Building Blocks for HTTP APIs
Internet-Draft
Intended status: Standards Track
Expires: 7 June 2026

C. Vasters
Microsoft Corporation

4 December 2025

JSON Structure: Core
draft-vasters-json-structure-core-02

Abstract

This document specifies JSON Structure, a data structure definition language that enforces strict typing, modularity, and determinism. JSON Structure describes JSON-encoded data such that mapping to and from programming languages and databases and other data formats is straightforward.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://json-structure.github.io/core/draft-vasters-json-structure-core.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-vasters-json-structure-core/.

Source for this draft and an issue tracker can be found at https://github.com/json-structure/core.

Status of This Memo

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

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

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

This Internet-Draft will expire on 7 June 2026.

Table of Contents

1. Introduction

This document specifies _JSON Structure_, a data structure definition language that enforces strict typing, modularity, and determinism. _JSON Structure_ documents (schemas) describe JSON-encoded data such that mapping JSON encoded data to and from programming languages and databases and other data formats becomes straightforward.

JSON Structure is extensible, allowing additional features to be layered on top. The core language is a data-definition language.

The "Validation" and "Conditional Composition" extension specifications add rules that allow for complex pattern matching of _JSON Structure_ documents against JSON data for document validation purposes.

Complementing _JSON Structure_ are a set of extension specifications that extend the core schema language with additional, OPTIONAL features:

* _JSON Structure: Import_ [JSTRUCT-IMPORT]: Defines a mechanism for importing external schemas and definitions into a schema document.

* _JSON Structure: Alternate Names and Descriptions_ [JSTRUCT-ALTNAMES]: Provides a mechanism for declaring multilingual descriptions, and alternate names and symbols for types and properties.

* _JSON Structure: Symbols, Scientific Units, and Currencies_ [JSTRUCT-UNITS]: Defines annotation keywords for specifying symbols, scientific units, and currency codes complementing type information.

* _JSON Structure: Validation_ [JSTRUCT-VALIDATION]: Specifies extensions to the core schema language for declaring validation rules for JSON data that have no structural impact on the schema.

* _JSON Structure: Composition_ [JSTRUCT-COMPOSITION]: Defines a set of conditional composition rules for evaluating schemas.

These extension specifications are enabled by the extensibility (Section 3.10) features and can be applied to meta-schemas, schemas, and JSON document instances.

2. Conventions

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.

3. JSON Structure Core Specification

3.1. Schema Elements

3.1.1. Schema

A "schema" is a JSON object that describes, constrains, and interprets a JSON node.

This schema constrains a JSON node to be of type string:

```
{
  "name": "myname",
  "type": "string"
}
```

In the case of a schema that references a compound type (object, set, array, map, tuple, choice), the schema further describes the structure of the compound type. Schemas can be placed into a namespace (Section 3.3.1) for reuse in other schemas.

```
{
  "name": "myname",
  "type": "object",
  "properties": {
    "name": { "type": "string" }
  }
}
```

All schemas have an associated name that serves as an identifier. In the example above where the schema is a root object, the name is the value of the name property.

When the schema is placed into a namespace (Section 3.3.1) or embedded into a properties (Section 3.7.2) section of an object type, the name is the key under which the schema is stored.

Further rules for schemas are defined in Section 3.4.

A "schema document" is a schema that represents the root of a schema hierarchy and is the container format in which schemas are stored on disk or exchanged. A schema document MAY contain multiple type declarations and namespaces. The structure of schema documents is defined in Section 3.3.

JSON Structure is extensible. All keywords that are not explicitly defined in this document MAY be used for custom annotations and extensions. This also applies to keywords that begin with the $ character. A complete list of reserved keywords is provided in Section 4.

The semantics of keywords defined in this document MAY be expanded by extension specifications, but the core semantics of the keywords defined in this document MUST NOT be altered.

Be mindful that the use of custom keywords and annotations might conflict with future versions of this specification or other extensions and that the authors of this specification will not go out of their way to avoid such conflicts.

Section 3.10 details the extensibility features.

Formally, a schema is a constrained non-schema (Section 3.1.2) that requires a type (Section 3.7.1) keyword or a $ref (Section 3.3.6) keyword to be a schema.

3.1.2. Non-Schema

Non-schemas are objects that do not declare or refer to a type. The root of a schema document (Section 3.3) is a non-schema unless it contains a type keyword.

A namespace is a non-schema that contains type declarations and other namespaces.

3.1.3. Meta-Schemas

A meta-schema is a schema document that defines the structure and constraints of another schema document. Meta-schemas are used to validate schema documents and to ensure that schemas are well-formed and conform to the JSON Structure specification.

The meta-schemas for JSON Structure and the extension specifications are enumerated in the Appendix: Metaschemas (Section 3.1.1).

Meta-schemas can extend existing meta-schemas by adding new keywords or constraints. The $schema keyword is used to reference the meta-schema that a schema document conforms to, the $id keyword is used to define the identifier of the new meta-schema, and the $import keyword defined in the [JSTRUCT-IMPORT] extension specification is used to import all definitions from the foundational meta-schema.

3.2. Data Types

The data types that can be used with the type keyword are categorized into JSON primitive types, extended types, compound types, and reusable types Section 3.4.2.

While JSON Structure builds on the JSON data type model, it introduces a rich set of types to represent structured data more accurately and to allow more precise integration with common data types used in programming languages and data formats. All these extended types have a well-defined representation in JSON primitive types.

3.2.1. JSON Primitive Types

These types map directly to the underlying JSON representation:

3.2.1.1. string

A sequence of Unicode characters enclosed in double quotes.

* Base type: string Section 7 of [RFC8259]

* Annotations: The maxLength keyword can be used on a schema with the string type to specify the maximum length of the string. By default, the maximum length is unlimited. The purpose of the keyword is to inform consumers of the maximum space required to store the string.

3.2.1.2. number

A numeric literal without quotes.

