Stream:	Independent Submission
RFC:	8927
Category:	Experimental
Published:	October 2020
ISSN:	2070-1721
Author:	U. Carion
	Segment

RFC 8927
JSON Type Definition

Abstract

This document proposes a format, called JSON Type Definition (JTD), for describing the shape of JavaScript Object Notation (JSON) messages. Its main goals are to enable code generation from schemas as well as portable validation with standardized error indicators. To this end, JTD is intentionally limited to be no more expressive than the type systems of mainstream programming languages. This intentional limitation, as well as the decision to make JTD schemas be JSON documents, makes tooling atop of JTD easier to build.

This document does not have IETF consensus and is presented here to facilitate experimentation with the concept of JTD.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document describes a schema language for JSON [RFC8259] called JSON Type Definition (JTD).

There exist many options for describing JSON data. JTD's niche is to focus on enabling code generation from schemas; to this end, JTD's expressiveness is intentionally limited to be no more powerful than what can be expressed in the type systems of mainstream programming languages.

The goals of JTD are to:

- Provide an unambiguous description of the overall structure of a JSON document.
- Be able to describe common JSON data types and structures (that is, the data types and structures necessary to support most JSON documents and that are widely understood in an interoperable way by JSON implementations).
- Provide a single format that is readable and editable by both humans and machines and that can be embedded within other JSON documents. This makes JTD a convenient format for tooling to accept as input or produce as output.
- Enable code generation from JTD schemas. JTD schemas are meant to be easy to convert into data structures idiomatic to mainstream programming languages.

- Provide a standardized format for error indicators when data does not conform with a schema.

JTD is intentionally designed as a rather minimal schema language. Thus, although JTD can describe some categories of JSON, it is not able to describe its own structure; this document uses Concise Data Definition Language (CDDL) [RFC8610] to describe JTD's syntax. By keeping the expressiveness of the schema language minimal, JTD makes code generation and standardized error indicators easier to implement.

Examples in this document use constructs from the C++ programming language. These examples are provided to aid the reader in understanding the principles of JTD but are not limiting in any way.

JTD's feature set is designed to represent common patterns in JSON-using applications, while still having a clear correspondence to programming languages in widespread use. Thus, JTD supports:

- Signed and unsigned 8-, 16-, and 32-bit integers. A tool that converts JTD schemas into code can use `int8_t`, `uint8_t`, `int16_t`, etc., or their equivalents in the target language, to represent these JTD types.
- A distinction between `float32` and `float64`. Code generators can use `float` and `double`, or their equivalents, for these JTD types.
- A "properties" form of JSON objects, corresponding to some sort of struct or record. The "properties" form of JSON objects is akin to a C++ `struct`.
- A "values" form of JSON objects, corresponding to some sort of dictionary or associative array. The "values" form of JSON objects is akin to a C++ `std::map`.
- A "discriminator" form of JSON objects, corresponding to a discriminated (or "tagged") union. The "discriminator" form of JSON objects is akin to a C++ `std::variant`.

The principle of common patterns in JSON is why JTD does not support 64-bit integers, as these are usually transmitted over JSON in non-interoperable (i.e., ignoring the recommendations in Section 2.2 of [RFC7493]) or mutually inconsistent ways. Appendix A.1 further elaborates on why JTD does not support 64-bit integers.

The principle of clear correspondence to common programming languages is why JTD does not support, for example, a data type for integers up to 2**53-1.

It is expected that for many use cases, a schema language of JTD's expressiveness is sufficient. Where a more expressive language is required, alternatives exist in CDDL and others.

This document does not have IETF consensus and is presented here to facilitate experimentation with the concept of JTD. The purpose of the experiment is to gain experience with JTD and to possibly revise this work accordingly. If JTD is determined to be a valuable and popular approach, it may be taken to the IETF for further discussion and revision.

This document has the following structure. Section 2 defines the syntax of JTD. Section 3 describes the semantics of JTD; this includes determining whether some data satisfies a schema and what error indicators should be produced when the data is unsatisfactory. Appendix A discusses why certain features are omitted from JTD. Appendix B presents various JTD schemas and their CDDL equivalents.

1.1. Terminology

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The term "JSON Pointer", when it appears in this document, is to be understood as it is defined in [RFC6901].

The terms "object", "member", "array", "number", "name", and "string" in this document are to be interpreted as described in [RFC8259].

The term "instance", when it appears in this document, refers to a JSON value being validated against a JTD schema. This value can be an entire JSON document, or it can be a value embedded within a JSON document.

1.2. Scope of Experiment

JTD is an experiment. Participation in this experiment consists of using JTD to validate or document interchanged JSON messages or building tooling atop of JTD. Feedback on the results of this experiment may be emailed to the author. Participants in this experiment are anticipated to mostly be nodes that provide or consume JSON-based APIs.

Nodes know if they are participating in the experiment if they are validating JSON messages against a JTD schema or if they are relying on another node to do so. Nodes are also participating in the experiment if they are running code generated from a JTD schema.

The risk of this experiment "escaping" takes the form of a JTD-supporting node expecting another node, which lacks such support, to validate messages against some JTD schema. In such a case, the outcome will likely be that the nodes fail to interchange information correctly.

This experiment will be deemed successful when JTD has been implemented by multiple independent parties and these parties successfully use JTD to facilitate information interchange within their internal systems or between systems operated by independent parties.

If this experiment is deemed successful, and JTD is determined to be a valuable and popular approach, it may be taken to the IETF for further discussion and revision. One possible outcome of this discussion and revision could be that a working group produces a Standards Track specification of JTD.

Some implementations of JTD, as well as code generators and other tooling related to JTD, are available at <https://github.com/jsontypedef>.

2. Syntax

This section describes when a JSON document is a correct JTD schema. Because Concise Data Definition Language (CDDL) is well suited to the task of defining complex JSON formats, such as JTD schemas, this section uses CDDL to describe the format of JTD schemas.

JTD schemas may recursively contain other schemas. In this document, a "root schema" is one that is not contained within another schema, i.e., it is "top level".

A JTD schema is a JSON object taking on an appropriate form. JTD schemas may contain "additional data", discussed in Section 2.3. Root JTD schemas may optionally contain definitions (a mapping from names to schemas).

