Abstract

This document proposes a format, called JSON Data Definition Format (JDDF), 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, JDDF is strategically limited to be no more expressive than the type systems of mainstream programming languages. This strategic limitation, as well as the decision to make JDDF schemas be JSON documents, also makes tooling atop of JDDF easier to build.

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

This document describes a schema language for JSON [RFC8259] called JSON Data Definition Format (JDDF). The name JDDF is chosen to avoid confusion with "JSON Schema" from [I-D.handrews-json-schema].

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

The goals of JDDF are to:

- Provide an unambiguous description of the overall structure of a JSON document.
- Be able to describe common JSON datatypes and structures. That is, the datatypes and structures necessary to support most JSON documents, and which 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 which can be embedded within other JSON documents. This makes JDDF a convenient format for tooling to accept as input, or produce as output.
- Enable code generation from JDDF schemas. JDDF schemas are meant to be easy to convert into data structures idiomatic to a given mainstream programming language.
- Provide a standardized format for errors when data does not conform with a schema.

JDDF is intentionally designed as a rather minimal schema language. Thus, although JDDF can describe JSON, it is not able to describe its own structure: the Concise Data Definition Language (CDDL) [RFC8610] is used to describe JDDF in this document. By keeping the expressiveness of the schema language minimal, JDDF makes code generation and standardized errors 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 JDDF, but are not limiting in any way.

JDDF’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, JDDF supports:

- Signed and unsigned 8, 16, and 32-bit integers. A tool which converts JDDF schemas into code can use "int8_t", "uint8_t", "int16_t", etc., or their equivalents in the target language, to represent these JDDF types.
o A distinction between "float32" and "float64". Code generators can use "float" and "double", or their equivalents, for these JDDF types.

o 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".

o 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".

o 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 JDDF does not support 64-bit integers, as these are usually transmitted over JSON in a non-interoperable (i.e., ignoring the recommendations in Section 2.2 of [RFC7493]) or mutually inconsistent (e.g., using hexadecimal versus base64) ways. Appendix A.1 further elaborates on why JDDF does not support 64-bit integers.

The principle of clear correspondence to common programming languages is why JDDF does not support, for example, a data type for numbers up to $2^{53}-1$.

It is expected that for many use-cases, a schema language of JDDF'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 JDDF. The purpose of the experiment is to gain experience with JDDF and to possibly revise this work accordingly. If JDDF 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:

The syntax of JDDF is defined in Section 2. Section 3 describes the semantics of JDDF; 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 JDDF. Appendix B presents various JDDF 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. These words may also appear in this document in lower case as plain English words, absent their normative meanings.

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 JDDF schema.

1.2. Scope of Experiment

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

Nodes know if they are participating in the experiment if they are validating JSON messages against a JDDF 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 JDDF schema.

The risk of this experiment "escaping" takes the form of a JDDF-supporting node expecting another node, which lacks such support, to validate messages against some JDDF 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 JDDF has been implemented by multiple independent parties, and these parties successfully use JDDF to facilitate information interchange within their internal systems or between systems operated by independent parties.

If this experiment is deemed successful, and JDDF 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 JDDF.

2. Syntax

This section describes when a JSON document is a correct JDDF schema. Because CDDL is well-suited to the task of defining complex JSON formats, such as JDDF schemas, this section uses CDDL to describe the format of JDDF schemas.

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

A JDDF schema is a JSON object taking on an appropriate form. JDDF schemas may contain "additional data", discussed in Section 2.1. Root JDDF schemas may optionally contain definitions (a mapping from names to schemas).

