avoid writing long URIs.

2.7. Qualified Name

Type names are Qualified Names, which are made of a namespace and of a local name. Local names must be unique in a namespace, but not across namespaces -- like in C++ or Java. Qualified Names have the following string representation: "Q{<namespace>}<local name>". For example: "Q{http://www.example.com/types}small-integer"

For convenience, Namespaces that are imported can be bound to prefixes in the containing Schema Document, and the prefix can be used as a shortcut to the full namespace like so: "<prefix>:<local name>". If the prefix "my" is bound to the Namespace "http://www.example.com/types" in the Schema Document, the above example can also be represented as "my:small-integer".

Builtin Type names are special in that they are in no Namespace and can also be represented with no prefix : "integer".

2.8. Validation

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

2.9. Annotation

Annotation is the action of passing a Candidate Instance through a Schema Type (identified with a name in a set of of Schema Documents) and recursively:

- annotating an object or an array with the current Schema Type as well as determining against which Schema Type the object pair values or array members will be annotated.

  casting an atomic to the current Schema Type.

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 a Candidate Instance, a Type (typically, a set of Schema Documents and the name of a Type defined in one of them) and results in a set of annotations on the Instance that describe the Types which the nested Instances match. This action also works on Instances that validate partially. When a Type cannot be found that matches a given nested Candidate Instance, a special annotation is used.

2.10. Meta Keys

Schema Documents mix keys that are describing actual data fields (actual keys) and keys that define the 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.

Schema Documents

3.1. Scope

Schema Documents have a namespace and define multiple Types in this namespace.

3.2. Schema Document properties

Schema Documents are (serialized) JSON objects which have the following properties

- $namespace (JSON string): the namespace (URI) in which the types defined in this Schema Document live.
- $about (JSON value): free content (documentation, comments, ...).
- $imports (JSON array of objects) : used to import Types located in other Schema Documents (i.e., in other Namespaces). Imports are not recursive, i.e., importing a Schema Document does not import the further Schema Documents that it may itself import.

  Each JSON object in this JSON array has the following properties:

  - $namespace (JSON string): the namespace in which the Types being imported live.
  - $location (JSON string): a hint about where to find the Schema Document for this namespace.
  - $prefix (JSON string): the prefix used in Qualified Names to refer to Types in the above namespace. Must not contain a colon.
- $types (JSON array of objects representing Types) : the Types defined in this document. How these objects look like is explained in subsequent sections.

3.3. Examples

This Schema Document defines two Atomic Types in the "http://www.example.com/my-schema" namespace and with the local names "small-number" and "big-number".

```
{
  "$namespace" : "http://www.example.com/my-schema",
  "$types" : [
    {
      "$kind" : "atomic",
      "$name" : "small-number",
      "$baseType" : "integer",
      "$enumeration" : [ 1, 2, 4, 8 ]
    },
    {
      "$kind" : "atomic",
      "$name" : "Q{http://www.example.com/my-schema}big-number",
      "$baseType" : "integer",
      "$enumeration" : [ 1000, 2000, 4000, 8000 ]
    }
  ]
}
```

This Schema Document defines one Object Type in the "http://www.example.com/my-new-schema" namespace named "small-and-big".

```
{
  "$namespace" : "http://www.example.com/my-new-schema",
  "$imports" : [
    {
      "$namespace" : "http://www.example.com/my-schema",
      "$prefix" : "other"
    }
  ],
  "$types" : [
    {
      "$kind" : "object",
      "$name" : "small-and-big",
      "$content" : {
        "small" : { "$type" : "other:small-number" },
        "big" : { "$type" : "other:big-number", "$optional" : true }
      }
    }
  ]
}
```

Given this set of two Schema Documents, the following JSON object:

```
{
  "small" : 4
}
```

is valid against the Type named "Q{http://www.example.com/my-new-schema}small-and-big".

This JSON object is not valid, because the value associated with "big" is not in the value space of the Type "Q{http://www.example.com/my-schema}big-number".

```
{
  "small" : 4,
  "big" : 3
}
```

3.4. Type Names and references to Types

Type Names are Qualified Names, made of a namespace and of a local name, as described in the former chapter.

Types names are used to (optionally) name Types, or to refer to another Type as a base type.

References to Types that are defined in the same Schema Document can be referred to with no prefix as well: the namespace is that of the defining Schema Document. If there is a collision with Builtin Type names, the locally defined Type has precedence (the Builtin Type is hidden).