A correct root JTD schema **MUST** match the `root-schema` CDDL rule described in this section. A correct non-root JTD schema **MUST** match the `schema` CDDL rule described in this section.

```
; root-schema is identical to schema, but additionally allows for
; definitions.
;
; definitions are prohibited from appearing on non-root schemas.
root-schema = {
  ? definitions: { * tstr => { schema}},
  schema,
}
; schema is the main CDDL rule defining a JTD schema.
;
; All JTD schemas are JSON objects taking on one of eight forms
; listed here.
schema = (
  ref //
  type //
  enum //
  elements //
  properties //
  values //
  discriminator //
  empty //
)
; shared is a CDDL rule containing properties that all eight schema
; forms share.
shared = (
  ? metadata: { * tstr => any },
  ? nullable: bool,
)
; empty describes the "empty" schema form.
empty = shared
; ref describes the "ref" schema form.
;
; There are additional constraints on this form that cannot be
; expressed in CDDL. Section 2.2.2 describes these additional
; constraints in detail.
ref = ( ref: tstr, shared )
; type describes the "type" schema form.
type = (
  type: "boolean"
    / "float32"
    / "float64"
    / "int8"
    / "uint8"
    / "int16"
    / "uint16"
    / "int32"
    / "uint32"
    / "string"
    / "timestamp",
  shared,
)
; enum describes the "enum" schema form.
;
; There are additional constraints on this form that cannot be
; expressed in CDDL. Section 2.2.4 describes these additional
; constraints in detail.
enum = ( enum: [+ tstr], shared )
```

```
; elements describes the "elements" schema form.
elements = ( elements: { schema }, shared )
; properties describes the "properties" schema form.
;
; This CDDL rule is defined so that a schema of the "properties" form
; may omit a member named "properties" or a member named
; "optionalProperties", but not both.
;
; There are additional constraints on this form that cannot be
; expressed in CDDL. Section 2.2.6 describes these additional
; constraints in detail.
properties = (with-properties // with-optional-properties)
with-properties = (
  properties: { * tstr => { schema }},
  ? optionalProperties: { * tstr => { schema }},
  ? additionalProperties: bool,
  shared,
)
with-optional-properties = (
  ? properties: { * tstr => { schema }},
  optionalProperties: { * tstr => { schema }},
  ? additionalProperties: bool,
  shared,
)
; values describes the "values" schema form.
values = ( values: { schema }, shared )
; discriminator describes the "discriminator" schema form.
;
; There are additional constraints on this form that cannot be
; expressed in CDDL. Section 2.2.8 describes these additional
; constraints in detail.
discriminator = (
  discriminator: tstr,
  ; Note well: this rule is defined in terms of the "properties"
  ; CDDL rule, not the "schema" CDDL rule.
  mapping: { * tstr => { properties } }
  shared,
)
```

Figure 1: CDDL Definition of a Schema

The remainder of this section will describe constraints on JTD schemas that cannot be expressed in CDDL. It will also provide examples of valid and invalid JTD schemas.

2.1. Root vs. Non-root Schemas

The `root-schema` rule in Figure 1 permits a member named `definitions`, but the `schema` rule does not permit for such a member. This means that only root (i.e., "top-level") JTD schemas can have a `definitions` object, and subschemas may not.

Thus,

```
   { "definitions": {} }
```

is a correct JTD schema, but

```
{
  "definitions": {
    "foo": {
      "definitions": {}
    }
  }
}
```

is not, because subschemas (such as the object at `/definitions/foo`) must not have a member named `definitions`.

2.2. Forms

JTD schemas (i.e., JSON objects satisfying the `schema` CDDL rule in Figure 1) must take on one of eight forms. These forms are defined so as to be mutually exclusive; a schema cannot satisfy multiple forms at once.

2.2.1. Empty

The `empty` form is defined by the `empty` CDDL rule in Figure 1. The semantics of the `empty` form are described in Section 3.3.1.

Despite the name "empty", schemas of the `empty` form are not necessarily empty JSON objects. Like schemas of any of the eight forms, schemas of the `empty` form may contain members named `nullable` (whose value must be `true` or `false`) or `metadata` (whose value must be an object) or both.

Thus,

```
{}
```

and

```
{ "nullable": true }
```

and

```
{ "nullable": true, "metadata": { "foo": "bar" }}
```

are correct JTD schemas of the empty form, but

```
{ "nullable": "foo" }
```

is not, because the value of the member named `nullable` must be `true` or `false`.

2.2.2. Ref

The `ref` form is defined by the `ref` CDDL rule in Figure 1. The semantics of the `ref` form are described in Section 3.3.2.

For a schema of the `ref` form to be correct, the value of the member named `ref` must refer to one of the definitions found at the root level of the schema it appears in. More formally, for a schema *S* of the `ref` form:

- Let *B* be the root schema containing the schema or the schema itself if it is a root schema.
- Let *R* be the value of the member of *S* with the name `ref`.

If the schema is correct, then *B* **MUST** have a member *D* with the name `definitions`, and *D* **MUST** contain a member whose name equals *R*.

Thus,

```
   {
     "definitions": {
       "coordinates": {
         "properties": {
           "lat": { "type": "float32" },
           "lng": { "type": "float32" }
         }
       }
     },
     "properties": {
       "user_location": { "ref": "coordinates" },
       "server_location": { "ref": "coordinates" }
     }
   }
```

is a correct JTD schema and demonstrates the point of the `ref` form: to avoid redefining the same thing twice. However,

```
   { "ref": "foo" }
```

is not a correct JTD schema, as there are no top-level `definitions`, and so the `ref` form cannot be correct. Similarly,

```
   { "definitions": { "foo": {}}, "ref": "bar" }
```

is not a correct JTD schema, as there is no member named `bar` in the top-level `definitions`.

2.2.3. Type

The `type` form is defined by the `type` CDDL rule in Figure 1. The semantics of the `type` form are described in Section 3.3.3.

As an example of a correct JTD schema of the `type` form,

```
   { "type": "uint8" }
```

is a correct JTD schema, whereas

```
   { "type": true }
```

and

```
   { "type": "foo" }
```

are not correct schemas, as neither `true` nor the JSON string `foo` are in the list of permitted values of the `type` member described in the `type` CDDL rule in Figure 1.

2.2.4. Enum

The `enum` form is defined by the `enum` CDDL rule in Figure 1. The semantics of the `enum` form are described in Section 3.3.4.

For a schema of the `enum` form to be correct, the value of the member named `enum` must be a nonempty array of strings, and that array must not contain duplicate values. More formally, for a schema *S* of the `enum` form:

- Let *E* be the value of the member of *S* with name `enum`.

If the schema is correct, then there **MUST NOT** exist any pair of elements of *E* that encode equal string values, where string equality is defined as in Section 8.3 of [RFC8259].

Thus,

```
   { "enum": [] }
```

is not a correct JTD schema, as the value of the member named `enum` must be nonempty, and

```
   { "enum": ["a\\b", "a\u005Cb"] }
```

is not a correct JTD schema, as

```
"a\\b"
```

and

```
"a\u005Cb"
```

encode strings that are equal by the definition of string equality given in Section 8.3 of [RFC8259]. By contrast,

```
{ "enum": ["PENDING", "IN_PROGRESS", "DONE" ]}
```

is an example of a correct JTD schema of the `enum` form.