* Base type: number Section 6 of [RFC8259]

Note that the number representation in JSON is a textual representation of a decimal number (base-10) and therefore cannot accurately represent all possible values of IEE754 floating-point numbers (base-2), in spite of JSON number leaning on the IEEE754 standard as a reference for the value space.

3.2.1.3. integer

An alias for int32 (Section 3.2.2.6), provided for compatibility with JSON Schema.

* Base type: number

* Constraints:

  - The numeric literal MUST be in the range -2 to 21.

  - No decimal points or quotes are allowed.

3.2.1.4. boolean

A literal true or false (without quotes).

* Base type: boolean Section 3 of [RFC8259]

3.2.1.5. null

A literal null (without quotes).

* Base type: null Section 3 of [RFC8259]

3.2.2. Extended Primitive Types

Extended types impose additional semantic constraints on the underlying JSON types. These types are used to represent binary data, high-precision numeric values, date and time information, and structured data.

Large integer and decimal types are used to represent high-precision numeric values that exceed the range of IEEE 754 double-precision format, which is the foundation for the number type in JSON. Per Section 6 of [RFC8259], interoperable JSON numbers have a range of -2 to 21, which is less than the range of 64-bit and 128-bit values. Therefore, the int64, uint64, int128, uint128, and decimal types are represented as strings to preserve precision.

The syntax for strings representing large integer and decimal types is based on the Section 6 of [RFC8259] syntax for integers and decimals:

* integer = [minus] int

* decimal = [minus] int frac

3.2.2.1. binary

A binary value. The default encoding is Base64 [RFC4648]. The type annotation keywords contentEncoding, contentCompression, and contentMediaType can be used to specify the encoding, compression, and media type of the binary data.

* Base type: string

* Constraints:

  - The string value MUST be an encoded binary value, with the encoding specified in the contentEncoding keyword.

3.2.2.2. int8

An 8-bit signed integer.

* Base type: number
* Constraints:
  - The numeric literal MUST be in the range -2 to 21.
  - No decimal points or quotes are allowed.

3.2.2.3. uint8

An 8-bit unsigned integer.

* Base type: number
* Constraints:
  - The numeric literal MUST be in the range 0 to 21.
  - No decimal points or quotes are allowed.

3.2.2.4. int16

A 16-bit signed integer.

* Base type: number
* Constraints:
  - The numeric literal MUST be in the range -2 to 21.
  - No decimal points or quotes are allowed.

3.2.2.5. uint16

A 16-bit unsigned integer.

* Base type: number
* Constraints:
  - The numeric literal MUST be in the range 0 to 21.
  - No decimal points or quotes are allowed.

3.2.2.6. int32

A 32-bit signed integer.

* Base type: number
* Constraints:
  - The numeric literal MUST be in the range -2 to 21.
  - No decimal points or quotes are allowed.

3.2.2.7. uint32

A 32-bit unsigned integer.

* Base type: number
* Constraints:
  - The numeric literal MUST be in the range 0 to 21.
  - No decimal points or quotes are allowed.

3.2.2.8. int64

A 64-bit signed integer.

* Base type: string
* Constraints:
  - The string MUST conform to the Section 6 of [RFC8259] definition for the [minus] int syntax.
  - The string value MUST represent a 64-bit integer in the range -2 to 21.

3.2.2.9. uint64

A 64-bit unsigned integer.

* Base type: string
* Constraints:
  - The string MUST conform to the Section 6 of [RFC8259] definition for the int syntax.

- The string value MUST represent a 64-bit integer in the range 0 to 21.

3.2.2.10. int128

A 128-bit signed integer.

* Base type: string

* Constraints:

  - The string MUST conform to the Section 6 of [RFC8259] definition for the [minus] int syntax.

  - The string value MUST represent a 128-bit integer in the range -2 to 21.

3.2.2.11. uint128

A 128-bit unsigned integer.

* Base type: string

* Constraints:

  - The string MUST conform to the Section 6 of [RFC8259] definition for the int syntax.

  - The string value MUST represent a 128-bit integer in the range 0 to 21.

3.2.2.12. float8

An 8-bit floating-point number.

* Base type: number

* Constraints:

  - Conforms to IEEE 754 single-precision value range limits (8 bits), which are 3 bits of significand and 4 bits of exponent, with a range of approximately ±3.4× 10.

3.2.2.13. float

A single-precision floating-point number.

* Base type: number

* Constraints:

  - Conforms to IEEE 754 single-precision value range limits (32 bits), which are 24 bits of significand and 8 bits of exponent, with a range of approximately ±3.4×10.

IEEE754 binary32 are base-2 encoded and therefore cannot represent all decimal numbers accurately, and vice versa. In cases where you need to encode IEEE754 values precisely, store the IEE754 binary32 value as an int32 or uint32 number.

3.2.2.14. double

A double-precision floating-point number.

* Base type: number

* Constraints:

  - Conforms to IEEE 754 double-precision value range limits (64 bits), which are 53 bits of significand and 11 bits of exponent, with a range of approximately ±1.7×10.

IEEE754 binary64 are base-2 encoded and therefore cannot represent all decimal numbers accurately, and vice versa. In cases where you need to encode IEEE754 values precisely, store the IEE754 binary64 value as an int64 or uint64 number.

3.2.2.15. decimal

A decimal number supporting high-precision values.

* Base type: string

* Constraints:

  - The string value MUST conform to the Section 6 of [RFC8259] definition for the [minus] int frac syntax.

  - Defaults: 34 significant digits and 7 fractional digits, which is the maximum precision supported by the IEEE 754 decimal128 format.

* Annotations:

  - The precision keyword MAY be used to specify the total number of significant digits.

- The scale keyword MAY be used to specify the number of fractional digits.

3.2.2.16. date

A date in YYYY-MM-DD form.

* Base type: string

* Constraints:

  - The string value MUST conform to the [RFC3339] full-date format.