A correct root JDDF schema MUST match the "root-schema" CDDL rule described in this section. A correct non-root JDDF 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 = {
    schema,
    ? definitions: { * tstr => schema },
}

schema is the main CDDL rule defining a JDDF schema. Certain JDDF
schema forms will be defined recursively in terms of this rule.
schema = {
    form,
    * non-keyword => *
}

any of the keywords defined later.
non-keyword = 
    ( ((((((.ne "definitions")
        .ne "ref")
        .ne "type")
        .ne "enum")
        .ne "elements")
        .ne "properties")
        .ne "optionalProperties")
        .ne "additionalProperties")
        .ne "values")
        .ne "discriminator"

Figure 1: CDDL definition of a schema

Thus Figure 2 is not a correct JDDF schema, as its "definitions"
object contains a number, which is not a schema:

{ "definitions": { "foo": 3 } }

Figure 2: An incorrect JDDF schema. JSON numbers are not JDDF
schemas

Figure 3 is also incorrect, as a "definitions" object may not appear
on non-root schemas. See Figure 16 for more details on how
"elements" is defined in terms of the "schema" CDDL rule.
Figure 3: An incorrect JDDF schema. "definitions" may appear only in
root schemas

Figure 4 is an example of a correct schema that uses "definitions":

```json
{
  "definitions": {
    "user": {
      "properties": {
        "name": { "type": "string" },
        "create_time": { "type": "timestamp" }
      }
    },
    "elements": {
      "ref": "user"
    }
  }
}
```

Figure 4: A correct JDDF schema using "definitions"

JDDF schemas can take on one of eight forms. These forms are defined
so as to be mutually exclusive; a schema cannot satisfy multiple
forms at once.

```
form = empty / ref / type / enum / elements / properties / values / discriminator
```

Figure 5: CDDL definition of the JDDF schema forms

The first form, "empty", is trivial. It is meant for matching any
instance:

```
empty = {}
```

Figure 6: CDDL definition of the "empty" form
Thus, Figure 7 is a correct schema:

{}  

Figure 7: A JDDF schema of the "empty" form

The empty form is not very useful by itself, and it meant to be used as a sub-schema. Schema authors can use the empty form to describe parts of a message format which do not contain predictable data, or which the author does not want to specify.

The semantics of schemas of the empty form are described in Section 3.3.1.

The second form, "ref", is for when a schema is defined in terms of something in the "definitions" of the root schema:

ref = { ref: tstr }

Figure 8: CDDL definition of the "ref" form

For a schema to be correct, the "ref" value 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_.

Figure 9 is a correct example of "ref" being used to avoid re-defining the same thing twice:
{ "definitions": { "coordinates": { "properties": { "lat": { "type": "float32" }, "lng": { "type": "float32" } } }, "properties": { "user_location": { "ref": "coordinates" }, "server_location": { "ref": "coordinates" } } }

Figure 9: A correct JDDF schema using the "ref" form

However, Figure 10 is incorrect, as it refers to a definition that doesn’t exist:

{ "definitions": { "foo": { "type": "float32" }}, "ref": "bar" }

Figure 10: An incorrect JDDF schema. There is no "bar" in "definitions"

The semantics of schemas of the "ref" form are described in Section 3.3.2.

The third form, "type", constrains instances to have a particular primitive type. The precise meaning of each of the primitive types is described in Section 3.3.3.

type = { type: "boolean" / num-type / "string" / "timestamp" }
num-type = "float32" / "float64" / "int8" / "uint8" / "int16" / "uint16" / "int32" / "uint32"

Figure 11: CDDL Definition of the Type Form

For example, Figure 12 constrains instances to be strings that are correct [RFC3339] timestamps:

{ "type": "timestamp" }

Figure 12: A correct JDDF schema using the "type" form
The semantics of schemas of the "type" form are described in
Section 3.3.3.

The fourth form, "enum", describes instances whose value must be one
of a finite, predetermined set of values:

```
enum = { enum: [+ tstr] }
```

Figure 13: CDDL definition of the "enum" form

The values within "[+ tstr]" MUST NOT contain duplicates. Thus,
Figure 14 is a correct schema:

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

Figure 14: A correct JDDF schema using the "enum" form

But Figure 15 is not a correct schema, as "B" is duplicated:

```
{ "enum": ["A", "B", "B"] }
```

Figure 15: An incorrect JDDF schema. "B" appears twice.

The semantics of schemas of the "enum" form are described in
Section 3.3.4.

The fifth form, "elements", describes instances that must be arrays.
A further sub-schema describes the elements of the array.

```
elements = { elements: schema }
```

Figure 16: CDDL definition of the "elements" form

Figure 17 is a schema describing an array of [RFC3339] timestamps:

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

Figure 17: A correct JDDF schema using the "elements" form

The semantics of schemas of the "elements" form are described in
Section 3.3.5.

