A Language for Rules Describing JSON Content
draft-newton-json-content-rules-03

Abstract

This document describes a language useful for documenting the expected content of JSON structures found in specifications using JSON.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 17, 2015.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.
1. Introduction

The goal of this document is to provide a way to document the expected content of data expressed in JSON [RFC4627] format. That is, the primary purpose of this document is to specify a means for one person to communicate with another person the expected nature of a JSON data structure in a method more concise than prose. The programmatic validation of a JSON data structure against content rules is a lesser goal of this document, though such a practice is useful in both the writing of specifications and the communications of programs.

Unlike JSON Schema, this language is not JSON though the syntax described here is "JSON-like" (a comparison with JSON Schema can be found in Appendix A and a "real world" example can be found in Appendix B). A specialized syntax is used to reduce the tedium in reading and writing rules as the complexity describing allowable content is often more involved than most of the actual content. Figure 2 is an example of this language describing the JSON of Figure 1.

Example JSON lifted from RFC 4627

```json
[  
  
  {
    "precision": "zip",
    "Latitude": 37.7668,
    "Longitude": -122.3959,
    "Address": "",
    "City": "SAN FRANCISCO",
    "State": "CA",
    "Zip": "94107",
    "Country": "US"
  },

  {
    "precision": "zip",
    "Latitude": 37.371991,
    "Longitude": -122.026020,
    "Address": "",
    "City": "SUNNYVALE",
    "State": "CA",
    "Zip": "94085",
    "Country": "US"
  }
]  

Figure 1
Rules describing Figure 1

root [
  2*2{
    "precision" : string,
    "Latitude" : float,
    "Longitude" : float,
    "Address" : string,
    "City" : string,
    "State" : string,
    "Zip" : string,
    "Country" : string
  }
]

Figure 2

Depending on need and desired style, an alternate mapping can be used where certain symbols are substituted for words.

Rules describing Figure 1 with an alternate syntax

root ARRAY
  2*2 OBJECT
    MEMBER "precision" VALUE string AND
    MEMBER "Latitude" VALUE float AND
    MEMBER "Longitude" VALUE float AND
    MEMBER "Address" VALUE string AND
    MEMBER "City" VALUE string AND
    MEMBER "State" VALUE string AND
    MEMBER "Zip" VALUE string AND
    MEMBER "Country" VALUE string
  END_OBJECT
END_ARRAY

Figure 3

The JSON Content Rules are of five types:

- value rules
- member rules
- array rules
- object rules
- group rules
Each rule has two components, a rule name and a rule definition. Anywhere in a rule definition where a rule name is allowed, another rule definition may be used.

This is an example of a value rule:

\[ v1 : integer 0..3 \]

It specifies a rule named "v1" that has a definition of ": integer 0..3" (value rule definitions begin with a ':' character). This defines values of type "v1" to be integers in the range 0 to 3 (minimum value of 0, maximum value of 3). Value rules can define the limits of JSON values, such as stating that numbers must fall into a certain range or that strings must be formatted according to certain patterns or standards (i.e. URIs, phone numbers, etc...).

Member rules specify JSON object members. The following example member rule states that the rules name is 'm1' with a value defined by the 'v1' value rule:

\[ m1 "m1name" v1 \]

Since rule names can be substituted by rule definitions, this member rule can also be written as follows:

\[ m1 "m1name" : integer 0..3 \]

Object rules are composed of member rules, since JSON objects are composed of members. Object rules can specify members that are mandatory, optional, and even choices between members. In this example, the rule 'o1' defines an object that must contain a member as defined by member rule 'm1' and optionally a member defined by the rule 'm2':

\[ o1 { m1, ?m2 } \]

Finally, array rules are composed of value and object rules. Like object rules, array rules can specify the cardinality of the contents of an array. The following array rule defines an array that must contain value rule 'v1' and zero or more objects as defined by rule 'o1':

\[ a1 [ v1, *o1 ] \]

Putting it all together, Figure 5 describes the JSON in Figure 4.
Example JSON shamelessly lifted from RFC 4627