2.2.5. Elements

The `elements` form is defined by the `elements` CDDL rule in Figure 1. The semantics of the `elements` form are described in Section 3.3.5.

As an example of a correct JTD schema of the `elements` form,

```
{ "elements": { "type": "uint8" }}
```

is a correct JTD schema, whereas

```
{ "elements": true }
```

and

```
{ "elements": { "type": "foo" } }
```

are not correct schemas, as neither

```
true
```

nor

```
{ "type": "foo" }
```

are correct JTD schemas, and the value of the member named `elements` must be a correct JTD schema.

2.2.6. Properties

The `properties` form is defined by the `properties` CDDL rule in Figure 1. The semantics of the `properties` form are described in Section 3.3.6.

For a schema of the `properties` form to be correct, properties must either be required (i.e., in `properties`) or optional (i.e., in `optionalProperties`), but not both.

More formally, if a schema has both a member named `properties` (with value *P*) and another member named `optionalProperties` (with value *O*), then *O* and *P* **MUST NOT** have any member names in common; that is, no member of *P* may have a name equal to the name of any member of *O*, under the definition of string equality given in Section 8.3 of [RFC8259].

Thus,

```
   {
     "properties": { "confusing": {} },
     "optionalProperties": { "confusing": {} }
   }
```

is not a correct JTD schema, as `confusing` appears in both `properties` and `optionalProperties`. By contrast,

```
   {
     "properties": {
       "users": {
         "elements": {
           "properties": {
             "id": { "type": "string" },
             "name": { "type": "string" },
             "create_time": { "type": "timestamp" }
           },
           "optionalProperties": {
             "delete_time": { "type": "timestamp" }
           }
         }
       },
       "next_page_token": { "type": "string" }
     }
   }
```

is a correct JTD schema of the `properties` form, describing a paginated list of users and demonstrating the recursive nature of the syntax of JTD schemas.

2.2.7. Values

The `values` form is defined by the `values` CDDL rule in Figure 1. The semantics of the `values` form are described in Section 3.3.7.

As an example of a correct JTD schema of the `values` form,

```
   { "values": { "type": "uint8" }}
```

is a correct JTD schema, whereas

```
   { "values": true }
```

and

```
   { "values": { "type": "foo" } }
```

are not correct schemas, as neither

```
   true
```

nor

```
   { "type": "foo" }
```

are correct JTD schemas, and the value of the member named `values` must be a correct JTD schema.

2.2.8. Discriminator

The `discriminator` form is defined by the `discriminator` CDDL rule in Figure 1. The semantics of the `discriminator` form are described in Section 3.3.8. Understanding the semantics of the `discriminator` form will likely aid the reader in understanding why this section provides constraints on the `discriminator` form beyond those in Figure 1.

To prevent ambiguous or unsatisfiable constraints on the `discriminator` property of a tagged union, an additional constraint on schemas of the `discriminator` form exists. For schemas of the discriminator form:

- Let *D* be the member of the schema with the name `discriminator`.
- Let *M* be the member of the schema with the name `mapping`.

If the schema is correct, then all member values *S* of *M* will be schemas of the "properties" form. For each *S*:

- If *S* has a member *N* whose name equals `nullable`, *N*'s value **MUST NOT** be the JSON primitive value `true`.

- For each member *P* of *S* whose name equals `properties` or `optionalProperties`, *P*'s value, which must be an object, **MUST NOT** contain any members whose name equals *D*'s value.

Thus,

```
   {
     "discriminator": "event_type",
     "mapping": {
       "can_the_object_be_null_or_not?": {
         "nullable": true,
         "properties": { "foo": { "type": "string" } }}
       }
     }
   }
```

is an incorrect schema, as a member of `mapping` has a member named `nullable` whose value is `true`. This would suggest that the instance may be null. Yet, the top-level schema lacks such a `nullable` set to `true`, which would suggest that the instance in fact cannot be null. If this were a correct JTD schema, it would be unclear which piece of information takes precedence.

JTD handles such possible ambiguity by disallowing, at the syntactic level, the possibility of contradictory specifications of whether an instance described by a schema of the `discriminator` form may be null. The schemas in a discriminator `mapping` cannot have `nullable` set to `true`; only the discriminator itself can use `nullable` in this way.

It also follows that

```
   {
     "discriminator": "event_type",
     "mapping": {
       "is_event_type_a_string_or_a_float32?": {
         "properties": { "event_type": { "type": "float32" }}
       }
     }
   }
```

and

```
   {
     "discriminator": "event_type",
     "mapping": {
       "is_event_type_a_string_or_an_optional_float32?": {
         "optionalProperties": { "event_type": { "type": "float32" }}
       }
     }
   }
```

are incorrect schemas, as `event_type` is both the value of `discriminator` and a member name in one of the `mapping` member `properties` or `optionalProperties`. This is ambiguous, because ordinarily the `discriminator` keyword would indicate that `event_type` is expected to be a string, but another part of the schema specifies that `event_type` is expected to be a number.

JTD handles such possible ambiguity by disallowing, at the syntactic level, the possibility of contradictory specifications of discriminator "tags". Discriminator "tags" cannot be redefined in other parts of the schema.

By contrast,

```
   {
     "tag": "event_type",
     "mapping": {
       "account_deleted": {
         "properties": {
           "account_id": { "type": "string" }
         }
       },
       "account_payment_plan_changed": {
         "properties": {
           "account_id": { "type": "string" },
           "payment_plan": { "enum": ["FREE", "PAID"] }
         },
         "optionalProperties": {
           "upgraded_by": { "type": "string" }
         }
       }
     }
   }
```

is a correct schema, describing a pattern of data common in JSON-based messaging systems. Section 3.3.8 provides examples of what this schema accepts and rejects.

2.3. Extending JTD's Syntax

This document does not describe any extension mechanisms for JTD schema validation, which is described in Section 3. However, schemas are defined to optionally contain a `metadata` keyword, whose value is an arbitrary JSON object. Call the members of this object "metadata members".

Users **MAY** add metadata members to JTD schemas to convey information that is not pertinent to validation. For example, such metadata members could provide hints to code generators or trigger some special behavior for a library that generates user interfaces from schemas.

Users **SHOULD NOT** expect metadata members to be understood by other parties. As a result, if consistent validation with other parties is a requirement, users **MUST NOT** use metadata members to affect how schema validation, as described in Section 3, works.

Users **MAY** expect metadata members to be understood by other parties and **MAY** use metadata members to affect how schema validation works, if these other parties are somehow known to support these metadata members. For example, two parties may agree, out of band, that they will support an extended JTD with a custom metadata member that affects validation.