The sixth form, "properties", describes JSON objects being used as a
"struct". A schema of this form specifies the names of required and
optional properties, as well as the schemas each of those properties
must satisfy:

```
; One of properties or optionalProperties may be omitted, 
; but not both.
properties = with-properties / with-optional-properties

with-properties = {
  properties: * tstr => schema,
  ? optionalProperties * tstr => schema,
  ? additionalProperties: bool,
}

with-optional-properties = {
  ? properties: * tstr => schema,
  optionalProperties: * tstr => schema,
  ? additionalProperties: bool,
}

Figure 18: CDDL definition of the "properties" form

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. This is to prevent
ambiguity as to whether a property is optional or required.

Thus, Figure 19 is not a correct schema, as "confusing" appears in
both "properties" and "optionalProperties":

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

Figure 19: An incorrect JDDF schema. "confusing" is repeated between
"properties" and "optionalProperties"

Figure 20 is a correct schema, describing a paginated list of users:
The semantics of schemas of the "properties" form are described in Section 3.3.6.

The seventh form, "values", describes JSON objects being used as an associative array. A schema of this form specifies the form all member values must satisfy, but places no constraints on the member names:

\[
\text{values} = \{ \text{values: } * \text{ tstr} \Rightarrow \text{schema} \}
\]

Figure 21: CDDL definition of the "values" form

Thus, Figure 22 is a correct schema, describing a mapping from strings to numbers:

\[
\{ \text{"values": } \{ \text{"type": "float32" } \} \}
\]

Figure 22: A correct JDDF schema using the "values" form

The semantics of schemas of the "values" form are described in Section 3.3.7.

Finally, the eighth form, "discriminator", describes JSON objects being used as a discriminated union. A schema of this form specifies the "tag" (or "discriminator") of the union, as well as a mapping from tag values to the appropriate schema to use.
; Note well: the values of mapping are of the properties form.
discriminator = { tag: tstr, mapping: * tstr => properties }

Figure 23: CDDL definition of the "discriminator" form

To prevent ambiguous or unsatisfiable constraints on the "tag" of a
discriminator, an additional constraint on schemas of the
discriminator form exists. For schemas of the discriminator form:

- Let _D_ be the schema member with the name "discriminator".
- Let _T_ be the member of _D_ with the name "tag".
- Let _M_ be the member of _D_ 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 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
_T_’s value.

Thus, Figure 24 is an incorrect schema, as "event_type" is both the
value of "tag" and a member name in one of the "mapping" member
"properties":

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

Figure 24: An incorrect JDDF schema. "event_type" appears both in
"tag" and in the "properties" of a "mapping" value

However, Figure 25 is a correct schema, describing a pattern of data
common in JSON-based messaging systems:
The semantics of schemas of the "discriminator" form are described in Section 3.3.8. Section 3.3.8 also includes examples of what Figure 25 accepts and rejects.

2.1. Extending JDDF’s Syntax

This document does not describe any extension mechanisms for JDDF schema validation, which is described in Section 3. However, schemas (through the "non-keyword" CDDL rule in Section 2) are defined to allow members whose names are not equal to any of the specially-defined keywords (i.e. "definitions", "elements", etc.). Call these members "non-keyword members".

Users MAY add additional, non-keyword members to JDDF schemas to convey information that is not pertinent to validation. For example, such non-keyword 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 non-keyword members to be understood by other parties. As a result, if consistent validation with other parties is a requirement, users SHOULD NOT use non-keyword members to affect how schema validation, as described in Section 3, works.

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

3. Semantics

This section describes when an instance is valid against a correct
JDDF 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
JDDF schema in Figure 26:

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

Figure 26: An illustrative JDDF schema

Some users may expect that

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

satisfies the schema in Figure 26. 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 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, consider the schema in Figure 27:

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

Figure 27: A JDDF schema that does not allow additional properties
The schema in Figure 27 rejects

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

However, consider the schema in Figure 28:

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

Figure 28: A JDDF schema that allows additional properties

The schema in Figure 28 accepts

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

Note that "additionalProperties" does not get "inherited" by sub-schemas. For example, the JDDF 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 sub-schemas.

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 the members:

- 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 JDDF 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.

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.

If a schema is of the ref form, then:

- 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".

The values for "instancePath" and "schemaPath" depend on the form of the schema, and are described in detail in Section 3.3.
Let \( S \) be the value of the member of \( D \) whose name equals \( R \). By Section 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"
}
```