3.2.2.17. datetime

A date and time value with time zone offset.

* Base type: string

* Constraints:

  - The string value MUST conform to the [RFC3339] date-time format.

3.2.2.18. time

A time-of-day value.

* Base type: string

* Constraints:

  - The string value MUST conform to the [RFC3339] time format.

3.2.2.19. duration

A time duration.

* Base type: string

* Constraints:

  - The string value MUST conform to the [RFC3339] duration format.

3.2.2.20. uuid

A universally unique identifier.

* Base type: string
* Constraints:
  - The string value MUST conform to the [RFC9562] UUID format.

3.2.2.21. uri

A URI reference, relative or absolute.

* Base type: string
* Constraints:
  - The string value MUST conform to the [RFC3986] uri-reference format.

3.2.2.22. jsonpointer

A JSON Pointer reference.

* Base type: string
* Constraints:
  - The string value MUST conform to the [RFC6901] JSON Pointer format.

3.2.3. Compound Types

Compound types are used to structure related data elements. JSON Structure supports the following compound types:

3.2.3.1. object

The object type is used to define structured data with named properties. It’s represented as a JSON object, which is an unordered collection of keyvalue pairs.

The object type MUST include a name attribute that defines the name of the type.

The object type MUST include a properties attribute that defines the properties of the object. The properties attribute MUST be a JSON object where each key is a property name and each value is a schema definition for the property. The object MUST contain at least one property definition.

The object type MAY include a required attribute that defines the required properties of the object.

The object type MAY include an additionalProperties attribute that defines whether additional properties are allowed and/or what their schema is.

Example:

```
{
  "name": "Person",
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "age": { "type": "int32" }
  },
  "required": ["name"],
  "additionalProperties": false
}
```

3.2.3.2. array

An array type is used to define an ordered collection of elements. It’s represented as a JSON array, which is an ordered list of values.

The items attribute of an array MUST reference a reusable type or a primitive type or a locally declared compound type.

Examples:

```
{
  "type": "array",
  "items": { "type": { "$ref": "#/definitions/Namespace/TypeName" } }
}
```

```
{
  "type": "array",
  "items": { "type": "string" }
}
```

3.2.3.3. set

The set type is used to define an unordered collection of unique elements. It's represented as a JSON array where all elements are unique.

The items attribute of a set MUST reference a reusable type or a primitive type or a locally declared compound type.

Example:

```
{
  "type": "set",
  "items": { "type": { "$ref": "#/definitions/Namespace/TypeName" } }
}

{
  "type": "set",
  "items": { "type": "string" }
}
```

3.2.3.4. map

The map type is used to define dynamic keyvalue pairs. It's represented as a JSON object where the keys are strings and the values are of a specific type.

Map keys MAY be any valid JSON string.

The values attribute of a map MUST reference a reusable type or a primitive type or a locally declared compound type.

Example:

```
{
  "type": "map",
  "values": { "type": { "$ref": "#/definitions/StringType" } }
}
```

3.2.3.5. tuple

The tuple type is used to define an ordered collection of elements with a specific length. It's represented as a JSON array where each element is of a specific type.

The elements are defined using a properties map as with the object (Section 3.2.3.1) type and each element is named. This permits straightforward mapping into application constructs. All declared properties of a tuple are implicitly REQUIRED.

The order of the elements in a tuple is declared using the tuple keyword Section 3.7.11, which is REQUIRED. The tuple keyword MUST be a JSON array of strings, where each declared property name MUST be an element of the array. The order of the elements in the array defines the order of the properties in the tuple.

A tuple type MUST include a name attribute that defines the name of the type.

Example:

```
{
  "type": "tuple",
  "name": "Person",
  "properties": {
    "name": { "type": "string" },
    "age": { "type": "int32" }
  },
  "tuple": ["name", "age"]
}
```

The following JSON node is an valid instance of the tuple type defined above:

```
["Alice", 42]
```

3.2.3.6. any

The any type is used to define a type that can be any JSON value, including primitive types, compound types, and extended types.

Example:

```
{
  "type": "any"
}
```

3.2.3.7. choice

The choice type is used to define a "discriminated union" of types. A choice is a set of types where only one type can be selected at a time and where the selected type is determined by the value of a selector.

The choice type can declare two variants of discriminated unions that are represented differently in JSON:

* _Tagged unions_: The choice type is represented as a JSON object with a single property whose name is the selector of the type as declared in the choices (Section 3.7.9) map and whose value is of the selected type.

* _Inline unions_: The choice type is represented as a JSON object of the selected type with the selector as a property of the object.

3.2.3.7.1. Tagged Unions

A tagged union is declared as follows:

```
{
  "type": "choice",
  "name": "MyChoice",
  "choices": {
    "string": { "type": "string" },
    "int32": { "type": "int32" }
  }
}
```

The JSON node described by the schema above is a tagged union. For the example, the following JSON node is a valid instance of the MyChoice type:

```
{
  "string": "Hello, world!"
}
```

or:

```
{
  "int32": 42
}
```

3.2.3.7.2. Inline Unions

Inline unions require for all type choices to extend a common base type.

This is expressed by using the $extends (Section 3.10.2) keyword in the choice declaration. The $extends keyword MUST refer to a schema that defines the base type and the base type MUST be abstract.

If $extends is present, the selector property declares the name of the injected property that acts as the selector for the inline union. The type of the selector property is string. The selector property MAY shadow a property of the base type; in this case, the base type property MUST be of type string.

The selector is defined as a property of the base type and the value of the selector property MUST be a string that matches the name of one of the options in the choices map.