3. Semantics

This section describes when an instance is valid against a correct JTD schema and the error indicators to produce when an instance is invalid.

3.1. Allowing Additional Properties

Users will have different desired behavior with respect to "unspecified" members in an instance. For example, consider the JTD schema in Figure 2:

```
{ "properties": { "a": { "type": "string" }}}
```

Figure 2: An Illustrative JTD Schema

Some users may expect that

```
   {"a": "foo", "b": "bar"}
```

satisfies the schema in Figure 2. Others may disagree, as `b` is not one of the properties described in the schema. In this document, allowing such "unspecified" members, like `b` in this example, happens when evaluation is in "allow additional properties" mode.

Evaluation of a schema does not allow additional properties by default, but this can be overridden by having the schema include a member named `additionalProperties`, where that member has a value of `true`.

More formally, evaluation of a schema *S* is in "allow additional properties" mode if there exists a member of *S* whose name equals `additionalProperties` and whose value is a boolean `true`. Otherwise, evaluation of *S* is not in "allow additional properties" mode.

See Section 3.3.6 for how allowing unknown properties affects schema evaluation, but briefly, the schema

```
   { "properties": { "a": { "type": "string" }}}
```

rejects

```
   { "a": "foo", "b": "bar" }
```

However, the schema

```
{
  "additionalProperties": true,
  "properties": { "a": { "type": "string" }}
}
```

accepts

```
{ "a": "foo", "b": "bar" }
```

Note that `additionalProperties` does not get "inherited" by subschemas. For example, the JTD schema

```
{
  "additionalProperties": true,
  "properties": {
    "a": {
      "properties": {
        "b": { "type": "string" }
      }
    }
  }
}
```

accepts

```
{ "a": { "b": "c" }, "foo": "bar" }
```

but rejects

```
{ "a": { "b": "c", "foo": "bar" }}
```

because the `additionalProperties` at the root level does not affect the behavior of subschemas.

Note from Figure 1 that only schemas of the `properties` form may have a member named `additionalProperties`.

3.2. Errors

To facilitate consistent validation error handling, this document specifies a standard error indicator format. Implementations **SHOULD** support producing error indicators in this standard form.

The standard error indicator format is a JSON array. The order of the elements of this array is not specified. The elements of this array are JSON objects with:

- A member with the name `instancePath`, whose value is a JSON string encoding a JSON Pointer. This JSON Pointer will point to the part of the instance that was rejected.
- A member with the name `schemaPath`, whose value is a JSON string encoding a JSON Pointer. This JSON Pointer will point to the part of the schema that rejected the instance.

The values for `instancePath` and `schemaPath` depend on the form of the schema and are described in detail in Section 3.3.

3.3. Forms

This section describes, for each of the eight JTD schema forms, the rules dictating whether an instance is accepted, as well as the error indicators to produce when an instance is invalid.

The forms a correct schema may take on are formally described in Section 2.

3.3.1. Empty

The `empty` form is meant to describe instances whose values are unknown, unpredictable, or otherwise unconstrained by the schema. The syntax of the `empty` form is described in Section 2.2.1.

If a schema is of the empty form, then it accepts all instances. A schema of the empty form will never produce any error indicators.

3.3.2. Ref

The `ref` form is for when a schema is defined in terms of something in the `definitions` of the root schema. The ref form enables schemas to be less repetitive and also enables describing recursive structures. The syntax of the `ref` form is described in Section 2.2.2.

If a schema is of the ref form, then:

- If the schema has a member named `nullable` whose value is the boolean `true`, and the instance is the JSON primitive value `null`, then the schema accepts the instance. Otherwise:
- Let *B* be the root schema containing the schema or the schema itself if it is a root schema.
- Let *D* be the member of *B* with the name `definitions`. By Section 2, *D* exists.
- Let *R* be the value of the schema member with the name `ref`.
- Let *S* be the value of the member of *D* whose name equals *R*. By Section 2.2.2, *S* exists and is a schema.

The schema accepts the instance if and only if *S* accepts the instance. Otherwise, the error indicators to return in this case are the union of the error indicators from evaluating *S* against the instance.

For example, the schema

```
{
  "definitions": { "a": { "type": "float32" }},
  "ref": "a"
}
```

accepts

```
123
```

but rejects

```
null
```

with the error indicator

```
[{ "instancePath": "", "schemaPath": "/definitions/a/type" }]
```

The schema

```
{
  "definitions": { "a": { "type": "float32" }},
  "ref": "a",
  "nullable": true
}
```

accepts

```
null
```

because the schema has a `nullable` member whose value is `true`.

Note that `nullable` being `false` has no effect in any of the forms described in this document. For example, the schema

```
{
  "definitions": { "a": { "nullable": false, "type": "float32" }},
  "ref": "a",
  "nullable": true
}
```

accepts

```
null
```

In other words, it is not the case that putting a `false` value for `nullable` will ever override a `nullable` member in schemas of the `ref` form; it is correct, though ineffectual, to have a value of `false` for the `nullable` member in a schema.

3.3.3. Type

The `type` form is meant to describe instances whose value is a boolean, number, string, or timestamp [RFC3339]. The syntax of the `type` form is described in Section 2.2.3.

If a schema is of the type form, then:

- If the schema has a member named `nullable` whose value is the boolean `true`, and the instance is the JSON primitive value `null`, then the schema accepts the instance. Otherwise:
- Let *T* be the value of the member with the name `type`. The following table describes whether the instance is accepted, as a function of *T*'s value:

If _T_ equals ...	**then the instance is accepted if it is ...**
boolean	equal to `true` or `false`
float32	a JSON number
float64	a JSON number
int8	See Table 2
uint8	See Table 2
int16	See Table 2
uint16	See Table 2
int32	See Table 2
uint32	See Table 2
string	a JSON string
timestamp	a JSON string that follows the standard format described in [RFC3339], as refined by Section 3.3 of [RFC4287]

Table 1: Accepted Values for Type

`float32` and `float64` are distinguished from each other in their intent. `float32` indicates data intended to be processed as an IEEE 754 single-precision float, whereas `float64` indicates data intended to be processed as an IEEE 754 double-precision float. Tools that generate code from JTD schemas will likely produce different code for `float32` than for `float64`.

If *T* starts with `int` or `uint`, then the instance is accepted if and only if it is a JSON number encoding a value with zero fractional part. Depending on the value of *T*, this encoded number must additionally fall within a particular range:

_T_	Minimum Value (Inclusive)	Maximum Value (Inclusive)
int8	-128	127
uint8	0	255
int16	-32,768	32,767
uint16	0	65,535
int32	-2,147,483,648	2,147,483,647
uint32	0	4,294,967,295

Table 2: Ranges for Integer Types

Note that

```
10
```

and

```
10.0
```

and

```
1.0e1
```

encode values with zero fractional part, whereas

```
10.5
```

encodes a number with a non-zero fractional part. Thus, the schema

```
{"type": "int8"}
```

accepts

```
10
```

and

```
10.0
```

and

```
1.0e1
```

but rejects

```
10.5
```

as well as

```
false
```

because "false" is not a number at all.

