json-schema.org

Common interfaces across JSON Schema implementations

13-16 minutos

JSON Schema is extremely widely used and nearly equally widely implemented. There are implementations of JSON Schema validation for over 20 languages or environments. This prevalence is fantastic for user choice -- as someone wishing to use JSON Schema you can be almost certain you'll find an implementation suitable for your target environment.

But when it comes to community support, it can be challenging to know how to perform various JSON Schema operations in a particular library or implementation, as each may have slightly differing (or more than slightly differing) APIs. This can become particularly challenging when considering behavior not necessarily prescribed or required by the specification itself, but which is nonetheless common, useful, or even required behavior *in practice* in order to use JSON Schema. This page serves to document a number of these *common operations*, documenting the interfaces offered by many implementations across the existing JSON Schema ecosystem.

For each, we first name and describe the operation using terminology from the specification where available, or otherwise using terminology that is aimed to be succinct but precise. The intention of this page is partially to be a reference of important JSON Schema operations as well as a way of tailoring help for a particular language and implementation. We therefore include examples of each interface or piece of functionality across implementations, when known. The ordering of sections on this page is not necessarily indicative of their importance relative to each other. Omission of functionality (i.e. if you do not find the way to do something below with a given implementation) is *not* necessarily a sign that the implementation does not support the functionality, though it might mean it could not be easily found by the page authors. Contributions and corrections from implementers are particularly

welcome, as are documentation links!

For purposes of making some sections of this page more precise, we assume the existence of a set of *abstract types* all of which may be referenced below, and which will have specific concrete types within a given programming language or implementation. In addition to placeholder types like `String`, `Number`, `Boolean`, `Mapping`, `Callable` and the like, the JSON Schema-related types include:

`Schema`

The type of JSON Schemas, which may also differ across dialects of JSON Schema. For common or modern versions of JSON Schema this type must essentially be able to represent both arbitrary JSON objects as well as booleans.

`Instance`

The type of JSON instances. This type should essentially be `Any` or a type capable of representing all JSON values, given JSON Schema's applicability to any representable JSON.

`Dialect`

The type of JSON dialects or dialect identifiers. This type may simply be `String` if the dialect is represented within the implementation by its URI or by some short name.

`EvaluationOptions`

The type of "fully ready" schemas and instances *along* with any additional implementation-specific customizable behavior. At minimum, this type is either `Schema → Instance` or `Schema × Instance` (depending on whether the implementation takes both schema and instance together or compiles schemas and then produces a separate function taking the instance to validate). It is highly likely to be richer in at least some of the following ways:

- Given likely support for some form of schema registry in order to support referencing external schemas, this type will likely include a registry of some sort, i.e. `Mapping<URI, Schema> → Schema → ...`

- If an implementation supports customizing which JSON types match to which language types (such as discussed below) then this type likely includes some representation of this mapping, i.e. `Mapping<String, Callable[...]> → Schema → ...` encapsulating what each type is mapped to during this evaluation.
- Similarly, if there is a specific API for format assertion enablement, some representation of the `format` keyword's behavior is present in the context
- If the implementation supports the creation or customization of dialects, and especially if schemas can contain subschemas across different dialects, then the context will contain some representation of dialects, e.g. `Dialect → Schema → ...`

`Result`

The type of JSON Schema validation results, i.e. an object which encapsulates the validity of a given `Instance` under a `Schema`. This type may simply be `Boolean` in some implementations or languages.

`ResultWithAnnotations`

The type of JSON Schema validation results that *include* JSON Schema annotations collected during validation.

`URI`

The type of RFC 3986 compliant URIs. This type should not generally be the same as `String`, as URIs have multiple possible string representations and require normalization to have correct semantics.

Because this page deals with concerns which cross language barriers, and different programming languages have different capabilities particularly around how they represent out of band errors (via exceptions, option types, wrapper return types, sentinel values or some other mechanism), this page also introduces[1] specific notation for representing the types of functions which include explicit representation for their out-of-band errors. Interpreting these types will depend on the programming language. It's easier to explain the above with an example:

Consider a division function on floats, represented as `x / y`.

We will write the type of this function as `Float → Float → Float <!> DivideByZeroError`, where the interpretation of this signature is that the function takes 2 floats and returns a float, but the `<!>` signals it may also propagate a `DivideByZeroError` (in this case when the function is asked to divide by zero).

The specific manifestation of `DivideByZeroError` will depend on the programming language. In a language with exceptions, this function may raise a `DivideByZeroError`-equivalent as an exception, which must or may be handled by the caller. In a language with option types, the "correct" type signature may really be wrapped in an `Option` type which represents the division by zero case (so the "true" signature in such a language would be `Option[Float → Float → Float]`). In a language with wrapper return types, the true type signature may be `Result<Float, DivideByZeroError>` where the returned value must be inspected to ensure it contains a successful result, and where the divide by zero error is instead a possible error value. In a language with a convention to return "junk" values, the true type may be precisely `Float → Float → Float` where some arbitrary float value is returned when dividing by zero, with no further indication.

In all cases above, when an exception (corresp. error or junk value) is raised (corresp. returned), we by convention assume that any normal return value is either completely not present by the mechanisms of the programming language, or else is considered meaningless.