Example:

```
{
  "$schema": "https://json-structure.org/meta/core/v0/#",
  "$id": "https://schemas.vasters.com/TypeName",
  "type": "choice",
  "$extends": "#/definitions/Address",
  "selector": "addressType",
  "choices": {
    "StreetAddress": { "type": { "$ref": "#/definitions/StreetAddress" } },
    "PostOfficeBoxAddress": { "type": { "$ref": "#/definitions/PostOfficeBoxAddress" }
}
  },
  "definitions" : {
    "Address": {
      "abstract": true,
      "type": "object",
      "properties": {
          "city": { "type": "string" },
          "state": { "type": "string" },
          "zip": { "type": "string" }
      }
    },
    "StreetAddress": {
      "type": "object",
      "$extends": "#/definitions/Address",
      "properties": {
          "street": { "type": "string" }
      }
    },
    "PostOfficeBoxAddress": {
      "type": "object",
      "$extends": "#/definitions/Address",
      "properties": {
          "poBox": { "type": "string" }
      }
    }
  }
}
```

The JSON node described by the schema above is an inline union. This example shows a JSON node that is a street address:

```
{
  "addressType": "StreetAddress",
  "street": "123 Main St",
  "city": "Seattle",
  "state": "WA",
  "zip": "98101"
}
```

This example shows a JSON node that is a post office box address:

```
{
  "addressType": "PostOfficeBoxAddress",
  "poBox": "1234",
  "city": "Seattle",
  "state": "WA",
  "zip": "98101"
}
```

3.3. Document Structure

A JSON Structure document is a JSON object that contains schemas (Section 3.1.1)

The root of a JSON Structure document MUST be a JSON object.

The root object MUST contain the following REQUIRED keywords:

* $id: A URI that is the unique identifier for this schema document.

* $schema: A URI that identifies the version and meta-schema of the JSON Structure specification used.

* name: A string that provides a name for the document. If the root object defines a type, the name attribute is also the name of the type.

The presence of both keywords identifies the document as a JSON Structure document.

The root object MAY contain the following OPTIONAL keywords:

* $root: A JSON Pointer that designates a reusable type as the root type for instances.

* definitions: The root of the type declaration namespace hierarchy.

* type: A type declaration for the root type of the document. Mutually exclusive with $root.

* if type is present, all annotations and constraints applicable to this declared root type are also permitted at the root level.

3.3.1. Namespaces

A namespace is a JSON object that provides a scope for type declarations or other namespaces. Namespaces MAY be nested within other namespaces.

The definitions keyword forms the root of the namespace hierarchy for reusable type definitions. All type declarations immediately under the definitions keyword are in the root namespace.

A type definition at the root is placed into the root namespace as if it were a type declaration under definitions.

Any object in the definitions map that is not a type declaration is a namespace.

Example with inline type:

```
{
    "$schema": "https://json-structure.org/meta/core/v0/#",
    "$id": "https://schemas.vasters.com/TypeName",
    "name": "TypeName",
    "type": "object",
    "properties": {
        "name": { "type": "string" }
    }
}
```

Example with $root:

```
{
    "$schema": "https://json-structure.org/meta/core/v0/#",
    "$id": "https://schemas.vasters.com/TypeName",
    "$root": "#/definitions/TypeName",
    "definitions": {
        "TypeName": {
            "type": "object",
            "properties": {
                "name": { "type": "string" }
            }
        }
    }
}
```

Example with the root type in a namespace:

```
{
    "$schema": "https://json-structure.org/meta/core/v0/#",
    "$id": "https://schemas.vasters.com/TypeName",
    "$root": "#/definitions/Namespace/TypeName",
    "definitions": {
        "Namespace": {
            "TypeName": {
                "name": "TypeName",
                "type": "object",
                "properties": {
                    "name": { "type": "string" }
                }
            }
        }
    }
}
```

3.3.2. $schema Keyword

The value of the REQUIRED $schema keyword MUST be an absolute URI. The keyword has different functions in JSON Structure documents and JSON documents.

* In JSON Structure schema documents, the $schema keyword references a meta-schema that this document conforms to.

* In JSON documents, the $schema keyword references a JSON Structure schema document that defines the structure of the JSON document.

The value of $schema MUST correspond to the $id of the referenced meta-schema or schema document.

The $schema keyword MUST be used at the root level of the document.

Example:

```
{
    "$schema": "https://json-structure.org/meta/core/v0/#",
    "name": "TypeName",
    "type": "object",
    "properties": {
        "name": { "type": "string" }
    }
}
```

Use of the keyword $schema does NOT import the referenced schema document such that its types become available for use in the current document.

3.3.3. $id Keyword

The REQUIRED $id keyword is used to assign a unique identifier to a JSON Structure schema document. The value of $id MUST be an absolute URI. It SHOULD be a resolvable URI (a URL).

The $id keyword is used to identify a schema document in references like $schema.

The $id keyword MUST only be used once in a document, at the root level.

Example:

```
{
    "$schema": "https://json-structure.org/meta/core/v0/#",
    "$id": "https://schemas.vasters.com/TypeName",
    "name": "TypeName",
    "type": "object",
    "properties": {
        "name": { "type": "string" }
    }
}
```

3.3.4. $root Keyword

The OPTIONAL $root keyword is used to designate any reusable type defined in the document as the root type of this schema document. The value of $root MUST be a valid JSON Pointer that resolves to an existing type definition inside the definitions object.

The $root keyword MUST only be used once in a document, at the root level. Its use is mutually exclusive with the type keyword.

Example:

```
{
  "$schema": "https://json-structure.org/meta/core/v0/#",
  "$id": "https://schemas.vasters.com/TypeName",
  "$root": "#/definitions/Namespace/TypeName",
  "definitions": {
      "Namespace": {
          "TypeName": {
            "name": "TypeName",
            "type": "object",
            "properties": {
                "name": { "type": "string" }
            }
          }
      }
  }
}
```

3.3.5. definitions Keyword

The definitions keyword defines a namespace hierarchy for reusable type declarations. The keyword MUST be used at the root level of the document.

The value of the definitions keyword MUST be a map of types and namespaces. The namespace at the root level of the definitions keyword is the root namespace.

