Abstract

Teleport is a minimal type system designed as an extension of JSON. It comes with 10 types sufficient for basic use and provides two patterns for extending it with new 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 for Teleport and can be used for implementing libraries.

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 March 30, 2016.

Copyright Notice

Copyright (c) 2015 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
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 [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 [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:

\[ a :: A \quad \text{Set A contains element a.} \]
\[ D \rightarrow C \quad \text{The set of functions that map values from set D to values from set C.} \]

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 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:

```json
    t({"Struct": {
        "optional": {},
        "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 $S$.

$$V = \{\text{null, true, false, 0, 1, 2, 3, 4, ...}\}$$

$$S = \{\{\}, \{\text{null}\}, \{\text{null, true}\}, \{\text{null, ...}\}, ...\}$$

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

$$t : (V \rightarrow S)$$

This document does not give a full definition of the $t$ function, it merely provides some instances of its inputs and outputs. Expanding the definition of the $t$ function is the basis for extending Teleport.

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

JSON $t("JSON")$ is the set of all JSON values. This type can be used as a wildcard for type-checking or as a noop for composable serialization.
Schema    t("Schema") is the set of all type definitions, including all strings representing concrete types as well as every instance of every generic type.

Decimal  t("Decimal") is the set of all numbers. This type represents real numbers and arbitrary-precision approximations of real numbers.

Integer   t("Integer") is the set of all numbers that don’t have a fractional or exponent part.

String    t("String") is the set of all strings. Note that JSON strings are sequences of Unicode characters.

Boolean   t("Boolean") is a set containing the JSON values true and false.

DateTime  t("DateTime") is the set of all strings that are valid according to RFC 3339 [RFC3339]. This type represents timestamps with optional timezone data.

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. If p has the key "required", p.required must be an object, similarly if p has the key "optional", p.optional must be an object. If p has both, the names in these two objects must be disjoint, that is, they cannot 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).
NOTE: the definition of Struct implies that its parameter p can contain arbitrary metadata in the form of other object members.

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


10.2. Informative References


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