```json
{
    "Image": {
        "Width": 800,
        "Height": 600,
        "Title": "View from 15th Floor",
        "Thumbnail": {
            "Url": "http://www.example.com/image/481989943",
            "Height": 125,
            "Width": "100"
        },
        "IDs": [116, 943, 234, 38793]
    }
}
```

Figure 4

Rules describing Figure 4

width_v : integer 0..1280

eight_v : integer 0..1024

```json
width "width" width_v
eight "height" height_v
```

```json
thumbnail "thumbnail" {
    width, height, "Url" : uri
}
```

```json
image "Image" {
    width, height, "Title" : string,
    thumbnail, "IDs" [ *: integer ]
}
```

root { image }

Figure 5

The rules from Figure 5 can be written more compactly (see Figure 6).
Compact rules describing Figure 4

width "width" : integer 0..1280
height "height" : integer 0..1024

root {
  "Image" {
    width, height, "Title" :string,
    "thumbnail" { width, height, "Url" :uri },
    "IDs" [ *:integer ]
  }
}

Figure 6

For completeness, Figure 7 demonstrates the same rules as Figure 6 using the alternate syntax.

Compact, alternative rules describing Figure 4

width MEMBER "width" VALUE integer 0..1280
height MEMBER "height" VALUE integer 0..1024

root OBJECT
  MEMBER "Image" OBJECT
    width AND height AND
    MEMBER "Title" VALUE string AND
    MEMBER "thumbnail" OBJECT
    width, height, MEMBER "Url" VALUE uri
    END_OBJECT AND
    MEMBER "IDs" ARRAY *VALUE integer END_ARRAY
  END_OBJECT
END_OBJECT

Figure 7

2. Lines and Comments

There is no statement terminator and therefore no need for a line continuation syntax. Blank lines are allowed.

Comments are very similar to comments in ABNF [RFC4234]. They start with a semi-colon (‘;’) and continue to the end of the line.
3. Rules

Rules are composed of two parts, a rule name and a rule definition. Rule names allow a rule definition to be referenced easily by a name. With the exception of value rules, rule definitions refer to other rules using the rule names of other appropriate types of rules. Because of this, it is also possible to use a rule definition of the appropriate type where a rule name of that type would be appropriate.

The type of rule to use in a rule definition, either directly or by reference of a name, depends on the type of rule being defined and fall along the structure of allowable JSON grammar:

- Since a member of a JSON object can contain a "primitive value", an array, or an object, member rules can be composed of value rules, array rules, and object rules.
- JSON objects are composed of members, so object rules can only be composed of member rules.
- Finally, as JSON arrays may contain other arrays, objects, and values, array rules may be composed of value rules, object rules, and array rules.

A fifth rule type, group rules, exist to help reference a collection of rules.

Rule names must start with an alphabetic character (a-z,A-Z) and must contain only alphabetic characters, numeric characters, the hyphen character ('-') and the underscore character ('_'). Rule names must not be used more than once.

3.1. Value Rules

Value rules define content for JSON values. JSON allows values to be objects, arrays, numbers, booleans, strings, and null. Arrays and objects are handled by the array and object rules, and the value rules define the rest.

3.1.1. Numbers, Booleans and Null

The rules for booleans and null are the simplest and take the following forms:

rule_name : boolean

rule_name : null
Rules for numbers can specify the number as either an integer or floating point number and may specify a range:

\[
\text{rule_name : integer } n..m
\]
\[
\text{rule_name : float } n..m
\]

where \( n \) is the minimum allowable value of the number and \( m \) is the maximum allowable value of the number. The range doesn’t have to be given, but if it is given either the minimum, maximum, or both are required. If the minimum is not given then the minimum is considered to be the minimum number value possible to represent in JSON. Likewise, if the maximum is not given then the maximum is considered to be the maximum number value possible to represent in JSON.

### 3.1.2. Strings

String values may be specified generically as:

\[
\text{rule_name : string}
\]

However, the content of strings can be narrowed in the following ways:

- **Regular Expression**: A rule can state that a string must match a regular expression by giving the regular expression after the string literal:

  \[
  \text{rule_name : string /regex/}
  \]

- **URIs and URI templates**: A rule can state that a string must be a URI [RFC3986]:

  \[
  \text{rule_name : uri}
  \]

  URIs may be further scoped to a specific URI pattern by prepending a URI template [RFC6570]:

  \[
  \text{rule_name : uri http://{stuff}}
  \]
  \[
  \text{rule_name : uri http://{authority}/{thing1}?q={thing2}}
  \]

When using URI templates, the variable names are ignored for pattern matching, but they should be provided for construction of a valid URI template. Providing the variable names also aids in the description of what is to be matched.
IP Addresses: Narrowing the content of strings down to IP addresses can be done with either the ‘ip4’ (see [RFC1166]) or ‘ip6’ (see [RFC5952]) literals:

   rule_name : ip4
   rule_name : ip6

Domain Names: Fully qualified A-label and U-label domain names can be specified with the ‘fqdn’ and ‘idn’ literals:

   rule_name : fqdn
   rule_name : idn

Dates and Times: Dates and times are specified using the ABNF rules from RFC 3339 [RFC3339] as literals:

   rule_name : date-time
   rule_name : full-date
   rule_name : full-time

Email Addresses: A string can be scoped to the syntax of email addresses using the literal ‘email’:

   rule_name : email

Email addresses must conform to the syntax of RFC 5322 [RFC5322].

Phone Numbers: Strings conforming to E.123 phone number format can be specified as follows:

   rule_name : phone

Base 64: Strings containing base 64 data, as described by RFC 4648 [RFC4648], can be specified as follows:

   rule_name : base64

3.1.3. Enumerations

Enumerations allow a value to be one of the items in an enumerated list of possible values. They take the following form:

   rule_name : < "item1" "item2" "item3" >
Items in the enumerated list may be quoted strings, integer or floating point numbers, or the literals 'true', 'false' or 'null'. The types of the items may be mixed, as the following example demonstrates:

truthy : < 1 true "yes" "Y" >

3.2. Member Rules

Member rules are the simplest of the rules and define members of JSON objects. Member rules follow the format:

\[
\text{rule_name} \ "member_name" \ \text{target_rule_name}
\]

where rule_name is the name of the rule being defined, member_name (in quotes) is the name of the JSON object member, and target_rule_name is a reference to a value rule, array rule, or object rule specifying the allowable content of the JSON object member.

Since rule names in rule definitions may be substituted for rule definitions, member rules may also be written in this form:

\[
\text{rule_name} \ "member_rule" \ \text{target_rule_definition}
\]

The following is an example:

\[
\text{location_uri} \ "locationURI" : \text{uri}
\]

3.3. Object Rules

Object rules define the allowable members of a JSON object. Their rule definitions are composed of member rules and group rules. They take the following form:

\[
\text{rule_name} \ \{ \ \text{member_rule}_1, \ \text{member_rule}_2 \ \}
\]

The following rule example defines an object composed of two member rules:

\[
\text{response} \ \{ \ \text{location_uri}, \ \text{status_code} \ \}
\]

Given the general rule that where a rule name is found a rule definition of the appropriate type may be used, the above example might also be written:

\[
\text{response} \ \{ \ "locationUri" : \text{uri}, \ "statusCode" : \text{integer} \ }
\]
Rules given in the rule definition of an object rule do not imply order. Given the example object rule above both

    { "locationUri" : "http://example.com", "statusCode" : 200 }

and

    { "statusCode" : 200, "locationUri" : "http://example.com" }

are JSON objects that match the rule.

Member rules or member rule definitions may not be repeated in the rule definition of an object rule. However, a member of an object can be marked as optional if the member rule defining it is preceded by the question mark ('?') character. In the following example, the location_uri member is optional while the status_code member is required to be in the defined object:

    response { ?location_uri, status_code }

An object rule can also define the choice between members by placing the forward slash ('/') character between two member rules. In the following example, the object being defined can have either a location_uri member or content_type member and must have a status_code member:

    response { location_uri / content_type, status_code }

Finally, the specification of a member of an object can be conditioned upon the specification of another member of that object by placing the ampersand ('&') character between two member rules. Using this syntax, the member defined by the second rule is only allowed in the object if the member defined by the first rule is given. Or in other words, the appearance of the second member depends upon the appearance of the first member. In the following example, the object defined can have a referrer_uri so long as location_uri is also present:

    response { location_uri & referrer_uri }

3.4. Array Rules

Array rules define the allowable content of JSON arrays. Their rule definitions are composed of value rules, object rules, group rules, and other array rules and have the following form:

    rulename [ target_rule_name_1, target_rule_name_2 ]
The following example defines an array where element 1 is defined by the width_value rule and element 2 is defined by the height_value rule:

    size [ width_value, height_value ]

Unlike object rules, order is implied by the array rule definition. That is, the first rule referenced or defined within an array rule specifies that the first element of the array will match that rule, the second rule given with the array rule specifies that the second element of the array will match that rule, and so on.

Take for example the following array rule definition:

    person [ : string, : integer ]

This JSON array matches the above rule:

    [ "Bob Smurd", 24 ]

while this one does not:

    [ 24, "Bob Smurd" ]

As with object rules, the forward slash character ('/') can be used to indicate a choice between two elements. Take for example the following rules:

    name_value : string
    age_value : integer
    birthdate_value : date-time

    person [ name_value, age_value / birthdate_vale ]

which would validate

    [ "Bob Smurd", 24 ]

or

    [ "Bob Smurd", "1988-04-12T23:20:50.52Z" ]

Repetition of array values may also be specified by preceding a rule with an asterisk ('*') character surrounded by the lower bound and upper bound of the repetition (e.g. "0*1"). The following rules define an array that has between one and three strings:
child_value : string

children [ 1*3 child_value ]

Both the lower bound and the upper bound are optional. If lower bound is not given then it is assumed to be zero. If the upper bound is not given then it is assumed to be infinity. The following example defines an array with an infinite number of child_value defined strings:

children [ * child_value ]

3.5. Group Rules

Unlike the other types of rules, group rules have no direct tie with JSON syntax. Group rules simply group together other rules. They take the form:

rule_name ( target_rule_1, target_rule_2 )

Group rule definitions and any nesting of group rule definitions, must conform to the allowable set of rules of the rule containing them. A group rule referenced inside of an array rule may not contain a member rule since member rules are not allowed in array rules directly. Likewise, a group rule referenced inside an object rule must only contain member rules, and once group rules used in an object rule are fully dereferenced there must be no duplicate member rules as member rules in object rules are required to be unique.

Take for example the following rules:

child_1 "first_child" : string

child_2 "second_child" : string

child_3 "third_child" : string

child_4 "fourth_child" : string

first_two_children ( child_1, child_2 )

second_two_children ( child_3, child_4 )

the_children { first_two_children, second_two_children }

These rules describe a JSON object that might look like this:
Groups can also be used with the choice and dependency syntax in member rules. Here the object can either have first_two_children or second_two_children:

the_children { first_two_children / second_two_children }

and here the object can have second_two_children only if first_two_children are given:

the_children { first_two_children & second_two_children }

Group rules can be used to create object mixins. In the example in Figure 8, both obj1 and obj2 have a members "foo" and "fob" with obj1 having the additional member "bar" and obj2 having the additional member "baz".

mixin_group ( "foo" : integer, "fob" : uri )

obj1 { mixin_group, "bar" : string }

obj2 { mixin_group, "baz" : string }

Figure 8

3.6. Any Value and Any Member

It is possible to specify that a value can be of any type allowable by JSON using the any value rule. This is done with the 'any' literal in a value rule:

rule_name : any

However, unlike other value rules which define primitive data types, this rule defines a value of any kind, either primitive (null, boolean, number, string), object, or array.

Use of the any value rule in arrays can be used with repetition to define arrays that may contain any value:

any_value : any

array_of_any [ *any_value ]

Specifying any object member name in a member rule with the any member rule is done by pre-pending a carat character ('^') to an
empty member name (that is, "^^" signifies any member name). This has the following form:

    rule_name ^"" target_rule_name

As an example, the following defines an object member with any name that has a value that is a string:

    user_data ^"" : string

Usage of the any member rule must still satisfy the criteria that all member names of an object be unique.

Constructing an object member of any name with any type would therefore take the form:

    rule_name ^"" : any

Unlike other types of member rules, it is possible to use repetition with the any member rule in an object rule. The repetition syntax and semantics are the same as the repetition syntax and semantics of repetition with array rules. The following example rules define an object that may contain any number of members where each member may have any value.

    any_member ^"" : any

    object_of_anything { *any_member }

Use of the repetition of any member rules must satisfy the criteria that all member names of an object be unique.

3.7. A Root Rule

In some contexts it is necessary that there be a rule that defines the outer most JSON object or array, or if thought of as an inverted object tree the structure at the very top. If in a collection of rules there is no rule explicitly specified for this purpose and a rule named "root" is given, it can be assumed to be the outer most JSON structure or the root of an object/array tree. If a rule is explicitly specified other than "root" and there exists a rule named "root", that rule name holds no special meaning.

4. Directives

Directives change the interpretation of a collection of rules. They begin with a hash character ("#") and are terminated by the end of a line. They take the following form:
# directive_name

Directives may have other qualifiers after the directive name. They may appear intermixed with rules but cannot appear in a rule definition.

4.1. ignore-unknown-members

This directive specifies that any member of any object which has not been specified should be ignored. Ignored object members may have a value of any type. This directive cannot be used in any collection of rules that has an any member rule.

4.2. language-compatible-members

This directive specifies that every member name of every object, either explicitly defined or specified via an any member rule or the ignore-unknown-members directive must be a name compatible with programming languages. The intent is to specify object member names that may be promoted to first-order object attributes or methods in an API. The following ABNF describes the restrictions upon the member names:

ABNF for programming language compatible JSON names