3.5. Types

There are four kinds of Types: atomic, object, array and union.

Types are either Builtin, in which case their name is in no namespace, or Derived.

The topmost Type is builtin and is named "item".

The topmost Object Type is builtin and is named "object".

The topmost Array Type is builtin and is named "array".

The topmost Atomic Type is builtin and is named "atomic". There are many further Builtin Atomic Types.

Derived Types are always defined by restricting the value space of a *base type* by means of *facets*. They have a JSON object representation.

Derived Object Types are always directly derived from "object". Derived Array Types are always directly derived from "array". Derived Union Types are always directly derived from "item". Derived Atomic Types may be derived from any other Atomic Type.

3.6. Derived Type properties

A Derived Type has the following properties:

- $kind (JSON string): the kind of the Type. One of "atomic, "object", "array", "union".
- $name (JSON string): a string containing the Qualified Name (as defined above) of this Type.
- $baseType (JSON string): a string containing the Qualified Name of the Type which is the base type of this Type.
- $about (JSON value): free content (documentation, comments, ...).
- various facets properties. Which facets are available defines on the $kind of the Type.

There are the following constraints on these properties:

- $name is optional and must live in the namespace of the Schema Document in which this Type is defined.
- A prefix may not appear twice in the $imports.
- Types defined directly in the top-level $types array must be named.
- $baseType must refer to a known Type - builtin, in the same Schema Document or in an imported Schema Document. In particular, if a prefix is used, it must be bound to an imported namespace.
- If $kind is "object", $baseType must be "object" if provided.
- If $kind is "array", $baseType must be "array" if provided.
- If $kind is "union", $baseType must be "item" if provided.
- If $kind is "atomic", $baseType must be the Qualified Name of an existing Atomic Type.

Here is an example of an invalid Schema Document, because it does not fulfill many of the above constraints.

```
{
  "$namespace" : "http://www.example.com/my-schema",
  "$types" : [
    {
      "$kind" : "atomic",
      "$name" : "type1",
      "$baseType" : "unbound:type", (: prefix is not bound :)
      "$maxInclusive" : 4
    },
```

```
    {
      "$kind" : "atomic",
      "$name" : "Q{http://www.example.com/other}type2", (: the namespace must match that of
 the Schema document :)
      "$baseType" : "integer",
      "$maxInclusive" : 4
    },
    {
      "$kind" : "atomic",
      "$name" : "Q{http://www.example.com/my-schema}type3",
      "$baseType" : "object", (: base type MUST also be an atomic type :)
      "$maxInclusive" : 4
    },
    {
      "$kind" : "object",
      "$name" : "object1",
      "$baseType" : "type1" (: base type MUST be "object":)
      "$content" : {}
    },
    {
      "$kind" : "object",
      "$name" : "object2",
      "$baseType" : "object1" (: base type MUST be "object":)
    }
  ]
}
```

There are two facets common to all types:

- $enumeration (array of JSON values): Constrains a value space to a specified set of values.
- $constraints (array of JSON strings): Constrains 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 Serialized Instance being validated, after parsing.

```
{
  "$namespace" : "http://www.example.com/my-schema",
  "$types" : [
    {
      "$kind" : "object"
      (: "$baseType" : "object" is implicit :)
      "$name" : "two-objects",
      "$enumeration" : [ { "foo" : "bar" }, {} ] (: only these two objects :)
    },
    {
      "$kind" : "array"
      (: "$baseType" : "array" is implicit :)
      "$name" : "uniform-array",
      "$constraints" : [ "every $i in 1 to size($$) satisfies deep-equals($$($i), $$(1))" ]
 (: all members must be the same :)
    },
  ]
}
```

The following JSON object is valid against Q{http://www.example.com/my-schema}two-objects.

```
{ "foo" : "bar" }
```

The following JSON array is valid against Q{http://www.example.com/my-schema}uniform-array.

```
```

```
[ 42, 42, 42 ]
```

Atomic Types

4.1. Scope

Atomic Types match atomics (JSON leaf values: strings, numbers, booleans, nulls).

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

An Atomic 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 Type (except *atomic*).

A Derived Atomic Type can be defined by restricting the value space of another Atomic Type by specifying atomic facets. A restriction can also be made with the general facets $enumeration and $constraints.