A namespace is a JSON object that provides a scope for type declarations or other namespaces. Any JSON object under the definitions keyword that is not a type definition (containing the type attribute) is considered a namespace.

Example:

```
{
    "$schema": "https://json-structure.org/meta/core/v0/#",
    "$id": "https://schemas.vasters.com/TypeName",
    "definitions": {
        "Namespace": {
            "TypeName": {
                "name": "TypeName",
                "type": "object",
                "properties": {
                    "name": { "type": "string" }
                }
            }
        }
    }
}
```

3.3.6. $ref Keyword

References to type declarations within the same document MUST use a schema containing a single property with the name $ref as the value of type. The value of $ref MUST be a valid JSON Pointer Fragment Identifier (see Section 6 of [RFC6901]) that resolves to an existing type definition.

Example:

```
{
  "$schema": "https://json-structure.org/meta/core/v0/#",
  "$id": "https://schemas.vasters.com/TypeName",
  "properties": {
      "name1": { "type": { "$ref": "#/definitions/Namespace/TypeName" }},
      "name2": { "type": { "$ref": "#/definitions/Namespace2/TypeName2" }}
  },
  "definitions": {
      "Namespace": {
          "TypeName": {
              "name": "TypeName",
              "type": "object",
              "properties": {
                  "name": { "type": "string" }
              }
          }
      },
      "Namespace2": {
          "TypeName2": {
              "name": "TypeName2",
              "type": "object",
              "properties": {
                  "name": { "type": { "$ref": "#/definitions/Namespace/TypeName" }}
              }
          }
      }
  }
}
```

The $ref keyword is only permitted inside the type attribute value of a schema definition, including in type unions.

$ref is NOT permitted in other attributes and MUST NOT be used inside the type of the root object.

3.3.7. Cross-references

In JSON Structure documents, the $schema keyword references the meta-schema of this specification. In JSON documents, the $schema keyword references the schema document that defines the structure of the JSON document. The value of $schema is a URI. Ideally, the URI SHOULD be a resolvable URL to a schema document, but it’s primarily an identifier. As an identifier, it can be used as a lookup key in a cache or schema-registry.

The OPTIONAL [JSTRUCT-IMPORT] extension specification is the exception and provides a mechanism for importing definitions from external schemas.

3.4. Type System Rules

3.4.1. Schema Declarations

* Every schema element MUST declare a type referring to a primitive, compound, or reusable type.

* To reference a reusable type, the type attribute MUST be a schema with a single $ref property resolving to an existing type declaration.

* Compound types SHOULD be declared in the definitions section as reusable types. Inline compound types in arrays, maps, unions, or property definitions MUST NOT be referenced externally.

* Primitive and compound type declarations are confined to this specification.

* Defined types:

  - *JSON Primitives:* string, number, integer, boolean, null.

  - *Extended Primitives:* int32, uint32, int64, uint64, int128, uint128, float, double, decimal, date, datetime, time, duration, uuid, uri, binary, jsonpointer.

  - *JSON Compounds:* object, array.

  - *Extended Compounds:* map, set, tuple, any, choice.

3.4.2. Reusable Types

* Reusable types MUST be defined in the definitions section.

* Each declaration in definitions MUST have a unique, case-sensitive name within its namespace. The same name MAY appear in different namespaces.

3.4.3. Type References

* Use $ref to reference types declared in the same document.

* $ref MUST be a valid JSON Pointer to an existing type declaration.

* $ref MAY include a description attribute for additional context.

3.4.4. Dynamic Structures

* Use the map type for dynamic keyvalue pairs. The object type requires at least one property and cannot model fully dynamic properties with additionalProperties.

* The values attribute of a map and the items attribute of an array or set MUST reference a reusable type, a primitive type, or a locally declared compound type.

3.5. Composition Rules

This section defines the rules for composing schemas. Further, OPTIONAL composition rules are defined in the [JSTRUCT-COMPOSITION] extension specification.

3.5.1. Unions

* Non-discriminated type unions are formed as sets of primitive types and type references. It is NOT permitted to define a compound type inline inside a non-discriminated type union. Discriminated unions are formed as a choice (Section 3.2.3.7) type to which the rules of this section do not apply.

* A type union is a composite type reference and not a standalone compound type and is therefore not named.

* The JSON node described by a schema with a type union MUST conform to at least one of the types in the union.

* If the JSON node described by a schema with a type union conforms to more than one type in the union, the JSON node MUST be considered to be of the first matching type in the union.

Examples:

Union of a string and a compound type:

```
{
  "type": ["string", { "$ref": "#/definitions/Namespace/TypeName" } ]
}
```

Union of a string and an int32:

```
{
  "type": ["string", "int32"]
}
```

A valid union of a string and a map of strings:

```
{
  "type": ["string", { "type": "map", "values": { "type": "string" } } ]
}
```

An inline definition of a compound type in a union is NOT permitted:

```
{
  "type": ["string", { "type": "object", "properties": { "name": { "type": "string" }
} } ]
}
```

3.5.2. Prohibition of Top-Level Unions

* The root of a JSON Structure document MUST NOT be an array.

* If a type union is desired as the type of the root of a document instance, the $root keyword MUST be used to designate a type union as the root type.

3.6. Identifier Rules

All property names and type names MUST conform to the regular expression [A-Za-z_][A-Za-z0-9_]*. They MUST begin with a letter or underscore and MAY contain letters, digits, and underscores. Keys and type names are case-sensitive.

If names need to contain characters outside of this range, consider using the [JSTRUCT-ALTNAMES] extension specification to define those.

3.7. Structural Keywords

3.7.1. The type Keyword

Declares the type of a schema element as a primitive or compound type. The type keyword MUST be present in every schema element. For unions, the value of type MUST be an array of type references or primitive type names.

Example:

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

3.7.2. The properties Keyword

properties defines the properties of an object type.

The properties keyword MUST contain a map of property names mapped to schema definitions.