If the instance is not accepted, then the error indicator for this case shall have an `instancePath` pointing to the instance and a `schemaPath` pointing to the schema member with the name `type`.

For example, the schema

```
{"type": "boolean"}
```

accepts

```
false
```

but rejects

```
127
```

The schema

```
   {"type": "float32"}
```

accepts

```
   10.5
```

and

```
   127
```

but rejects

```
   false
```

The schema

```
   {"type": "string"}
```

accepts

```
   "1985-04-12T23:20:50.52Z"
```

and

```
   "foo"
```

but rejects

```
   false
```

The schema

```
   {"type": "timestamp"}
```

accepts

```
"1985-04-12T23:20:50.52Z"
```

but rejects

```
"foo"
```

and

```
false
```

The schema

```
{"type": "boolean", "nullable": true}
```

accepts

```
null
```

and

```
false
```

but rejects

```
127
```

In all of the examples of rejected instances given in this section, the error indicator to produce is:

```
[{ "instancePath": "", "schemaPath": "/type" }]
```

3.3.4. Enum

The `enum` form is meant to describe instances whose value must be one of a given set of string values. The syntax of the `enum` form is described in Section 2.2.4.

If a schema is of the enum form, then:

- If the schema has a member named `nullable` whose value is the boolean `true`, and the instance is the JSON primitive value `null`, then the schema accepts the instance. Otherwise:

- Let *E* be the value of the schema member with the name `enum`. The instance is accepted if and only if it is equal to one of the elements of *E*.

If the instance is not accepted, then the error indicator for this case shall have an `instancePath` pointing to the instance and a `schemaPath` pointing to the schema member with the name `enum`.

For example, the schema

```
{ "enum": ["PENDING", "DONE", "CANCELED"] }
```

accepts

```
"PENDING"
```

and

```
"DONE"
```

and

```
"CANCELED"
```

but rejects all of

```
0
```

and

```
1
```

and

```
2
```

and

```
"UNKNOWN"
```

and

```
null
```

with the error indicator

```
[{ "instancePath": "", "schemaPath": "/enum" }]
```

The schema

```
{ "enum": ["PENDING", "DONE", "CANCELED"], "nullable": true }
```

accepts

```
"PENDING"
```

and

```
null
```

but rejects

```
1
```

and

```
"UNKNOWN"
```

with the error indicator

```
[{ "instancePath": "", "schemaPath": "/enum" }]
```

3.3.5. Elements

The `elements` form is meant to describe instances that must be arrays. A further subschema describes the elements of the array. The syntax of the `elements` form is described in Section 2.2.5.

If a schema is of the elements form, then:

- If the schema has a member named `nullable` whose value is the boolean `true`, and the instance is the JSON primitive value `null`, then the schema accepts the instance. Otherwise:

- Let *S* be the value of the schema member with the name `elements`. The instance is accepted if and only if all of the following are true:
    - The instance is an array. Otherwise, the error indicator for this case shall have an `instancePath` pointing to the instance and a `schemaPath` pointing to the schema member with the name `elements`.
    - If the instance is an array, then every element of the instance must be accepted by *S*. Otherwise, the error indicators for this case are the union of all the errors arising from evaluating *S* against elements of the instance.

For example, the schema

```
   {
     "elements": {
       "type": "float32"
     }
   }
```

accepts

```
   []
```

and

```
   [1, 2, 3]
```

but rejects

```
   null
```

with the error indicator

```
   [{ "instancePath": "", "schemaPath": "/elements" }]
```

and rejects

```
   [1, 2, "foo", 3, "bar"]
```

with the error indicators

```
[
  { "instancePath": "/2", "schemaPath": "/elements/type" },
  { "instancePath": "/4", "schemaPath": "/elements/type" }
]
```

The schema

```
{
  "elements": {
    "type": "float32"
  },
  "nullable": true
}
```

accepts

```
null
```

and

```
[]
```

and

```
[1, 2, 3]
```

but rejects

```
[1, 2, "foo", 3, "bar"]
```

with the error indicators

```
[
  { "instancePath": "/2", "schemaPath": "/elements/type" },
  { "instancePath": "/4", "schemaPath": "/elements/type" }
]
```

3.3.6. Properties

The `properties` form is meant to describe JSON objects being used as a "struct". The syntax of the `properties` form is described in Section 2.2.6.

If a schema is of the properties form, then:

- If the schema has a member named `nullable` whose value is the boolean `true`, and the instance is the JSON primitive value `null`, then the schema accepts the instance. Otherwise, the instance is accepted if and only if all of the following are true:
- The instance is an object.

  Otherwise, the error indicator for this case shall have an `instancePath` pointing to the instance and a `schemaPath` pointing to the schema member with the name `properties` if such a schema member exists; if such a member doesn't exist, `schemaPath` shall point to the schema member with the name `optionalProperties`.

- If the instance is an object, and the schema has a member named `properties`, then let *P* be the value of the schema member named `properties`. By Section 2.2.6, *P* must be an object. For every member name in *P*, a member of the same name in the instance must exist.

  Otherwise, the error indicator for this case shall have an `instancePath` pointing to the instance and a `schemaPath` pointing to the member of *P* failing the requirement just described.

- If the instance is an object, then let *P* be the value of the schema member named `properties` (if it exists) and *O* be the value of the schema member named `optionalProperties` (if it exists).

  For every member *I* of the instance, find a member with the same name as *I*'s in *P* or *O*. By Section 2.2.6, it is not possible for both *P* and *O* to have such a member. If the "discriminator tag exemption" is in effect on *I* (see Section 3.3.8), then ignore *I*. Otherwise:

  - If no such member in *P* or *O* exists and validation is not in "allow additional properties" mode (see Section 3.1), then the instance is rejected.

    The error indicator for this case has an `instancePath` pointing to *I* and a `schemaPath` pointing to the schema.

  - If such a member in *P* or *O* does exist, then call this member *S*. If *S* rejects *I*'s value, then the instance is rejected.

    The error indicators for this case are the union of the error indicators from evaluating *S* against *I*'s value.

An instance may have multiple errors arising from the third and fourth bullets in the list above. In this case, the error indicators are the union of the errors.

