referencing

Release 0.34.0

Julian Berman

Mar 16, 2024

CONTENTS

An implementation-agnostic implementation of JSON reference resolution.

In other words, a way for e.g. JSON Schema tooling to resolve the $ref keywords across all drafts without needing to implement support themselves.

This library is meant for use both by implementers of JSON referencing-related tooling – like JSON Schema implementations supporting the $ref keyword – as well as by end-users using said implementations who wish to then configure sets of resources (like schemas) for use at runtime.

The simplest example of populating a registry (typically done by end-users) and then looking up a resource from it (typically done by something like a JSON Schema implementation) is:

```
from referencing import Registry, Resource
import referencing.jsonschema

schema = Resource.from_contents(  # Parse some contents into a 2020-12 JSON Schema
    {
        "$schema": "https://json-schema.org/draft/2020-12/schema",
        "$id": "urn:example:a-202012-schema",
        "$defs": {
            "nonNegativeInteger": {
                "$anchor": "nonNegativeInteger",
                "type": "integer",
                "minimum": 0,
            },
        },
    }
)
registry = schema @ Registry()  # Add the resource to a new registry

# From here forward, this would usually be done within a library wrapping this one,
# like a JSON Schema implementation
resolver = registry.resolver()
resolved = resolver.lookup("urn:example:a-202012-schema#nonNegativeInteger")
assert resolved.contents == {
    "$anchor": "nonNegativeInteger",
    "type": "integer",
    "minimum": 0,
}
```

For fuller details, see the *Introduction*.

When authoring JSON documents, it is often useful to be able to reference other JSON documents, or to reference subsections of other JSON documents.

This kind of JSON referencing has historically been defined by various specifications, with slightly differing behavior. The JSON Schema specifications, for instance, define $ref and $dynamicRef keywords to allow schema authors to combine multiple schemas together for reuse or deduplication as part of authoring JSON schemas.

The *referencing* library was written in order to provide a simple, well-behaved and well-tested implementation of JSON reference resolution in a way which can be used across multiple specifications or specification versions.

CHAPTER

ONE

CORE CONCEPTS

There are 3 main objects to be aware of:

- *referencing.Registry*, which represents a specific immutable set of resources (either in-memory or retrievable)
- *referencing.Specification*, which represents a specific specification, such as JSON Schema Draft 7, which can have differing referencing behavior from other specifications or even versions of JSON Schema. JSON Schema-specific specifications live in the *referencing.jsonschema* module and are named like *referencing.jsonschema.DRAFT202012*.
- *referencing.Resource*, which represents a specific resource (often a Python `dict`) *along* with a specific *referencing.Specification* it is to be interpreted under.

As a concrete example, the simple JSON Schema `{"type": "integer"}` may be interpreted as a schema under either Draft 2020-12 or Draft 4 of the JSON Schema specification (amongst others); in draft 2020-12, the float `2.0` must be considered an integer, whereas in draft 4, it potentially is not. If you mean the former (i.e. to associate this schema with draft 2020-12), you'd use `referencing.Resource(contents={"type": "integer"}, specification=referencing.jsonschema.DRAFT202012)`, whereas for the latter you'd use *referencing.jsonschema.DRAFT4*.

A resource may be identified via one or more URIs, either because they identify themselves in a way proscribed by their specification (e.g. an `$id` keyword in suitable versions of the JSON Schema specification), or simply because you wish to externally associate a URI with the resource, regardless of a specification-specific way to refer to itself. You could add the aforementioned simple JSON Schema resource to a *referencing.Registry* by creating an empty registry and then identifying it via some URI:

```
from referencing import Registry, Resource
from referencing.jsonschema import DRAFT202012
resource = Resource(contents={"type": "integer"}, specification=DRAFT202012)
registry = Registry().with_resource(uri="http://example.com/my/resource",␣
↪resource=resource)
print(registry)
```

```
<Registry (1 uncrawled resource)>
```

Note: *referencing.Registry* is an entirely immutable object. All of its methods which add resources to itself return *new* registry objects containing the added resource.