4.2. Examples

Given the following Schema Document:

```
{
  "$namespace" : "http://www.example.com/my-schema",
  "$types" : [
    {
      "$kind" : "atomic",
      "$name" : "foo-and-bar",
      "$baseType" : "string",
      "$enumeration" : [ "foo", "bar" ]
    },
    {
      "$kind" : "atomic",
      "$name" : "digits",
      "$baseType" : "integer",
      "$minInclusive" : 1,
      "$maxExclusive" : 10
    },
    {
      "$kind" : "atomic",
      "$name" : "few-digits",
      "$baseType" : "my:digits",
      "$enumeration" : [ 4, 6 ]
    }
  ]
}
```

The strings "foo" and "bar" are valid against Type named "Q{http://www.example.com/my-schema}foo-and-bar". The string "foobar" and the array ["foo", "bar"] are not.

The atomics (integers) 2 and 7 are valid against the Type named "Q{http://www.example.com/my-schema}digits". The string "2", the integer 0 and the array ["foo", "bar"] are not.

The integer 4 is valid against the Type named "Q{http://www.example.com/my-schema}few-digits". The integer 2, the integer 0 and the array ["foo", "bar"] are not.

4.3. Builtin Atomic Types

A number of builtin Atomic Types are predefined. Most of them have counterparts in XML Schema 1.1, because they are very useful also in JSON (for example : dates, times, ...). 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 atomic facets.

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

- *string*[1] (primitive),
- *anyURI*[2] (primitive),
- *base64Binary*[3] (primitive),
- *hexBinary*[4] (primitive).
- *date*[5] (primitive),
- *dateTime*[6] (primitive),
- *time*[7] (primitive),
- *dateTimeStamp*[8] (derived from dateTime),
- *gYear*[9] (primitive),
- *gYearMonth*[10] (primitive),
- *gMonth*[11] (primitive),
- *gMonthDay*[12] (primitive),
- *gDay*[13] (primitive),
- *duration*[14] (primitive),
- *dayTimeDuration*[15] (derived from duration),
- *yearMonthDuration*[16] (derived from duration),
- *decimal*[17] (primitive),
- *integer*[18] (derived from decimal),

[1] http://www.w3.org/TR/xmlschema11-2/#string
[2] http://www.w3.org/TR/xmlschema11-2/#anyURI
[3] http://www.w3.org/TR/xmlschema11-2/#base64Binary
[4] http://www.w3.org/TR/xmlschema11-2/#hexBinary
[5] http://www.w3.org/TR/xmlschema11-2/#date
[6] http://www.w3.org/TR/xmlschema11-2/#dateTime
[7] http://www.w3.org/TR/xmlschema11-2/#time
[8] http://www.w3.org/TR/xmlschema11-2/#dateTimeStamp
[9] http://www.w3.org/TR/xmlschema11-2/#gYear
[10] http://www.w3.org/TR/xmlschema11-2/#gYearMonth
[11] http://www.w3.org/TR/xmlschema11-2/#gMonth
[12] http://www.w3.org/TR/xmlschema11-2/#gMonthDay
[13] http://www.w3.org/TR/xmlschema11-2/#gDay
[14] http://www.w3.org/TR/xmlschema11-2/#duration
[15] http://www.w3.org/TR/xmlschema11-2/#dayTimeDuration
[16] http://www.w3.org/TR/xmlschema11-2/#yearMonthDuration
[17] http://www.w3.org/TR/xmlschema11-2/#decimal
[18] http://www.w3.org/TR/xmlschema11-2/#integer

- *long*[19] (derived from integer),
- *int*[20] (derived from long),
- *short*[21] (derived from int),
- *byte*[22] (derived from short),
- *double*[23] (primitive),
- *float*[24] (primitive).
- *boolean*[25] (primitive)
- 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.

The lexical namespace of dateTime as defined in XML Schema 1.1 is a superset of the date representation defined in *ECMAScript*[26]. In addition, JSound extends the lexical representation of respectively date, time, dateTime defined above, to allow the format defined in *RFC 2822*[27] (nonterminals date, time, date-time respectively). This is because many JavaScript implementations do so.

4.4. Atomic facets

Restriction is done using the general facets, or the following atomic facets (they must be available for the base type).

These facets 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.

The following atomic facets are available for the primitive types string, anyURI, base64Binary, hexBinary:

- *$length*[28] (integer): Constraining a value space to values with a specific number of units of length, where units of length varies depending on the base type.
- *$minLength*[29] (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*[30] (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.

