Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: May 14, 2015

A. Boronine, Ed.

November 10, 2014

Minimal JSON Type System
draft-boronine-teleport-00

Abstract

Teleport is a minimal type system designed as an extension of JSON. It comes with 10 types sufficient for basic use, but it defines a pattern for extending it with arbitrary types. Teleport's type definitions are JSON values, for example, an array of strings is defined as {"Array": "String"}.

Teleport implementations can be used for data serialization, input validation, for documenting JSON APIs and for building API clients.

This document provides the mathematical basis of Teleport and can be used for implementing libraries.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

In Teleport, a type is a relation between a type definition and a value space. For example:

```
t("Integer") = {0, -1,  1, -2,  2, -3,  3, ...}
```

Here "Integer" is a type definition and t("Integer") is the set of all values this type can take. The t function is used to represent this relationship.

Because Teleport is based on JSON, all value spaces are sets of JSON values. More interestingly, type definitions are JSON values too, which makes it trivial to share them with other programs.

Teleport's design goals is to be a natural extension of JSON, be extremely lightweight, and extendable not only with rich types but with high-level type system concepts.

2. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

The terms "JSON", "JSON text", "JSON value", "member", "element", "object", "array", "number", "string", "boolean", "true", "false", and "null" in this document are to be interpreted as defined in RFC 4627 [RFC4627].

2.1. Syntax

Throughout this document, an extended JSON syntax is used. Unquoted strings are symbols representing JSON values, sets and functions. Also, the following set theory syntax is used:

a :: A   Set A contains element a.

D -> C   The set of functions that map values from set D to values from set C.

3. Type Patterns

Types defined simply by a string, like "Integer" above, are called concrete. Teleport ships with 7 concrete types.

A generic type maps a set of schemas to a set of value spaces. Each pair in the mapping is called an instance. For example, {"Array": "Integer"} is an instance of the Array type.

Three generic types are provided: Array, Map and Struct. Their precise definition is provided in the following sections, but these examples should be enough to understand how they work:

```
["foo", "bar"]       :: t({"Array": "String"})

{"a": 1, "b": 2}     :: t({"Map": "Integer"})

{"name": "Alexei"}   :: t({"Struct": {
                             "required": {"name": "String"},
                             "optional": {"age": "Integer"}})
```

4. JSON Schemas

"Schema", one of the build-in concrete types, is defined as the set of all known type definitions. This is made possible by the fact

that type definitions are JSON values. The Schema type is useful to specify APIs. For example, to describe a function you can use this:

```
t({"Struct": {
     "required": {
        "input": "Schema",
        "output": "Schema"}}}
```

5. Mathematical Basis

The set of all JSON values is called V. A subset of V is called a value space and the set of all value spaces is called S.

```
V = {null, true, false, 0, 1, 2, 3, 4, ...}

S = {{}, {null}, {null, true}, {null, ...}, ...}
```

There is a certain function t that maps JSON values to value spaces.

```
t :: (V -> S)
```

5.1. Concrete Types

x is of concrete type c if and only if

1. c is a string

2. x :: t(c).

5.2. Generic Types

x is of generic type g if and only if

1. g is a string

2. x :: t({g: p}) for some p

6. Built-in Concrete Types

t("JSON")	JSON
t("Schema")	The set of all type definitions, including all strings representing concrete types as well as every instance of every generic type.
t("Integer")	All numbers that don't have a fractional or exponent part.

t("Float") All numbers that have a fractional or exponent part.

t("String") All strings.

t("Boolean") All booleans.

t("DateTime") All strings that are valid according to ISO 8601 [ISO.8601.1988].

7. Built-in Generic Types

x :: t({"Array": p}) if and only if

x is an array

e :: t(p) for every element e in x

x :: t({"Map": p}) if and only if

x is an object

v :: t(p) for every pair (k, v) in x

x :: t({"Struct": p}) if and only if

p is an object with two members: required and optional. Both are objects and their names are disjoint, that is, they don't have a pair of members with the same name.

x is an object. The name of every member of p.required is also the name of a member of x.

For every pair (k, v) in x, there is a pair (k, s) in either p.required or p.optional such that v :: t(s).

8. IANA Considerations

This memo includes no request to IANA.

9. Security Considerations

All drafts are required to have a security considerations section. See RFC 3552 [RFC3552] for a guide.

10. References

10.1. Normative References

[ISO.8601.1988]
International Organization for Standardization, "Data elements and interchange formats - Information interchange - Representation of dates and times", ISO Standard 8601, June 1988.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

10.2. Informative References

[RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, July 2003.

Appendix A. Mailing List

Comments are solicited and should be addressed to the working group's mailing list at teleport-json@googlegroups.com and/or the author.

Author's Address

Alexei Boronine (editor)

Email: alexei@boronine.com