Example:

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "age": { "type": "int32" }
  }
}
```

3.7.3. The required Keyword

required defines the required properties of an object type. The required keyword MUST only be used in schemas of type object.

The value of the required keyword is a simple array of property names or an array of arrays of property names.

An array of arrays is used to define alternative sets of required properties. When alternative sets are used, exactly one of the sets MUST match the properties of the object, meaning they are mutually exclusive.

Property names in the required array MUST be present in properties.

Example:

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "age": { "type": "int32" }
  },
  "required": ["name"]
}
```

Example with alternative sets:

Because the name property is required in both sets, the name property is required in all objects. The fins property is required in the first set, and the legs property is required in the second set. That means that an object MUST have either fins or legs but not both.

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "fins": { "type": "int32" },
    "legs": { "type": "int32" },
    "wings": { "type": "int32" }
  },
  "required": [["name", "fins"], ["name", "legs"]]
}
```

3.7.4. The items Keyword

Defines the schema for elements in an array or set type. The value is a type reference or a primitive type name or a locally declared compound type.

Examples:

```
{
  "type": "array",
  "items": { "type": { "$ref": "#/definitions/Namespace/TypeName" }}
}
```

```
{
  "type": "array",
  "items": { "type": "string" }
}
```

3.7.5. The values Keyword

Defines the schema for values in a map type.

The values keyword MUST reference a reusable type or a primitive type or a locally declared compound type.

Example:

```
{
  "type": "map",
  "values": { "type": "string" }
}
```

3.7.6. The const Keyword

Constrains the values of the JSON node described by the schema to a single, specific value. The const keyword MUST appear only in schemas with a primitive type, and the instance value MUST match the provided constant exactly.

Example:

```
{
  "type": "string",
  "const": "example"
}
```

3.7.7. The enum Keyword

Constrains a schema to match one of a specific set of values. The enum keyword MUST appear only in schemas with a primitive type, and all values in the enum array MUST match that type. Values MUST be unique.

Example:

```
{
  "type": "string",
  "enum": ["value1", "value2", "value3"]
}
```

It is NOT permitted to use enum in conjunction with a type union in type.

3.7.8. The additionalProperties Keyword

additionalProperties defines whether additional properties are allowed in an object type and, optionally, what their schema is. The value MUST be a boolean or a schema. If set to false, no additional properties are allowed. If provided with a schema, each additional property MUST conform to it.

Example:

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string" }
  },
  "additionalProperties": false
}
```

3.7.9. The choices Keyword

choices defines the choices of a choice type. The value MUST be a map of type names to schemas. Each type name MUST be unique within the choices map.

The value of each type name MUST be a schema. Inline compound types are permitted.

The choices keyword MUST only be used in schemas of type choice (Section 3.2.3.7).

**Example*:

```
{
  "type": "choice",
  "name": "MyChoice",
  "choices": {
    "string": { "type": "string" },
    "int32": { "type": "int32" }
  }
}
```

3.7.10. The selector Keyword

The selector keyword defines the name of the property that acts as the selector for the type in a choice type. The value of selector MUST be a string.

The selector keyword MUST only be used in schemas of type choice (Section 3.2.3.7).

See choice (Section 3.2.3.7) for an example.

3.7.11. The tuple Keyword

The tuple keyword defines the order of properties in a tuple type. The value of tuple MUST be an array of strings, where each string is the name of a property defined in the properties map. The order of the strings in the array defines the order of the properties in the tuple.

The tuple keyword MUST only be used in schemas of type tuple (Section 3.2.3.5).

See tuple (Section 3.2.3.5) for an example.

3.8. Type Annotation Keywords

Type annotation keywords provide additional metadata about the underlying type. These keywords are used for documentation and validation of additional constraints on types.

3.8.1. The maxLength Keyword

Specifies the maximum allowed length for a string. The maxLength keyword MUST be used only with string types, and the string's length MUST not exceed this value.

The purpose of maxLength is to provide a known storage constraint on the maximum length of a string. The value MAY be used for validation.

Example:

```
{
  "type": "string",
  "maxLength": 255
}
```

3.8.2. The precision Keyword

Specifies the total number of significant digits for numeric values. The precision keyword is used as an annotation for number or decimal types.

Example:

```
{
  "type": "decimal",
  "precision": 10
}
```

3.8.3. The scale Keyword

Specifies the number of digits to the right of the decimal point for numeric values. The scale keyword is used as an annotation for number or decimal types to constrain the fractional part.

Example:

```
{
  "type": "decimal",
  "scale": 2
}
```

3.8.4. The contentEncoding Keyword

Specifies the encoding of a binary value. The contentEncoding keyword is used as an annotation for binary types.

The permitted values for contentEncoding are defined in [RFC4648]:

* base64: The binary value is encoded as a base64 string.

* base64url: The binary value is encoded as a base64url string.

* base16: The binary value is encoded as a base16 string.

* base32: The binary value is encoded as a base32 string.

* base32hex: The binary value is encoded as a base32hex string.

Example:

```
{
  "type": "binary",
  "encoding": "base64"
}
```

3.8.5. The contentCompression Keyword

Specifies the compression algorithm used for a binary value before encoding. The contentCompression keyword is used as an annotation for binary types.

The permitted values for contentCompression are:

* gzip: The binary value is compressed using the gzip algorithm. See [RFC1952].

* deflate: The binary value is compressed using the deflate algorithm. See [RFC1951].

* zlib: The binary value is compressed using the zlib algorithm. See [RFC1950].

* brotli: The binary value is compressed using the brotli algorithm. See [RFC7932].

Example:

```
{
  "type": "binary",
  "encoding": "base64",
  "compression": "gzip"
}
```

3.8.6. The contentMediaType Keyword

Specifies the media type of a binary value. The contentMediaType keyword is used as an annotation for binary types.

The value of contentMediaType MUST be a valid media type as defined in [RFC6838].