[19] http://www.w3.org/TR/xmlschema11-2/#long
[20] http://www.w3.org/TR/xmlschema11-2/#int
[21] http://www.w3.org/TR/xmlschema11-2/#short
[22] http://www.w3.org/TR/xmlschema11-2/#byte
[23] http://www.w3.org/TR/xmlschema11-2/#double
[24] http://www.w3.org/TR/xmlschema11-2/#float
[25] http://www.w3.org/TR/xmlschema11-2/#boolean
[26] http://www.ecma-international.org/ecma-262/5.1/#sec-15.9.1.15
[27] http://tools.ietf.org/html/rfc2822#page-14
[28] http://www.w3.org/TR/xmlschema11-2/#rf-length
[29] http://www.w3.org/TR/xmlschema11-2/#rf-minLength
[30] http://www.w3.org/TR/xmlschema11-2/#rf-maxLength

The following atomic facets are available for the primitive types date, dateTime, time, gYear, gYear-Month, gMonth, gMonthDay, gDay, duration, decimal, double, float:

- *$maxInclusive*[31] (atomic): Constraining a value space to values with a specific inclusive upper bound.
- *$maxExclusive*[32] (atomic): Constraining a value space to values with a specific exclusive upper bound.
- *$minExclusive*[33] (atomic): Constraining a value space to values with a specific exclusive lower bound.
- *$minInclusive*[34] (atomic): Constraining a value space to values with a specific inclusive lower bound.

The following atomic facets are available for the primitive type decimal:

- *$totalDigits*[35] (integer): Restricting the magnitude and arithmetic precision of values in the value spaces of decimal and datatypes derived from it.
- *$fractionDigits*[36] (integer): Placing an upper limit on the arithmetic precision of decimal values.

The following atomic facets are available for the primitive types date, dateTime, time:

- *$explicitTimezone*[37] ("required", "prohibited" or "optional"): Requiring or prohibiting the time zone offset in date/time datatypes.

The following atomic facets are available for all primitive types (including boolean and null):

- *$pattern*[38] (string): Constraining a value space to values that are denoted by literals which match each of a set of regular expressions.

[31] http://www.w3.org/TR/xmlschema11-2/#rf-maxInclusive
[32] http://www.w3.org/TR/xmlschema11-2/#rf-maxExclusive
[33] http://www.w3.org/TR/xmlschema11-2/#rf-minExclusive
[34] http://www.w3.org/TR/xmlschema11-2/#rf-minInclusive
[35] http://www.w3.org/TR/xmlschema11-2/#rf-totalDigits
[36] http://www.w3.org/TR/xmlschema11-2/#rf-fractionDigits
[37] http://www.w3.org/TR/xmlschema11-2/#rf-explicitTimezone
[38] http://www.w3.org/TR/xmlschema11-2/#rf-pattern

Object Types

5.1. Scope

Object Types match objects.

There is one builtin Object Type: "object" which is the direct base type of all other Object Types.

An Object Type can be defined by restricting the value space of "object" by specifying a layout (type of the pairs, optional or not, ...). A restriction can also be made with the general Types facets $enumeration and $constraints.

5.2. Examples

Against the following Object Type:

```
{
  "$namespace" : "http://www.example.com/my-schema",
  "$types" : [
    {
     "$kind" : "object",
     "$content" : {
       "foo" : {
         "$type" : "string",
       }
     },
     "$open" : false,
     "$name" : "only-foo"
    },
    {
      "$kind" : "object",
      "$content" : {
        "foo" : {
          "$type" : "string",
        },
        "bar" : {
         "$type" : "boolean",
         "$optional" : true
       }
     }
    "$name" : "foo-bar-and-arrays"
    }
  ]
}
```

The objects { "foo" : "bar" } and { "foo" : "foo" } are valid against the Type named "Q{http://www.example.com/my-schema}only-foo" because the foo pairs are strings.

The object {} is not because the foo pair is missing.

The object { "foo" : "bar", "bar" : "foo" } is not because no other pair than "foo" is allowed (closed Object Type).

Against the Type named "Q{http://www.example.com/my-schema}only-foo":

The objects { "foo" : "bar", "foobar" : ["foo"] } and { "foo" : "bar", "bar" : true } are valid because the foo pairs are strings, bar is optional and the Object Type is $open.

The objects {} and { "bar" : "foo" } and { "foo" : "bar", "bar" : "foo" } are not because the foo pair is missing or the bar pair is not a boolean.