You could also confirm your resource is in the registry if you'd like, via *referencing.Registry.contents*, which will show you the contents of a resource at a given URI:

```
print(registry.contents("http://example.com/my/resource"))
```

```
{'type': 'integer'}
```

CHAPTER

TWO

POPULATING REGISTRIES

There are a few different methods you can use to populate registries with resources. Which one you want to use depends on things like:

- do you already have an instance of *referencing.Resource*, or are you creating one out of some loaded JSON? If not, does the JSON have some sort of identifier that can be used to determine which specification it belongs to (e.g. the JSON Schema $schema keyword)?
- does your resource have an internal ID (e.g. the JSON Schema $id keyword)?
- do you have additional (external) URIs you want to refer to the same resource as well?
- do you have one resource to add or many?

We'll assume for example's sake that we're dealing with JSON Schema resources for the following examples, and we'll furthermore assume you have some initial *referencing.Registry* to add them to, perhaps an empty one:

```
from referencing import Registry
initial_registry = Registry()
```

Recall that registries are immutable, so we'll be "adding" our resources by creating new registries containing the additional resource(s) we add.

In the ideal case, you have a JSON Schema with an internal ID, and which also identifies itself for a specific version of JSON Schema e.g.:

```
{
  "$id": "urn:example:my-schema",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "integer"
}
```

If you have such a schema in some JSON text, and wish to add a resource to our registry and be able to identify it using its internal ID (`urn:example:my-schema`) you can simply use:

```
import json

loaded = json.loads(
    """
    {
      "$id": "urn:example:my-schema",
      "$schema": "https://json-schema.org/draft/2020-12/schema",
      "type": "integer"
    }
    """,
```

(continues on next page)

(continued from previous page)

```
)
resource = Resource.from_contents(loaded)
registry = resource @ initial_registry
```

which will give you a registry with our resource added to it. Let's check by using *Registry.contents*, which takes a URI and should show us the contents of our resource:

```
print(registry.contents("urn:example:my-schema"))
```

```
{'$id': 'urn:example:my-schema', '$schema': 'https://json-schema.org/draft/2020-12/schema
↪', 'type': 'integer'}
```

If your schema did *not* have a $schema keyword, you'd get an error:

```
another = json.loads(
    """
    {
      "$id": "urn:example:my-second-schema",
      "type": "integer"
    }
    """,
)
print(Resource.from_contents(another))
```

```
Traceback (most recent call last):
    ...
referencing.exceptions.CannotDetermineSpecification: {'$id': 'urn:example:my-second-
↪schema', 'type': 'integer'}
```

which is telling you that the resource you've tried to create is ambiguous – there's no way to know which version of JSON Schema you intend it to be written for.

You can of course instead directly create a *Resource*, instead of using *Resource.from_contents*, which will allow you to specify which version of JSON Schema you're intending your schema to be written for:

```
import referencing.jsonschema
second = Resource(contents=another, specification=referencing.jsonschema.DRAFT202012)
```

and now of course can add it as above:

```
registry = second @ registry
print(registry.contents("urn:example:my-second-schema"))
```

```
{'$id': 'urn:example:my-second-schema', 'type': 'integer'}
```

As a shorthand, you can also use *Specification.create_resource* to create a *Resource* slightly more tersely. E.g., an equivalent way to create the above resource is:

```
second_again = referencing.jsonschema.DRAFT202012.create_resource(another)
print(second_again == second)
```

```
True
```

If your resource doesn't contain an $id keyword, you'll get a different error if you attempt to add it to a registry:

```
third = Resource(
    contents=json.loads("""{"type": "integer"}"""),
    specification=referencing.jsonschema.DRAFT202012,
)
registry = third @ registry
```

```
Traceback (most recent call last):
    ...
referencing.exceptions.NoInternalID: Resource(contents={'type': 'integer'}, _
↪specification=<Specification name='draft2020-12'>)
```

which is now saying that there's no way to add this resource to a registry directly, as it has no `$id` – you must provide whatever URI you intend to use to refer to this resource to be able to add it.

You can do so using *referencing.Registry.with_resource* instead of the *@ operator* which we have used thus far, and which takes the explicit URI you wish to use as an argument:

```
registry = registry.with_resource(uri="urn:example:my-third-schema", resource=third)
```

which now allows us to use the URI we associated with our third resource to retrieve it:

```
print(registry.contents("urn:example:my-third-schema"))
```

```
{'type': 'integer'}
```

If you have more than one resource to add, you can use *Registry.with_resources* (with an `s`) to add many at once, or, if they meet the criteria to use `@`, you can use `[one, two, three] @ registry` to add all three resources at once.

You may also want to have a look at *Registry.with_contents* for a further method to add resources to a registry without constructing a *Resource* object yourself.