```plaintext
name = ALPHA *( ALPHA / DIGIT / "_" )
```

Figure 9

4.3. all-members-optional

This directive specifies that every member of every object is not required. This directive effectively pre-pends a ‘?’ to every member rule in every object rule.

4.4. include

This directive specifies that another collection of rules should be evaluated before the rules following this directive. This directive must be qualified with a quoted string describing the collection of rules to evaluate.

# include "Section 3 of RFC XXXX"

The quoted string may be optionally followed by a URI:
5. Alternate Syntax

The syntax given in the sections above does require specific knowledge of JSON content rules and may not be appropriate for describing simpler JSON structures or may not be desired for describing JSON to audiences unfamiliar with JSON content rules. In other words, choice of syntax is a matter of style and taste and can vary depending on conditions.

Therefore this section describes a syntax mapping where symbols are substituted for words, thus providing a more readable rule set to some audiences. Because it is a simple mapping between symbols and words, the structure of the rules does not change and the syntaxes may be interchanged.

The syntax map is as follows:

- : (colon) maps to the word VALUE
- { (left curly bracket) maps to OBJECT
- } (right curly bracket) maps to END_OBJECT
- , (comma) maps to AND
- / (forward slash) maps to OR
- & (ampersand) maps to DEPENDS
- [ (left square bracket) maps to ARRAY
- ] (right square bracket) maps to END_ARRAY
- ( (left parenthesis) maps to GROUP
- ) (right parenthesis) maps to END_GROUP
- < (less than symbol) maps to ENUM
- > (greater than symbol) maps to END_ENUM

Because member names in member rules must be quoted strings, there is no need to map the quotation marks. However the word MEMBER can precede the quoted string in member rules.
The carat symbol used in any rules and the asterisk symbol used for specifying cardinality do not have a mapping as there are probably no good words to describe their usage. Additionally, directives do not have mappings as they are descriptive.

The following are examples of the symbolic syntax and the alternate word syntax:

Value Rules