5.3. Builtin Object Type

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

This topmost type can be seen as having its $content facet as the empty object, and its $open facet as true.

5.4. Object facets

Restriction is done using the general facets, or the following object facets. For the moment, restriction can only be made on the topmost object type, but this will be relaxed later.

- $content (object): the layout definition. Each pair in $content is called a field descriptor. The value in each field descriptor has the following properties.
  - $type (string or object) - required: the name of a Type (Qualified Name in a string) or the type itself (an object) that the value must match.
  - $optional (boolean) - optional: indicates that the pair is optional. Default is false.
  - $default (item) - optional: indicates a default value to be taken the value is missing in the Serialized Instance. $optional is then ignored.

    However, if this value is an object with a pair named $computed (which must be associated with a string), then the JSONiq query in $content.$default.$computed is executed upon Annotation, with the context item bound to the Candidate Instance being matched against the containing Object Type. It must result in one item, which is the default value for the Pair Descriptor.

  An object $o is valid against the $content facet if the following conditions are met:

  - For each pair $k : $v in the field descriptor such that $v."$optional" is false and $v."$default" is absent, there must be a pair named $k in $o.
  - For each pair $k : $v in the field descriptor, if $o.$k exists, then $o.$k must be valid against the Type $v."$type".
- $open (boolean) : specifies whether pairs not specified in $content are to be accepted. The default is the same as the $baseType (true if $baseType is object).

  All objects are valid against the $open facet if it is set to true.

  If it is set to false, an object $o is valid against the $open facet if all its keys appear in $content, or in the $content of a super type.

The object facets must fulfill the following consistency constraints against the super types (i.e., in the transitive closure of the $baseType relationship).

These constraints make sure that the new value space is a subset of the base type's value space.

- If the $baseType's $open property is false, then $open cannot be set back to true.
- Field descriptors on new keys may only be defined in $content if the $baseType's $open property is true.

- Field descriptors on keys that were already defined in a super type are only allowed if they are more restrictive, i.e., $type must be a subtype of the $type associated to this key by the closest super type which does so.
- If a field descriptor redefines a key that was not $optional in the closest super type, $optional cannot be set back to true.

Note: since currently, the $baseType must be "object", these constraints are always fulfilled.

Array Types

6.1. Scope

Array Types match arrays.

There is one builtin topmost Array Type "array".

An Array Type can be defined by restricting the value space of "array" by specifying a layout (type of the members) or size bounds. A restriction can also be made with the general Types facets $enumeration and $constraints.

6.2. Examples

```
{
  "$namespace" : "http://www.example.com/my-schema",
  "$types" : [
    {
      "$kind" : "array",
      "$content" : [ "string" ],
      "$name" : "strings"
    },
    {
      "$kind" : "array",
      "$content" : [ "string" ],
      "$maxLength" : 5,
      "$name" : "less-than-five-members"
    },
    {
      "$kind" : "array",
      "$content" : [ "integer" ],
      "$constraints" : [ "every $i in $$ satisfies $i le 10" ],
      "$name" : "all-less-than-ten"
    }
  ]
}
```

["foo " "bar"] is valid against the Type named "Q{http://www.example.com/my-schema}strings" but not [1, 2, "foo"].

```

```

["foo " "bar"] is valid against the Type named "Q{http://www.example.com/my-schema}less-than-five-members" but not ["foo", "foo", "foo", "foo", "foo", "foo"].

```

```

[1, 3, 5] is valid against the Type named "Q{http://www.example.com/my-schema}all-less-than-ten" but not [1, 3, 72].

6.3. Builtin Array Type

There is one topmost, builtin Array Type named *array*, against which all arrays are valid.

6.4. Array facets

Restriction is done using the general facets, or the following array facets.

JSound supports the following array facets.

- $content (singleton array of one string or object) : the name of a Type (Qualified Name in a string) or the type itself (an object) that all members must match.
- $minLength (integer) : the minimum length.
- $maxLength (integer) : the maximum length.

Union Types

7.1. Scope

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

There is no Builtin Union Type. All Union Types have directly the topmost "item" as their base type and restrict the value space by specifying the $content facet. General facets can also be used.