CHAPTER

THREE

DYNAMICALLY RETRIEVING RESOURCES

Sometimes one wishes to dynamically retrieve or construct *Resource*s which *don't* already live in-memory within a *Registry*. This might be resources retrieved dynamically from a database, from files somewhere on disk, from some arbitrary place over the internet, or from the like. We'll refer to such resources not present in-memory as *external resources*.

The `retrieve` argument to `Registry` objects can be used to configure a callable which will be used anytime a requested URI is *not* present in the registry, thereby allowing you to retrieve it from whichever location it lives in. Here's an example of automatically retrieving external references by downloading them via httpx, illustrated by then automatically retrieving one of the JSON Schema metaschemas from the network:

```
from referencing import Registry, Resource
import httpx


def retrieve_via_httpx(uri):
    response = httpx.get(uri)
    return Resource.from_contents(response.json())


registry = Registry(retrieve=retrieve_via_httpx)
resolver = registry.resolver()
print(resolver.lookup("https://json-schema.org/draft/2020-12/schema"))
```

Note: In the case of JSON Schema, the specifications generally discourage implementations from automatically retrieving these sorts of external resources over the network due to potential security implications. See schema-references in particular.

referencing will of course therefore not do any such thing automatically, and this section generally assumes that you have personally considered the security implications for your own use case.

3.1 Caching

A common concern in these situations is also to *cache* the resulting resource such that repeated lookups of the same URI do not repeatedly call your retrieval function and thereby make network calls, hit the filesystem, etc.

You are of course free to use whatever caching mechanism is convenient even if it uses caching functionality entirely unrelated to this library (e.g. one specific to `httpx` in the above example, or one using `functools.lru_cache` internally).

Nonetheless, because it is so common to retrieve a JSON string and construct a resource from it, *`referencing.retrieval.to_cached_resource`* is a decorator which can help. If you use it, your retrieval callable should return a `str`, not a *`Resource`*, as the decorator will handle deserializing your response and constructing a *`Resource`* from it (this is mostly because otherwise, deserialized JSON is generally not hashable if it ends up being a Python `dict`).

The above example would be written:

```
from referencing import Registry, Resource
import httpx
import referencing.retrieval


@referencing.retrieval.to_cached_resource()
def cached_retrieve_via_httpx(uri):
    return httpx.get(uri).text


registry = Registry(retrieve=cached_retrieve_via_httpx)
resolver = registry.resolver()
print(resolver.lookup("https://json-schema.org/draft/2020-12/schema"))
```

It is otherwise functionally equivalent to the above, other than that retrieval will not repeatedly make a web request.

CHAPTER

FOUR

SCHEMA PACKAGES

The *Registry* object is a useful way to ship Python packages which essentially bundle a set of JSON Schemas for use at runtime from Python.

In order to do so, you likely will want to:

- Collect together the JSON files you wish to ship
- Put them inside a Python package (one you possibly tag with the `jsonschema` keyword for easy discoverability). Remember to ensure you have configured your build tool to include the JSON files in your built package distribution – for e.g. hatch this is likely automatic, but for `setuptools` may involve creating a suitable `MANIFEST.in`.
- Instantiate a *Registry* object somewhere globally within the package
- Call *Registry.crawl* at import-time, such that users of your package get a "fully ready" registry to use

For an example of such a package, see `jsonschema-specifications`, which bundles the JSON Schema "official" schemas for use.

CHAPTER

FIVE

COMPATIBILITY

`referencing` is currently in beta so long as version numbers begin with a `0`, meaning its public interface may change if issues are uncovered, though not typically without reason. Once it seems clear that the interfaces look correct (likely after `referencing` is in use for some period of time) versioning will move to CalVer and interfaces will not change in backwards-incompatible ways without deprecation periods.

Note: Backwards compatibility is always defined relative to the specifications we implement. Changing a behavior which is incorrect according to the relevant referencing specifications is not considered a backwards-incompatible change – on the contrary, it's considered a bug fix.