For example, the schema

```
{
  "properties": {
    "a": { "type": "string" },
    "b": { "type": "string" }
  },
  "optionalProperties": {
    "c": { "type": "string" },
    "d": { "type": "string" }
  }
}
```

accepts

```
{ "a": "foo", "b": "bar" }
```

and

```
{ "a": "foo", "b": "bar", "c": "baz" }
```

and

```
{ "a": "foo", "b": "bar", "c": "baz", "d": "quux" }
```

and

```
{ "a": "foo", "b": "bar", "d": "quux" }
```

but rejects

```
null
```

with the error indicator

```
[{ "instancePath": "", "schemaPath": "/properties" }]
```

and rejects

```
{ "b": 3, "c": 3, "e": 3 }
```

with the error indicators

```
    [
      { "instancePath": "",
        "schemaPath": "/properties/a" },
      { "instancePath": "/b",
        "schemaPath": "/properties/b/type" },
      { "instancePath": "/c",
        "schemaPath": "/optionalProperties/c/type" },
      { "instancePath": "/e",
        "schemaPath": "" }
    ]
```

If instead the schema had `additionalProperties: true` but was otherwise the same:

```
    {
      "properties": {
        "a": { "type": "string" },
        "b": { "type": "string" }
      },
      "optionalProperties": {
        "c": { "type": "string" },
        "d": { "type": "string" }
      },
      "additionalProperties": true
    }
```

and the instance remained the same:

```
    { "b": 3, "c": 3, "e": 3 }
```

then the error indicators from evaluating the instance against the schema would be:

```
    [
      { "instancePath": "",
        "schemaPath": "/properties/a" },
      { "instancePath": "/b",
        "schemaPath": "/properties/b/type" },
      { "instancePath": "/c",
        "schemaPath": "/optionalProperties/c/type" },
    ]
```

These are the same errors as before, except the final error (associated with the additional member named `e` in the instance) is no longer present. This is because `additionalProperties: true` enables "allow additional properties" mode on the schema.

Finally, the schema

```
{
  "nullable": true,
  "properties": {
    "a": { "type": "string" },
    "b": { "type": "string" }
  },
  "optionalProperties": {
    "c": { "type": "string" },
    "d": { "type": "string" }
  },
  "additionalProperties": true
}
```

accepts

```
null
```

but rejects

```
{ "b": 3, "c": 3, "e": 3 }
```

with the error indicators

```
[
  { "instancePath": "",
    "schemaPath": "/properties/a" },
  { "instancePath": "/b",
    "schemaPath": "/properties/b/type" },
  { "instancePath": "/c",
    "schemaPath": "/optionalProperties/c/type" },
]
```

3.3.7. Values

The `values` form is meant to describe instances that are JSON objects being used as an associative array. The syntax of the `values` form is described in Section 2.2.7.

If a schema is of the values form, then:

- If the schema has a member named `nullable` whose value is the boolean `true`, and the instance is the JSON primitive value `null`, then the schema accepts the instance. Otherwise:

- Let *S* be the value of the schema member with the name `values`. The instance is accepted if and only if all of the following are true:
    - The instance is an object. Otherwise, the error indicator for this case shall have an `instancePath` pointing to the instance and a `schemaPath` pointing to the schema member with the name `values`.
    - If the instance is an object, then every member value of the instance must be accepted by *S*. Otherwise, the error indicators for this case are the union of all the error indicators arising from evaluating *S* against member values of the instance.

For example, the schema

```
   {
     "values": {
       "type": "float32"
     }
   }
```

accepts

```
   {}
```

and

```
   {"a": 1, "b": 2}
```

but rejects

```
   null
```

with the error indicator

```
   [{ "instancePath": "", "schemaPath": "/values" }]
```

and rejects

```
   { "a": 1, "b": 2, "c": "foo", "d": 3, "e": "bar" }
```

with the error indicators

```
[
  { "instancePath": "/c", "schemaPath": "/values/type" },
  { "instancePath": "/e", "schemaPath": "/values/type" }
]
```

The schema

```
{
  "nullable": true,
  "values": {
    "type": "float32"
  }
}
```

accepts

```
null
```

but rejects

```
{ "a": 1, "b": 2, "c": "foo", "d": 3, "e": "bar" }
```

with the error indicators

```
[
  { "instancePath": "/c", "schemaPath": "/values/type" },
  { "instancePath": "/e", "schemaPath": "/values/type" }
]
```

3.3.8. Discriminator

The `discriminator` form is meant to describe JSON objects being used in a fashion similar to a discriminated union construct in C-like languages. The syntax of the `discriminator` form is described in Section 2.2.8.

When a schema is of the "discriminator" form, it validates that:

- the instance is an object,
- the instance has a particular "tag" property,
- this "tag" property's value is a string within a set of valid values, and
- the instance satisfies another schema, where this other schema is chosen based on the value of the "tag" property.

The behavior of the discriminator form is more complex than the other keywords. Readers familiar with CDDL may find the final example in Appendix B helpful in understanding its behavior. What follows in this section is a description of the discriminator form's behavior, as well as some examples.

If a schema is of the "discriminator" form, then:

- Let *D* be the schema member with the name `discriminator`.
- Let *M* be the schema member with the name `mapping`.
- Let *I* be the instance member whose name equals *D*'s value. *I* may, for some rejected instances, not exist.
- Let *S* be the member of *M* whose name equals *I*'s value. *S* may, for some rejected instances, not exist.

If the schema has a member named `nullable` whose value is the boolean `true`, and the instance is the JSON primitive value `null`, then the schema accepts the instance. Otherwise, the instance is accepted if and only if all of the following are true:

- The instance is an object.

  Otherwise, the error indicator for this case shall have an `instancePath` pointing to the instance and a `schemaPath` pointing to *D*.

- If the instance is a JSON object, then *I* must exist.

  Otherwise, the error indicator for this case shall have an `instancePath` pointing to the instance and a `schemaPath` pointing to *D*.

- If the instance is a JSON object and *I* exists, *I*'s value must be a string.

  Otherwise, the error indicator for this case shall have an `instancePath` pointing to *I* and a `schemaPath` pointing to *D*.

- If the instance is a JSON object and *I* exists and has a string value, then *S* must exist.

  Otherwise, the error indicator for this case shall have an `instancePath` pointing to *I* and a `schemaPath` pointing to *M*.

- If the instance is a JSON object, *I* exists, and *S* exists, then the instance must satisfy *S*'s value. By Section 2, *S*'s value must be a schema of the properties form. Apply the "discriminator tag exemption" afforded in Section 3.3.6 to *I* when evaluating whether the instance satisfies *S*'s value.

  Otherwise, the error indicators for this case shall be error indicators from evaluating *S*'s value against the instance, with the "discriminator tag exemption" applied to *I*.

The list items above are defined in a mutually exclusive way. For any given instance and schema, exactly one of the list items above will apply.