7.2. Examples

```
{
  "$namespace" : "http://www.example.com/my-schema",
  "$types" : [
    {
    "$kind" : "union",
    "$content" : [ "string", { "$kind" : "array", "$content" : [ "integer" ] } ],
    "$name" : "string-or-integer-array"
    },
    {
    "$kind" : "union",
    "$content" : [ "string", { "$kind" : "array", "$content" : [ "integer" ] } ],
    "$enumeration" : [ "foo", [ 1, 2, 3, 4 ] ],
    "$name" : "just-two"
    }
  ]
}
```

"foo", "bar" and [1, 2, 3] are valid against the Type named "Q{http://www.example.com/my-schema}string-or-integer-array" but 3.14 and true are not.

"foo", and [1, 2, 3, 4] are valid against the Type named "Q{http://www.example.com/my-schema}just-two" but [1] and "bar" are not.

7.3. Union facets

The specification of member types is done using one (compulsory) union facet, and optionally general facets.

- $content (array of (string or object)) : each member in the array is the name of a Type (Qualified Name in a string) or the member type itself (an object).

Validation and Annotation

8.1. Validation

A Candidate Instance is valid against a Builtin Type if it is in its value space.

A Candidate Instance is valid against a Derived Type if it is valid against its $baseType (recursively) and if it is valid against all facets.

8.2. Annotation

A Candidate Instance is annotated against an Atomic Type as follows:

- If it is valid against the Type (which implies that it is an atomic value), it is cast to the Atomic Type.
- Otherwise, it is replaced with an object with the fields $invalid (true), $expected (Atomic Type name), $value (the Candidate Instance)

A Candidate Instance is annotated against an Object Type $t as follows:

- If it is valid against the Type (which implies that it is an object), it is annotated with the Object Type's Qualified Name (if it has any).
- If it is valid against the Type, each pair value associated with a key $key is annotated recursively against the Types described with $t."$content".$key."$type". For missing pairs for which a $t."$content".$key."$default" value is provided, a new pair with the (possibly computed) $default value is added.
- Otherwise, it is replaced with an object with the fields $invalid (true), $expected (Object Type name), $value (the Candidate Instance)

A Candidate Instance is annotated against an Array Type $t as follows:

- If it is valid against the Type (which implies that it is an array), it is annotated with the Array Type's Qualified Name (if it has any).
- If it is valid against the Type, each member is annotated recursively against the Type described with $t."$content"(1).
- Otherwise, it is replaced with an object with the fields $invalid (true), $expected (Array Type name), $value (the Candidate Instance)

A Candidate Instance is annotated against a Union Type $t as follows:

- If it is valid against the Type, then it is annotated against the first Type of $t."$content"() against which the Candidate Instance is valid.
- Otherwise, it is replaced with an object with the fields $invalid (true), $expected (Union Type name), $value (the Candidate Instance)