Example:

```
{
  "type": "binary",
  "encoding": "base64",
  "mediaType": "image/png"
}
```

3.9. Documentation Keywords

Documentation keywords provide descriptive information for schema elements. They are OPTIONAL but RECOMMENDED for clarity.

3.9.1. The description Keyword

Provides a human-readable description of a schema element. The description keyword SHOULD be used to document any schema element.

Example:

```
{
  "type": "string",
  "description": "A person’s name"
}
```

For multi-lingual descriptions, the [JSTRUCT-ALTNAMES] companion provides an extension to define several concurrent descriptions in multiple languages.

3.9.2. The examples Keyword

Provides example instance values that conform to the schema. The examples keyword SHOULD be used to document potential instance values.

Example:

```
{
  "type": "string",
  "examples": ["example1", "example2"]
}
```

3.10. Extensions and Add-Ins

The abstract and $extends keywords enable controlled type extension, supporting basic object-oriented-programming-style inheritance while not permitting subtype polymorphism where a sub-type value can be assigned a base-typed property. This approach avoids validation complexities and mapping issues between JSON schemas, programming types, and databases.

An _extensible type_ is declared as abstract and serves as a base for extensions. For example, a base type _Address_ MAY be extended by _StreetAddress_ and _PostOfficeBoxAddress_ via $extends, but _Address_ cannot be used directly.

Example:

```
{
  "$schema": "https://json-structure.org/meta/core/v0/#",
  "definitions" : {
    "Address": {
      "abstract": true,
      "type": "object",
      "properties": {
          "city": { "type": "string" },
          "state": { "type": "string" },
          "zip": { "type": "string" }
      }
    },
    "StreetAddress": {
      "type": "object",
      "$extends": "#/definitions/Address",
      "properties": {
          "street": { "type": "string" }
      }
    },
    "PostOfficeBoxAddress": {
      "type": "object",
      "$extends": "#/definitions/Address",
      "properties": {
          "poBox": { "type": "string" }
      }
    }
  }
}
```

A _add-in type_ is declared as abstract and $extends a specific type that does not need to be abstract. For example, an add-in type _DeliveryInstructions_ might be applied to any _StreetAddress_ types in a document:

```
{
    "$schema": "https://json-structure.org/meta/core/v0/#",
    "$id": "https://schemas.vasters.com/Addresses",
    "$root": "#/definitions/StreetAddress",
    "$offers": {
        "DeliveryInstructions": "#/definitions/DeliveryInstructions"
    },
    "definitions" : {
      "StreetAddress": {
        "type": "object",
        "properties": {
            "street": { "type": "string" },
            "city": { "type": "string" },
            "state": { "type": "string" },
            "zip": { "type": "string" }
        }
      },
      "DeliveryInstructions": {
        "abstract": true,
        "type": "object",
        "$extends": "#/definitions/StreetAddress",
        "properties": {
            "instructions": { "type": "string" }
        }
      }
    }
}
```

Add-in types are options that a document author can enable for a schema. The definitions of add-in types are not part of the main schema by default, but are injected into the designated schema type when the document author chooses to use them.

Add-in types are advertised in the schema document through the $offers keyword, which is a map that defines add-in names for add-in schema definitions that exist in the document.

Add-ins are applied to a schema by referencing the add-in name in the $uses keyword that is available only in instance documents. The $uses keyword is a set of add-in names that are applied to the schema for the document.

```
{
  "$schema": "https://schemas.vasters.com/Addresses",
  "$uses": ["DeliveryInstructions"],
  "street": "123 Main St",
  "city": "Anytown",
  "state": "QA",
  "zip": "00001",
  "instructions": "Leave at the back door"
}
```

3.10.1. The abstract Keyword

The abstract keyword declares a type as abstract. This prohibits its direct use in any type declaration or as the type of a schema element. Abstract types are used as base types for extension via $extends or as add-in types via $addins.

Abstract types implicitly permit additional properties (additionalProperties is always true).

* *Value*: A boolean (true or false).

* *Rules*:

  - The abstract keyword MUST only be used in schemas of type object and tuple.

  - Abstract types MUST NOT be used as the type of a schema element or referenced via $ref.

  - The additionalProperties keyword MUST NOT be used on abstract types (its value is implicitly true).

  - Abstract types MAY extend other abstract types via $extends.

3.10.2. The $extends Keyword

The $extends keyword merges all properties from one or more abstract base types into the extending type.

If the type using $extends is marked as abstract and referenced via $addins, the composite type _replaces_ the base type in the type model of the document.

* *Value*: A JSON Pointer to an abstract type, or an array of JSON Pointers to abstract types.

* *Rules*:

- The $extends keyword MUST only be used in schemas of type object and tuple.

- The value of $extends MUST be a valid JSON Pointer or an array of valid JSON Pointers that point to abstract types within the same document.

- The extending type MUST merge the abstract type's properties and constraints and MUST NOT redefine any inherited property.

- When multiple base types are specified, properties are merged in array order. If multiple base types define a property with the same name, the property from the first base type in the array takes precedence.

3.10.3. The $offers Keyword

The $offers keyword is used to advertise add-in types that are available for use in a schema document. The $offers keyword is a map of add-in names to add-in schema definitions.

* *Value*: A map of add-in names to add-in schema definitions.

* *Rules*:

  - The $offers keyword MUST only be used in the root object of a schema document.

  - The value of $offers MUST be a map where each key is a string and each value is a JSON Pointer to an add-in schema definition in the same document or a set of JSON Pointers to add-in schema definitions in the same document. If the value is a set, the add-in name selects all add-in schema definitions at the same time.

  - The keys in the $offers map MUST be unique.

3.10.4. The $uses Keyword

The $uses keyword is used to apply add-in types to a schema _in an instance document_ that references the schema. The keyword MAY be used in a meta-schema that references a parent schema.

* *Value*: A set of add-in names or JSON Pointers to add-in schema definitions in the same meta-schema document.