For example, the schema

```
{
  "discriminator": "version",
  "mapping": {
    "v1": {
      "properties": {
        "a": { "type": "float32" }
      }
    },
    "v2": {
      "properties": {
        "a": { "type": "string" }
      }
    }
  }
}
```

rejects

```
null
```

with the error indicator

```
[{ "instancePath": "", "schemaPath": "/discriminator" }]
```

(This is the case of the instance not being an object.)

Also rejected is

```
{}
```

with the error indicator

```
[{ "instancePath": "", "schemaPath": "/discriminator" }]
```

(This is the case of *I* not existing.)

Also rejected is

```
{ "version": 1 }
```

with the error indicator

```
[
  {
    "instancePath": "/version",
    "schemaPath": "/discriminator"
  }
]
```

(This is the case of *I* existing but not having a string value.)

Also rejected is

```
{ "version": "v3" }
```

with the error indicator

```
[
  {
    "instancePath": "/version",
    "schemaPath": "/mapping"
  }
]
```

(This is the case of *I* existing and having a string value but *S* not existing.)

Also rejected is

```
{ "version": "v2", "a": 3 }
```

with the error indicator

```
[
  {
    "instancePath": "/a",
    "schemaPath": "/mapping/v2/properties/a/type"
  }
]
```

(This is the case of *I* and *S* existing but the instance not satisfying *S*'s value.)

Finally, the schema accepts

```
{ "version": "v2", "a": "foo" }
```

This instance is accepted even though `version` is not mentioned by `/mapping/v2/properties`; the "discriminator tag exemption" ensures that `version` is not treated as an additional property when evaluating the instance against *S*'s value.

By contrast, consider the same schema but with `nullable` being `true`. The schema

```
{
  "nullable": true,
   "discriminator": "version",
   "mapping": {
     "v1": {
       "properties": {
         "a": { "type": "float32" }
       }
     },
     "v2": {
       "properties": {
         "a": { "type": "string" }
       }
     }
   }
}
```

accepts

```
null
```

To further illustrate the discriminator form with examples, recall the JTD schema in Section 2.2.8, reproduced here:

```
{
  "discriminator": "event_type",
  "mapping": {
    "account_deleted": {
      "properties": {
        "account_id": { "type": "string" }
      }
    },
    "account_payment_plan_changed": {
      "properties": {
        "account_id": { "type": "string" },
        "payment_plan": { "enum": ["FREE", "PAID"] }
      },
      "optionalProperties": {
        "upgraded_by": { "type": "string" }
      }
    }
  }
}
```

This schema accepts

```
{ "event_type": "account_deleted", "account_id": "abc-123" }
```

and

```
{
  "event_type": "account_payment_plan_changed",
  "account_id": "abc-123",
  "payment_plan": "PAID"
}
```

and

```
{
  "event_type": "account_payment_plan_changed",
  "account_id": "abc-123",
  "payment_plan": "PAID",
  "upgraded_by": "users/mkhwarizmi"
}
```

but rejects

```
{}
```

with the error indicator

```
[{ "instancePath": "", "schemaPath": "/discriminator" }]
```

and rejects

```
{ "event_type": "some_other_event_type" }
```

with the error indicator

```
[
  {
    "instancePath": "/event_type",
    "schemaPath": "/mapping"
  }
]
```

and rejects

```
{ "event_type": "account_deleted" }
```

with the error indicator

```
[{
  "instancePath": "",
  "schemaPath": "/mapping/account_deleted/properties/account_id"
}]
```

and rejects

```
{
  "event_type": "account_payment_plan_changed",
  "account_id": "abc-123",
  "payment_plan": "PAID",
  "xxx": "asdf"
}
```

with the error indicator

```
[{
  "instancePath": "/xxx",
  "schemaPath": "/mapping/account_payment_plan_changed"
}]
```

4. IANA Considerations

This document has no IANA actions.

5. Security Considerations

Implementations of JTD will necessarily be manipulating JSON data. Therefore, the security considerations of [RFC8259] are all relevant here.

Implementations that evaluate user-inputted schemas **SHOULD** implement mechanisms to detect and abort circular references that might cause a naive implementation to go into an infinite loop. Without such mechanisms, implementations may be vulnerable to denial-of-service attacks.

6. References

6.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/info/rfc3339>.

[RFC4287] Nottingham, M., Ed. and R. Sayre, Ed., "The Atom Syndication Format", RFC 4287, DOI 10.17487/RFC4287, December 2005, <https://www.rfc-editor.org/info/rfc4287>.

[RFC6901] Bryan, P., Ed., Zyp, K., and M. Nottingham, Ed., "JavaScript Object Notation (JSON) Pointer", RFC 6901, DOI 10.17487/RFC6901, April 2013, <https://www.rfc-editor.org/info/rfc6901>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <https://www.rfc-editor.org/info/rfc8610>.

6.2. Informative References

[JSON-SCHEMA] Wright, A., Andrews, H., Hutton, B., and G. Dennis, "JSON Schema: A Media Type for Describing JSON Documents", Work in Progress, Internet-Draft, draft-handrews-json-schema-02, 17 September 2019, <https://tools.ietf.org/html/draft-handrews-json-schema-02>.

[OPENAPI] OpenAPI Initiative, "OpenAPI Specification", October 2018, <https://spec.openapis.org/oas/v3.0.2>.

[RFC7071] Borenstein, N. and M. Kucherawy, "A Media Type for Reputation Interchange", RFC 7071, DOI 10.17487/RFC7071, November 2013, <https://www.rfc-editor.org/info/rfc7071>.

[RFC7493] Bray, T., Ed., "The I-JSON Message Format", RFC 7493, DOI 10.17487/RFC7493, March 2015, <https://www.rfc-editor.org/info/rfc7493>.

Appendix A. Rationale for Omitted Features

This appendix is not normative.

This section describes possible features that are intentionally left out of JSON Type Definition and justifies why these features are omitted.

A.1. Support for 64-Bit Numbers

This document does not allow `int64` or `uint64` as values for the JTD `type` keyword (see Sections 2.2.3 and 3.3.3). Such hypothetical `int64` or `uint64` types would behave like `int32` or `uint32` (respectively) but with the range of values associated with 64-bit instead of 32-bit integers. That is:

- `int64` would accept numbers between -(2**63) and (2**63)-1
- `uint64` would accept numbers between 0 and (2**64)-1

Users of `int64` and `uint64` would likely expect that the full range of signed or unsigned 64-bit integers could interoperably be transmitted as JSON without loss of precision. But this assumption is likely to be incorrect, for the reasons given in Section 2.2 of [RFC7493].

`int64` and `uint64` likely would have led users to falsely assume that the full range of 64-bit integers can be interoperably processed as JSON without loss of precision. To avoid leading users astray, JTD omits `int64` and `uint64`.