Figure 29: A JDDF schema demonstrating the "ref" form

Accepts 123 but not false

The error indicators to produce when evaluating false against the schema in Figure 29 are:

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

Note that the ref form is defined to only look up definitions at the root level. Thus, with the schema:

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

The instance 123
is accepted, and
false
is rejected, and the error indicator would be:

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

Though non-root definitions are not syntactically disallowed in correct schemas, they are entirely immaterial to evaluating references.

3.3.3. Type

The type form is meant to describe instances whose value is a boolean, number, string, or timestamp ([RFC3339]).

If a schema is of the type form, then 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:

<table>
<thead>
<tr>
<th>If <em>T</em> equals ...</th>
<th>then the instance is accepted if it is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>equal to &quot;true&quot; or &quot;false&quot;</td>
</tr>
<tr>
<td>float32</td>
<td>a JSON number</td>
</tr>
<tr>
<td>float64</td>
<td>a JSON number</td>
</tr>
<tr>
<td>int8</td>
<td>See Table 2</td>
</tr>
<tr>
<td>uint8</td>
<td>See Table 2</td>
</tr>
<tr>
<td>int16</td>
<td>See Table 2</td>
</tr>
<tr>
<td>uint16</td>
<td>See Table 2</td>
</tr>
<tr>
<td>int32</td>
<td>See Table 2</td>
</tr>
<tr>
<td>uint32</td>
<td>See Table 2</td>
</tr>
<tr>
<td>string</td>
<td>a JSON string</td>
</tr>
<tr>
<td>timestamp</td>
<td>a JSON string encoding a [RFC3339] timestamp</td>
</tr>
</tbody>
</table>

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 which generate code from JDDF 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:

<table>
<thead>
<tr>
<th><em>T</em></th>
<th>Minimum Value (Inclusive)</th>
<th>Maximum Value (Inclusive)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8</td>
<td>-128</td>
<td>127</td>
</tr>
<tr>
<td>uint8</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>int16</td>
<td>-32,768</td>
<td>32,767</td>
</tr>
<tr>
<td>uint16</td>
<td>0</td>
<td>65,535</td>
</tr>
<tr>
<td>int32</td>
<td>-2,147,483,648</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>uint32</td>
<td>0</td>
<td>4,294,967,295</td>
</tr>
</tbody>
</table>

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

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 finite, predetermined set of string values.

If a schema is of the enum form, then 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:

```json

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

Accepts

"PENDING"

and

"DONE"

and

"CANCELED"

but rejects all of

0

and

1

and

2

and

"UNKNOWN"

with the error indicator:

```json

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

The elements form is meant to describe instances that must be arrays. A further sub-schema describes the elements of the array.

If a schema is of the elements form, then 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:

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

accepts

```
[]
```

and

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

but rejects

```
false
```

with the error indicator:

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

and rejects

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

with the error indicators:
3.3.6. Properties

The properties form is meant to describe JSON objects being used as a "struct".

If a schema is of the properties form, then 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". _P_, by Section 2, 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, 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 second and third bullet in the above. In this case, the error indicators are the union of the errors.

For example, the schema:

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

accepts

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

and

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

and

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

and

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

but rejects

```
123
```

with the error indicator
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 the schema would be
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.

3.3.7. Values

The elements form is meant to describe instances that are JSON objects being used as an associative array.

If a schema is of the values form, then 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:

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

accepts

{}

and

{"a": 1, "b": 2}
but rejects

false

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" }
]

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

- That the instance is an object,
- That the instance has a particular "tag" property,
- That this "tag" property’s value is a string within a set of valid values, and
- That 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 _T_ be the member of _D_ with the name "tag".
Let _M_ be the member of _D_ with the name "mapping".

Let _I_ be the instance member whose name equals _T_’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.