* *Rules*:

- The $uses keyword MUST only be used in instance documents.

- The value of $uses MUST be a set of strings that are either:

  o add-in names advertised in the $offers keyword of the schema document referenced by the $schema keyword of the instance document or

  o JSON Pointers to add-in schema definitions in the same meta-schema document.

4. Reserved Keywords

The following keywords are reserved in JSON Structure and MUST NOT be used as custom annotations or extension keywords:

* definitions
* $extends
* $id
* $ref
* $root
* $schema
* $uses
* $offers
* abstract
* additionalProperties
* choices
* const
* default
* description
* enum
* examples

* format
* items
* maxLength
* name
* precision
* properties
* required
* scale
* selector
* type
* values

5. CBOR Type System Mapping

CBOR [RFC8949] is a binary encoding of JSON-like data structures. The CBOR type system is a superset of the JSON type system and adds "binary strings" as its most substantial type system extension. Otherwise, CBOR is structurally compatible with JSON.

JSON Structure MAY be used to describe CBOR-encoded data structures. For encoding CBOR data structures, the data structure is first mapped to a JSON type model as described in this specification, with the exception that the Section 3.2.2.1 primitive type is preserved as a byte array. The resulting mapping is converted into CBOR per the rules spelled out in Section 6.2 of [RFC8949].

The decoding process is the reverse of the encoding process. The CBOR-encoded data structure is first decoded into a JSON type model, and then the JSON type model is validated against the JSON Structure schema, with binary types validated as byte arrays.

6. Media Type

The media type for JSON Structure documents is application/json-structure.

It is RECOMMENDED to append the structured syntax suffix +json to indicate unambiguously that the content is a JSON document, if the document is a JSON document. In spite of this specification being focused on JSON, the JSON Structure documents MAY be encoded using other serialization formats that can represent the same data structure, such as CBOR [RFC8949].

* Type name: application
* Subtype name: json-structure
* Required parameters: none
* Optional parameters: none
* Encoding considerations: binary
* Security considerations: see Section 8
* Interoperability considerations: none
* Published specification: this document
* Applications that use this media type: none
* Fragment identifier considerations: none
* Additional information: none

7. Media Type Parameters

While the media type application/json-structure does not have any parameters, this specification defines a parameter applicable to all JSON documents.

7.1. schema Parameter

The schema parameter is used to reference a JSON Structure document that defines the structure of the JSON document. The value of the schema parameter MUST be a URI that references and ideally resolves to a JSON Structure document.

The schema parameter MAY be used in conjunction with the application/json media type or the +json structured syntax suffix or any other media type that is known to be encoded as JSON.

Example using the HTTP Content-Type header:

```
Content-Type: application/json; schema="https://schemas.vasters.com/TypeName"
```

8. Security Considerations

JSON Structure documents are self-contained and MUST NOT allow external references except for the $schema and $addins keywords. Implementations MUST ensure that all $ref pointers resolve within the same document to eliminate security vulnerabilities related to external schema inclusion.

9. IANA Considerations

IANA shall be requested to register the media type application/json-structure as defined in this specification in the "Media Types" registry.

IANA shall be requested to register the parameter schema for the application/json media type in the "Media Type Structured Syntax Suffixes" registry.

10. References

10.1. 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/rfc/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/rfc/rfc3339>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/rfc/rfc3986>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/rfc/rfc4648>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/rfc/rfc6838>.

[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/rfc/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/rfc/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/rfc/rfc8259>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/rfc/rfc8949>.

[RFC9562] Davis, K., Peabody, B., and P. Leach, "Universally Unique IDentifiers (UUIDs)", RFC 9562, DOI 10.17487/RFC9562, May 2024, <https://www.rfc-editor.org/rfc/rfc9562>.

10.2. Informative References

[JSTRUCT-ALTNAMES]
Vasters, C., "JSON Structure Alternate Names", n.d., <https://json-structure.github.io/alternate-names>.

[JSTRUCT-COMPOSITION]
Vasters, C., "JSON Structure Conditional Composition", n.d., <https://json-structure.github.io/conditional-composition>.

[JSTRUCT-IMPORT]
Vasters, C., "JSON Structure Import", n.d., <https://json-structure.github.io/import>.

[JSTRUCT-UNITS]
Vasters, C., "JSON Structure Units", n.d., <https://json-structure.github.io/units>.

[JSTRUCT-VALIDATION]
Vasters, C., "JSON Structure Validation", n.d., <https://json-structure.github.io/validation>.

[RFC1950] Deutsch, P. and J. Gailly, "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, DOI 10.17487/RFC1950, May 1996, <https://www.rfc-editor.org/rfc/rfc1950>.

[RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, DOI 10.17487/RFC1951, May 1996, <https://www.rfc-editor.org/rfc/rfc1951>.

[RFC1952] Deutsch, P., "GZIP file format specification version 4.3", RFC 1952, DOI 10.17487/RFC1952, May 1996, <https://www.rfc-editor.org/rfc/rfc1952>.

[RFC7932] Alakuijala, J. and Z. Szabadka, "Brotli Compressed Data Format", RFC 7932, DOI 10.17487/RFC7932, July 2016, <https://www.rfc-editor.org/rfc/rfc7932>.

Changes from draft-vasters-json-structure-core-01

* Fixed $ref paths in examples to correctly include #/definitions/ prefix for types defined in the definitions section.

Changes from draft-vasters-json-structure-core-00

* Added integer as an alias for int32 to improve compatibility with simple JSON Schema definitions and user habits.

* Updated the $extends keyword to accept either a single JSON Pointer or an array of JSON Pointers, enabling multiple inheritance for object and tuple types.

* Fixed inconsistent $ref usage in examples throughout the document.

Acknowledgments

TODO acknowledge.

Author’s Address

Clemens Vasters
Microsoft Corporation
Email: clemensv@microsoft.com