A.2. Support for Non-root Definitions

This document disallows the `definitions` keyword from appearing outside of root schemas (see Figure 1). Conceivably, this document could have instead allowed `definitions` to appear on any schema, even non-root ones. Under this alternative design, `ref`s would resolve to a definition in the "nearest" (i.e., most nested) schema that both contained the `ref` and had a suitably named `definitions` member.

For instance, under this alternative approach, one could define schemas like the one in Figure 3.

```
{
  "properties": {
    "foo": {
      "definitions": {
        "user": { "properties": { "user_id": {"type": "string" }}}
      },
      "ref": "user"
    },
    "bar": {
      "definitions": {
        "user": { "properties": { "user_id": {"type": "string" }}}
      },
      "ref": "user"
    },
    "baz": {
      "definitions": {
        "user": { "properties": { "userId": {"type": "string" }}}
      },
      "ref": "user"
    }
  }
}
```

Figure 3: A Hypothetical Schema Had This Document Permitted Non-root Definitions. This Is Not a Correct JTD Schema.

If schemas like that in Figure 3 were permitted, code generation from JTD schemas would be more difficult, and the generated code would be less useful.

Code generation would be more difficult because it would force code generators to implement a name-mangling scheme for types generated from definitions. This additional difficulty is not immense, but it adds complexity to an otherwise relatively trivial task.

Generated code would be less useful because generated, mangled struct names are less pithy than human-defined struct names. For instance, the `user` definitions in Figure 3 might have been generated into types named `PropertiesFooUser`, `PropertiesBarUser`, and `PropertiesBazUser`; obtuse names like these are less useful to human-written code than names like `User`.

Furthermore, even though `PropertiesFooUser` and `PropertiesBarUser` would be essentially identical, they would not be interchangeable in many statically typed programming languages. A code generator could attempt to circumvent this by deduplicating identical definitions, but then the user might be confused as to why the subtly distinct `PropertiesBazUser`, defined from a schema allowing a property named `userId` (not `user_id`), was not deduplicated.

Because there seem to be implementation and usability challenges associated with non-root definitions, and because it would be easier to later amend JTD to permit for non-root definitions than to later amend JTD to prohibit them, this document does not permit non-root definitions in JTD schemas.

Appendix B. Comparison with CDDL

This appendix is not normative.

To aid the reader familiar with CDDL, this section illustrates how JTD works by presenting JTD schemas and CDDL schemas that accept and reject the same instances.

The JTD schema

```
{}
```

accepts the same instances as the CDDL rule

```
root = any
```

The JTD schema

```
{
  "definitions": {
    "a": { "elements": { "ref": "b" }},
    "b": { "type": "float32" }
  },
  "elements": {
    "ref": "a"
  }
}
```

accepts the same instances as the CDDL rule

```
root = [* a]
a = [* b]
b = number
```

The JTD schema

```
{ "enum": ["PENDING", "DONE", "CANCELED"]}
```

accepts the same instances as the CDDL rule

```
root = "PENDING" / "DONE" / "CANCELED"
```

The JTD schema

```
   {"type": "boolean"}
```

accepts the same instances as the CDDL rule

```
   root = bool
```

The JTD schemas:

```
   {"type": "float32"}
```

and

```
   {"type": "float64"}
```

both accept the same instances as the CDDL rule

```
   root = number
```

The JTD schema

```
   {"type": "string"}
```

accepts the same instances as the CDDL rule

```
   root = tstr
```

The JTD schema

```
   {"type": "timestamp"}
```

accepts the same instances as the CDDL rule

```
   root = tdate
```

The JTD schema

```
{ "elements": { "type": "float32" }}
```

accepts the same instances as the CDDL rule

```
root = [* number]
```

The JTD schema

```
{
  "properties": {
    "a": { "type": "boolean" },
    "b": { "type": "float32" }
  },
  "optionalProperties": {
    "c": { "type": "string" },
    "d": { "type": "timestamp" }
  }
}
```

accepts the same instances as the CDDL rule

```
root = { a: bool, b: number, ? c: tstr, ? d: tdate }
```

The JTD schema

```
{ "values": { "type": "float32" }}
```

accepts the same instances as the CDDL rule

```
root = { * tstr => number }
```

Finally, the JTD schema

```
{
  "discriminator": "a",
  "mapping": {
    "foo": {
      "properties": {
        "b": { "type": "float32" }
      }
    },
    "bar": {
      "properties": {
        "b": { "type": "string" }
      }
    }
  }
}
```

accepts the same instances as the CDDL rule

```
root = { a: "foo", b: number } / { a: "bar", b: tstr }
```

Appendix C. Example

This appendix is not normative.

As a demonstration of JTD, in Figure 4 is a JTD schema closely equivalent to the plain-English definition `reputation-object` described in Section 6.2.2 of [RFC7071]:

```
{
  "properties": {
    "application": { "type": "string" },
    "reputons": {
      "elements": {
        "additionalProperties": true,
        "properties": {
          "rater": { "type": "string" },
          "assertion": { "type": "string" },
          "rated": { "type": "string" },
          "rating": { "type": "float32" },
        },
        "optionalProperties": {
          "confidence": { "type": "float32" },
          "normal-rating": { "type": "float32" },
          "sample-size": { "type": "float64" },
          "generated": { "type": "float64" },
          "expires": { "type": "float64" }
        }
      }
    }
  }
}
```

Figure 4: A JTD Schema Describing "reputation-object" from Section 6.2.2 of [RFC7071]

This schema does not enforce the requirement that `sample-size`, `generated`, and `expires` be unbounded positive integers. It does not express the limitation that `rating`, `confidence`, and `normal-rating` should not have more than three decimal places of precision.

The example in Figure 4 can be compared against the equivalent example in Appendix H of [RFC8610].

Acknowledgments

Carsten Bormann provided lots of useful guidance and feedback on JTD's design and the structure of this document.

Evgeny Poberezkin suggested the addition of `nullable` and thoroughly vetted this document for mistakes and opportunities for simplification.

Tim Bray suggested the current `ref` model and the addition of `enum`. Anders Rundgren suggested extending `type` to have more support for numerical types. James Manger suggested additional clarifying examples of how integer types work. Adrian Farrel suggested many improvements to help make this document clearer.

Members of the IETF JSON mailing list -- in particular, Pete Cordell, Phillip Hallam-Baker, Nico Williams, John Cowan, Rob Sayre, and Erik Wilde -- provided lots of useful feedback.

OpenAPI's `discriminator` object [OPENAPI] inspired the `discriminator` form. [JSON-SCHEMA] influenced various parts of JTD's early design.

Author's Address

Ulysse Carion
Segment.io, Inc
100 California Street
San Francisco, CA 94111
United States of America
Email: ulysse@segment.com