In addition, this page will not, in general, consider or mention the possibility that an error might be raised for reasons other than those discussed in a particular section. For example, the division example mentioned above may raise other kinds of errors if provided with strings, in a dynamically typed language where such possibilities exist at runtime, such that the "real" division function may look more like `Float → Float → Float <!> DivideByZeroError | TypeError` or even further error possibilities. In the case of JSON Schema, this may mean every interface mentioned below has a type containing `<!> InvalidSchema | NotValidJSON | ...` representing cases where they are provided with schemas that are not valid according to the specification, or which do not fit the JSON data model, etc., but we consider these to be implicit and will not mention them explicitly unless directly relevant to the interface being discussed.

Instance Validation

Instance validation is one of the key capabilities of JSON Schema. It is the process in which a given piece of data is deemed valid or invalid under a specific schema. Implementations may offer one or more of the specific interfaces below in order to perform this validation.

Exception-Driven Validation

Type: `EvaluationOptions → None <!> ValidationError` or `EvaluationOptions → Result <!> ValidationError`

A validation API which causes a language-specific failure or exception when validation itself fails. If it succeeds, this API may return a result with further detail, or may simply continue execution silently.

Boolean Validation

Type: `EvaluationOptions` → Boolean

An API which produces a simple boolean result indicating an instance's validity under a schema.

Two-Argument Validation

Type: `Schema` × `Instance` → `Result`

An API which takes a schema and instance simultaneously and produces a result indicating whether the instance is valid under the given schema.

In some sense, two argument validation is the simplest possible API for JSON Schema validation; it asks simply whether a given instance is valid under a given schema with no presumptions of repeated use (of the schema) or additional calculation.

Repeated Validation / Schema Compilation

Type: `Schema` → `Instance` → `Result`

An API which attempts to prepare a schema for repeated use. It may (and likely will) perform some form of preoptimization, performing some set of work ahead of time such that it will not be necessary to repeat when validating many instances.

Subschema Validation

Type: `Schema` × `String` × `Instance` → `Result`

An API which supports validating an instance against a *subschema* contained within the given schema.

The subschema is typically identified via a JSON Pointer, or equivalent syntax, and is as opposed to validation using a new schema referencing the subschema via the `$ref` keyword.

Annotation Collection

Type: `EvaluationOptions` → `ResultWithAnnotations`

An API which collects annotations produced when processing a given schema and instance.

Schema Validation

Type: `Schema` → `Result`

An API which validates a schema itself under the dialect it is written for. This API likely makes use of a corresponding metaschema (or metaschemas) for the dialect, but generally must do additional work to ensure the schema is not invalid even under conditions not checked for within the metaschema.

Explicit Version Selection

Type: `Dialect` → `EvaluationOptions` → `Result`

An API which controls which dialect the implementation will assume when given a schema which does not otherwise indicate its dialect (i.e. which does not declare a `$schema` property).

Version Detection

Type: `Schema` → `Dialect`

An API which identifies which dialect a given schema is written for, returning either the dialect itself or otherwise a dialect-specific validation function.

Type Customization

An API which allows (re-)configuring which language-level types correspond to which JSON Schema types, and additionally potentially allowing for the definition of new types.

String Validation

Type: `String` → `String` → `Result`

An API which directly validates instances using schemas where both are represented as strings -- serialized JSON -- as opposed to deserialized JSON.

Language Object Validation

Type: `EvaluationOptions` → `Result`

An API which validates instances using schemas where both have been deserialized into language-level objects, or potentially built up programmatically directly as language-level objects.

`format` Validation

Format Assertion Enablement

An API which configures the behavior of the `format` keyword, in dialects where this behavior is not solely controlled by JSON Schema vocabulary enablement.

Format Registration

Type: `String` × (`Instance` → Boolean)→EvaluationOptions`

An API which allows registering a *new* format and its implementation for use with the `format` keyword, typically when it is used to assert.

Some implementations may further allow registering different sets of available formats for different dialects.

Format Querying

An API which queries or lists the set of available formats which have been registered.

Direct Format Validation

Type: `String` × `Instance` → `Boolean`

An API which allows directly checking whether a JSON-like value satisfies a specific format, without the presence of a schema.

Schema Registry Population

An API which configures the bundle of schemas available for referencing during the validation process.

Externally Identified Schemas

An API which allows associating one or more URIs with a schema where the URIs are not internally indicated by an `$id` keyword (or a dialect-specific equivalent).

Internally Identified Schemas

An API which allows associating URIs with a schema where the URI is internally indicated by the presence of an `$id` keyword (or a dialect-specific equivalent) in the schema.

Schema Discovery

An API which (recursively) crawls a root schema or schemas, discovering any subresources (subschemas) which are present and identifiable, and making those subresources' URIs available for further referencing.

Dynamic URI Resolution

An API which allows for arbitrary dynamic lookup behavior for any reference not prepopulated in the registry.

Output Format Selection / Generation

An API which configures which of JSON Schema's output format(s) are used when producing results, or which allows generating a particular output format given a `Result`.

Vocabulary Registration

An API which facilitates the creation of new JSON Schema vocabularies, typically for use when later building new dialects.

Dialect Creation

An API which facilitates the creation of new JSON Schema dialects, typically registering them in some way such that they can be further used inside the implementation.

Failure Details

An API which allows for programmatic introspection of the causes of a particular validation failure, at minimum containing the failing keyword(s), value(s) and individual message(s) which led to the failure.