```
birthday : date-time

birthday VALUE date-time
```

```
valid_widths_rule : < 640 960 1024 >

valid_widths_rule VALUE ENUM 640 960 1024 END_ENUM
```

Member Rules with Values

```
width_rule "width" : integer 0..1280

width_rule MEMBER "width" VALUE integer 0..1280
```

Array Rules with Values

```
person_array [ : string, : integer ]

person_array ARRAY VALUE string AND VALUE integer END_ARRAY
```

Object Rules with Members and Values

```
response { "locationUri" : uri, "statusCode" : integer }

response OBJECT MEMBER "locationUri" VALUE uri AND MEMBER "statusCode" VALUE integer END_OBJECT
```

Group Rules

```
first_two_children ( child_1, child_2 )

first_two_children GROUP child_1 AND child_2 END_GROUP
```

6. Normative References

Appendix A. Comparison with JSON Schema

This section compares this specification, JSON Content Rules, with JSON Schema using examples.

A.1. Example 1 from RFC 4627
Example JSON lifted from RFC 4627

```
[
  {
    "precision": "zip",
    "Latitude": 37.7668,
    "Longitude": -122.3959,
    "Address": "",
    "City": "SAN FRANCISCO",
    "State": "CA",
    "Zip": "94107",
    "Country": "US"
  },
  {
    "precision": "zip",
    "Latitude": 37.371991,
    "Longitude": -122.026020,
    "Address": "",
    "City": "SUNNYVALE",
    "State": "CA",
    "Zip": "94085",
    "Country": "US"
  }
]
```

JSON Content Rules