Schema of Schemas

```
{
  "$namespace" : "http://www.jsound.org/schemaschema",
  "$types" : [
    {
      "$kind" : "object",
      "$name" : "atomic-type",
      "$content" : {
        "$$kind" : {
          "$type" : {
            "$kind" : "atomic",
            "$baseType" : "string",
            "$enumeration" : [ "atomic" ]
          }
        },
        "$$name" :             { "$type" : "qualified-name", "$optional" : true },
        "$$baseType" :         { "$type" : "qualified-name" }
        "$$pattern" :          { "$type" : "string", "$optional" : true },
        "$$length" :           { "$type" : "integer","$optional" : true },
        "$$minLength" :        { "$type" : "integer","$optional" : true },
        "$$maxLength" :        { "$type" : "integer","$optional" : true },
        "$$totalDigits" :      { "$type" : "integer","$optional" : true },
        "$$fractionDigits" :   { "$type" : "integer","$optional" : true },
        "$$maxInclusive" :     { "$type" : "atomic", "$optional" : true },
        "$$maxExclusive" :     { "$type" : "atomic", "$optional" : true },
        "$$minExclusive" :     { "$type" : "atomic", "$optional" : true },
        "$$minInclusive" :     { "$type" : "atomic", "$optional" : true },
        "$$explicitTimezone" : {
          "$type" : {
            "$kind" : "atomic",
            "$baseType" : "string",
            "$enumeration" : [ "required", "prohibited", "optional" ]
          },
          "$optional" : true
        },
        "$$enumeration" : {
          "$type" : { "$kind" : "array", $content" : [ "atomic" ] },
          "$optional" : true
        },
        "$$constraints" : {
          "$type" : { "$kind" : "array", "$content" : [ "string" ] },
          "$optional" : true
        }
      }
    },
    {
      "$kind" : "object",
      "$name" : "object-type",
      "$content" : {
        "$$kind" : {
          "$type" : {
            "$kind" : "atomic",
            "$baseType" : "string",
            "$enumeration" : [ "object" ]
          }
        },
        "$$name" : { "$type" : "qualified-name", "$optional" : true },
        "$$content" : {
          "$type" : {
            "$kind" : "object",
            "$constraints" : [ "every $key in keys($$) satisfies $$.$key instance of pair-
descriptor" ]
          }
```

```
          "$optional" : true
        },
        "$$open" : { "$type" : "boolean", $optional" : true },
        "$$enumeration" : {
          "$type" : { "$kind" : "array", $content" : [ "atomic" ] },
          "$optional" : true
        },
        "$$constraints" : {
          "$type" : { "$kind" : "array", "$content" : [ "string" ] },
          "$optional" : true
        }
      }
    },
    {
      "$kind" : "object",
      "$name" : "pair-descriptor"
      "$content" : {
        "$$type" : { "$type" : "type-or-reference" },
        "$$optional" : { "$type" : "boolean", "$default" : "false" }
        "$$default" : { "$type" : "item", "$optional" : true }
      }
    },
    {
      "$kind" : "object",
      "$name" : "array-type",
      "$content" : {
        "$$kind" : {
          "$type" : {
            "$kind" : "atomic",
            "$baseType" : "string",
            "$enumeration" : [ "array" ]
          }
        },
        "$$name" : { "$type" : "qualified-name", "$optional" : true },
        "$$content" : {
          "$type" : {
            "$kind" : "array",
            "$content" : [ "type-or-reference" ],
            "$minLength" : 1,
            "$maxLength" : 1
          }
        },
        "$$minLength" : { "$type" : "integer", "$optional" : true },
        "$$maxLength" : { "$type" : "integer", "$optional" : true },
        "$$enumeration" : {
          "$type" : { "$kind" : "array", $content" : [ "atomic" ] },
          "$optional" : true
        },
        "$$constraints" : {
          "$type" : { "$kind" : "array", "$content" : [ "string" ] },
          "$optional" : true
        }
      }
    },
    {
      "$kind" : "object",
      "$name" : "union-type",
      "$content" : {
        "$$kind" : {
          "$type" : {
            "$kind" : "atomic",
            "$baseType" : "string",
            "$enumeration" : [ "union" ]
          }
        },
        "$$name" : { "$type" : "qualified-name", "$optional" : true },
```

```
        "$$content" : { "$type" : { "$kind" : "array", "$content" : [ "type-or-
reference" ] } },
        "$$enumeration" : {
          "$type" : { "$kind" : "array", $content" :  [ "atomic" ] },
          "$optional" : true },
        "$$constraints" : {
          "$type" : { "$kind" : "array", $content" :  [ "string" ] },
          "$optional" : true
        }
      }
    },
    {
      "$kind" : "atomic",
      "$name" : "qualified-name",
      "$baseType" : "string",
      "$pattern" : "([^:${}]+:|Q{[^${}]+})?[^:${}]+"
    },
    {
      "$kind" : "union",
      "$name" : "type-or-reference",
      "$content" : [ "qualified-name", "atomic-type", "object-type", "array-type", "union-
type" ]
    }
  ]
}
```

Appendix A. Revision History

Revision 0.1.3 Mon Jun 3, 2013 **Ghislain Fourny *g@28.io***

Added constraints on $content and $open for objects, to ensure proper object-oriented inheritance. But for the moment, object derivation is limited to the topmost type, so that these constraints are trivially fulfilled.
Fixed typos.
Cleaned up Schema Schema.

Revision 0.1.2 Thu May 30 2013 **Ghislain Fourny *g@28.io***

Annotating an atomic means casting it.
The lexical space of date/dateTime/time was extended to support RFC 2822.
Local types have no prefix.
The URI Qualified Name syntax may also be used to reference types.
$optional is ignored if a $default is provided for a pair.
Added $about field to Schema and Types for free content.
The Input of the Validation and Annotation processes is now a JDM instance (typically freshly parsed).
$layout and $member-types were renamed to $content
The special key $any was removed from object $content. JSONiq constraints can be used instead. Default values can be computed with a JSONiq query.

Revision 0.1.2 Wed May 29 2013 **Ghislain Fourny *g@28.io***

First Working Draft.

Index