The instance is accepted if and only if:

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 _I_, and a "schemaPath" pointing to _T_.

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 _T_.

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 have 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_.

By Section 2, _S_’s value must have the properties form.
Each of the list items above are defined to be mutually exclusive. For the same instance and schema, only one of the list items above will apply.

For example, the schema:

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

rejects

"example"

with the error indicator

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

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

Also rejected is

```json
{}
```

with the error indicator

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

(This is the case of _I_ not existing.)

Also rejected is

```json
{ "version": 1 }
```
with the error indicator

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

(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": "/discriminator/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": "/discriminator/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 despite the fact that "version" is not mentioned by "/discriminator/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.

To further illustrate the discriminator form with examples, recall the JDDF schema in Figure 25, reproduced here:

```json
{
  "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" }
      }
    }
  }
}
```

This schema accepts

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

and

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

and
but rejects

```json
{}
```
with the error indicator

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

and rejects

```json
{ "event_type": "some_other_event_type" }
```
with the error indicator

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

and rejects

```json
{ "event_type": "account_deleted" }
```
with the error indicator

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

and rejects

```json
{ "event_type": "account_payment_plan_changed",
  "account_id": "abc-123",
  "payment_plan": "PAID",
  "xxx": "asdf"
}
```
with the error indicator

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

4. IANA Considerations

No IANA considerations.

5. Security Considerations

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

Implementations which evaluate user-inputted schemas SHOULD implement mechanisms to detect, and abort, circular references which 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


6.2. Informative References


Appendix A. Other Considerations

This appendix is not normative.

This section describes possible features which are intentionally left out of JSON Data Definition Format, 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 JDDF "type" keyword (see Figure 11 and Section 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**63)-1

This appendix is not normative.
"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, JDDF omits "int64" and "uint64".

A.2. Support for Non-Root Schemas

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 which both contained the "ref" and which had a suitably-named "definitions" member.

For instance, under this alternative approach, one could define schemas like the one in Figure 30:
Figure 30: A hypothetical schema had this document permitted non-root definitions. This is not a correct JDDF schema.

If schemas like that in Figure 30 were permitted, code generation from JDDF 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 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 30 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 JDDF to permit for non-root definitions than to later amend JDDF to prohibit them, this document does not permit non-root definitions in JDDF schemas.

**Appendix B. Comparison with CDDL**

This appendix is not normative.

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

The JDDF schema:

```
{}  
```

accepts the same instances as the CDDL rule:

```
root = any 
```

The JDDF 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 JDDF schema:

```
{ "enum": ["PENDING", "DONE", "CANCELED"]} 
```
accepts the same instances as the CDDL rule:

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

The JDDF schema:

    {"type": "boolean"}

accepts the same instances as the CDDL rule:

    root = bool

The JDDF schemas:

    {"type": "float32"}

and

    {"type": "float64"}

both accept the same instances as the CDDL rule:

    root = number

The JDDF schema:

    {"type": "string"}

accepts the same instances as the CDDL rule:

    root = tstr

The JDDF schema:

    {"type": "timestamp"}

accepts the same instances as the CDDL rule:

    root = tdate

The JDDF schema:

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

accepts the same instances as the CDDL rule:

    root = [* number]
The JDDF 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 JDDF schema:

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

accepts the same instances as the CDDL rule:

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

Finally, the JDDF schema:

```
{
  "discriminator": {
    "tag": "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. Examples

This appendix is not normative.

As a demonstration of JDDF, in Figure 31 is a JDDF schema closely equivalent to the plain-English definition "reputation-object" described in Section 6.2.2 of [RFC7071]:

```json
{
    "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 31: A JDDF schema describing "reputation-object" from Section 6.6.2 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 31 can be compared against the equivalent example in Appendix H of [RFC8610].

Acknowledgments

Carsten Bormann provided lots of useful guidance and feedback on JDDF’s design and the structure of this document.
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. 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. [I-D.handrews-json-schema] influenced various parts of JDDF’s early design.

Author’s Address

Ulysse Carion
Segment.io, Inc
100 California Street
San Francisco 94111
United States of America

Email: ulysse@segment.com