```
root [2*2{
  "precision" : string,
  "Latitude" : float,
  "Longitude" : float,
  "Address" : string,
  "City" : string,
  "State" : string,
  "Zip" : string,
  "Country" : string
}
]
```
JSON Schema

{
    "type": "array",
    "items": [
    {
        "type": "object",
        "properties": {
            "precision": { "type": "string", "required": "true" },
            "Latitude": { "type": "number", "required": "true" },
            "Longitude": { "type": "number", "required": "true" },
            "Address": { "type": "string", "required": "true" },
            "City": { "type": "string", "required": "true" },
            "State": { "type": "string", "required": "true" },
            "Zip": { "type": "string", "required": "true" },
            "Country": { "type": "string", "required": "true" }
        }
    },
    {
        "minItems": 2,
        "maxItems": 2
    }
}

A.2. Example 2 from RFC 4627

Example JSON shamelessly lifted from RFC 4627

{
    "Image": {
        "Width": 800,
        "Height": 600,
        "Title": "View from 15th Floor",
        "Thumbnail": {
            "Url": "http://www.example.com/image/481989943",
            "Height": 125,
            "Width": "100"
        },
        "IDs": [116, 943, 234, 38793]
    }
}
JSON Content Rules

width "width" : integer 0..1280
height "height" : integer 0..1024

root {
  "Image" {
    width, height, "Title" : string,
    "thumbnail" { width, height, "Url" : uri },
    "IDs" [ *:integer ]
  }
}

JSON Schema

{  
  "type" : "object",
  "properties" : {
    "Image" : {  
      "type" : "object",
      "properties" : {  
        "Width" : {  
          "type" : "integer",
          "minimum" : 0,
          "maximum" : 1280,
          "required" : "true"
        }
      },
      "Height" : {  
        "type" : "integer",
        "minimum" : 0,
        "maximum" : 1024,
        "required" : "true"
      }
    },
    "Title" : {  "type": "string" },
    "Thumbnail" : {  
      "type" : "object",
      "properties" : {  
        "Url" : {  
          "type" : "string",
          "format" : "uri",
          "required" : "true"
        },
        "Width" : {  
          "type" : "integer",
          "minimum" : 0,
          "maximum" : 1280,
          "required" : "true"
        }
      }
    }
  }
}
"Height" : {
    "type" : "integer",
    "minimum" : 0,
    "maximum" : 1280,
    "required" : "true"
}

"IDs" : {
    "type": "array",
    "items": [{ "type" : "integer" }],
    "required" : "true"
}

Appendix B. A "Real World" Example

The following example is taken from draft-ietf-weirds-json-response-00. It describes the entity object (Section 4), the nameserver object (Section 5) and many of the other sub-structures used in objects defined in other sections of that draft.

JSON Content Rules for nameserver and entity from draft-ietf-weirds-json-response

# all-members-optional
# ignore-unknown-members
# language-compatible-members

; the nameserver object
; models nameserver host information
; this often referred to as 'host' object too
nameserver {

    ; the host name of the name server
    "name" : fqdn,

    ; the ip addresses of the nameserver
    "ipAddresses" [ *( :ip4 / :ip6 ) ],

    common
}

; the entity object
; This object object represents the information of organizations,
; corporations, governments, non-profits, clubs, individual persons, 
; and informal groups of people.
entity {

; the names by which the entity is commonly known
"names" [ *:string ],

; the roles this entity has with any containing object
"roles" [ *:string ],

; the place where the person, org, etc... receives postal mail
; THIS IS NOT LOCATION
"postalAddress" [ *:string ],

; electronic mailboxes where the person, org, etc... 
; receives messages
"emails" [ *:email ],

; phones where the person, org, etc... receives
; telephonic communication
"phones" {
  "office" [ *:phone ], ; office phones
  "fax" [ *:phone ], ; facsimile machines
  "mobile" [ *:phone ]; cell phones and the like
},

common
}

; The members "handle", "status", "remarks", "uris", "port43",
; "sponsoredBy", "resoldBy", "registrationBy", "registrationDate",
; "lastChangedDate", and "lastChangedBy" are used in many objects
common (

; a registry-unique identifier
"handle" : string,

; an array of status values
"status" [ *:string ],

; an array of strings, each containing comments about the object
"remarks" [ *:string ].

; an array of uri objects
; "type" refers to the application of the URI
; "uri" is the uri
"uris" [ 
  *{ "type" : string, "uri" : uri }


Appendix C. Design Notes

C.1. Member Uniqueness

JSON does not disallow non-unique object member names (in other words, it allows non-unique object member names) but strongly advises against the use of non-unique object member names. Many JSON implementations use hash-indexed maps to represent JSON objects, where the object’s member names are the key of the hash index. Non-uniqueness would break such implementations or result in the value of the last member given overwriting the value of all previous members of the same name.

Therefore, allowing non-unique object member names would be bad practice. For this reason, this specification does not accommodate the need for non-unique object member names.
C.2. Member Order

JSON gives awkward guidance regarding ordering of object member names. However, many JSON implementations use hash-indexed maps to represent JSON objects, where the object’s member names are the key of the hash index. Though it is possible, usually these maps have no explicit order as the only index is the hash.

Therefore, this specification does not provide a means to imply order of object member names.

C.3. Group Syntax for Arrays and Objects

It is possible to create a separate group syntax for array rules vs object rules, since allowable group rule content is determined by the containing rule. For instance, while the syntax for groups in objects could have been "( blah blah )", syntax for groups in arrays could have been "< blah blah >". That may be more distinctive and allow the formal syntax parser to handle rule content validity, but the added extra syntax appeared to hurt readability. There is only so many enclosure characters a person should reasonably be required to know, and adding yet another did not seem prudent.

C.4. Inspiration

The original approach to this problem was to find a concise way to describe JSON data structures; to do for JSON what RelaxNG compact syntax does for XML. The syntax itself hopefully has a JSON-ness or a JSON feel to it. And a good bit of inspiration came from ABNF.

C.5. Changelog

From -00 to -01

1. Added ABNF. Thanks Byron Ellacott.
2. Added section about root rules.
3. Other minor edits.

From -01 to -02

1. Other minor edits.
2. Added the Possible Future Changes section.
3. Mostly a keep-alive version.
From -02 to -03

1. Removed formal syntax (ABNF) until such time as the features are nailed down. It will appear in a future draft.

2. Took out the option for multiple email conformance levels as everything should be conformant to RFC 5322.

3. URIs conformance can now either be just ‘uri’ or match a URI template (suggestion from Andrew Biggs).

4. Added enumerated values based on a suggestion from Paul Jones.

5. Added a directive for including other collections of rules.

6. Added an example of object mixins using groups thanks to a discussion with Andrew Biggs.

7. Added an alternate syntax mapping.

Author’s Address

Andrew Lee Newton
American Registry for Internet Numbers
3635 Concorde Parkway
Chantilly, VA  20151
US

Email: andy@arin.net
URI:  http://www.arin.net