In the spirit of `having some explicit detail on referencing's public interfaces`, here is a non-exhaustive list of things which are *not* part of the `referencing` public interface, and therefore which may change without warning, even once no longer in beta:

- All commonly understood indicators of privacy in Python – in particular, (sub)packages, modules and identifiers beginning with a single underscore. In the case of modules or packages, this includes *all* of their contents recursively, regardless of their naming.
- All contents in the `referencing.tests` package unless explicitly indicated otherwise
- The precise contents and wording of exception messages raised by any callable, private *or* public.
- The precise contents of the `__repr__` of any type defined in the package.
- The ability to *instantiate* exceptions defined in *`referencing.exceptions`*, with the sole exception of those explicitly indicating they are publicly instantiable (notably *`referencing.exceptions.NoSuchResource`*).
- The instantiation of any type with no public identifier, even if instances of it are returned by other public API. E.g., *`referencing._core.Resolver`* is not publicly exposed, and it is not public API to instantiate it in ways other than by calling *`referencing.Registry.resolver`* or an equivalent. All of its public attributes are of course public, however.
- The concrete types within the signature of a callable whenever they differ from their documented types. In other words, if a function documents that it returns an argument of type `Mapping[int, Sequence[str]]`, this is the promised return type, not whatever concrete type is returned which may be richer or have additional attributes and methods. Changes to the signature will continue to guarantee this return type (or a broader one) but indeed are free to change the concrete type.
- Any identifiers in any modules which are imported from other modules. In other words, if `referencing.foo` imports `bar` from `referencing.quux`, it is *not* public API to use `referencing.foo.bar`; only `referencing.quux.bar` is public API. This does not apply to any objects exposed directly on the `referencing` package (e.g. *`referencing.Resource`*), which are indeed public.
- Subclassing of any class defined throughout the package. Doing so is not supported for any object, and in general most types will raise exceptions to remind downstream users not to do so.

If any API usage may be questionable, feel free to open a discussion (or issue if appropriate) to clarify.

CHAPTER

SIX

API REFERENCE

Cross-specification, implementation-agnostic JSON referencing.

class referencing.**Anchor**(*name: str*, *resource:* Resource*[*D*]*)

A simple anchor in a *Resource*.

name: str

resource: *Resource*[*D*]

resolve(*resolver:* Resolver*[*D*]*)

Return the resource for this anchor.

class referencing.**Registry**(*resources=HashTrieMap({})*, *anchors: HashTrieMap[tuple[URI, str], AnchorType[D]] = HashTrieMap({})*, *uncrawled: HashTrieSet[URI] = HashTrieSet({})*, *retrieve: Retrieve[D] = <function _fail_to_retrieve>*)

A registry of *Resource*s, each identified by their canonical URIs.

Registries store a collection of in-memory resources, and optionally enable additional resources which may be stored elsewhere (e.g. in a database, a separate set of files, over the network, etc.).

They also lazily walk their known resources, looking for subresources within them. In other words, subresources contained within any added resources will be retrievable via their own IDs (though this discovery of subresources will be delayed until necessary).

Registries are immutable, and their methods return new instances of the registry with the additional resources added to them.

The `retrieve` argument can be used to configure retrieval of resources dynamically, either over the network, from a database, or the like. Pass it a callable which will be called if any URI not present in the registry is accessed. It must either return a *Resource* or else raise a *NoSuchResource* exception indicating that the resource does not exist even according to the retrieval logic.

__getitem__(*uri: str*) → *Resource*[*D*]

Return the (already crawled) *Resource* identified by the given URI.

__iter__() → Iterator[str]

Iterate over all crawled URIs in the registry.

__len__() → int

Count the total number of fully crawled resources in this registry.

__rmatmul__(*new:* Resource*[*D*]* | *Iterable*[Resource*[*D*]]*) → *Registry*[*D*]

Create a new registry with resource(s) added using their internal IDs.

Resources must have a internal IDs (e.g. the $id keyword in modern JSON Schema versions), otherwise an error will be raised.

Both a single resource as well as an iterable of resources works, i.e.:

- `resource @ registry` or
- `[iterable, of, multiple, resources] @ registry`

which – again, assuming the resources have internal IDs – is equivalent to calling *Registry.with_resources* as such:

```
registry.with_resources(
    (resource.id(), resource) for resource in new_resources
)
```

Raises

NoInternalID – if the resource(s) in fact do not have IDs

get_or_retrieve(*uri: str*) → *Retrieved*[*D*, *Resource*[*D*]]

Get a resource from the registry, crawling or retrieving if necessary.

May involve crawling to find the given URI if it is not already known, so the returned object is a *Retrieved* object which contains both the resource value as well as the registry which ultimately contained it.

remove(*uri: str*)

Return a registry with the resource identified by a given URI removed.

anchor(*uri: str*, *name: str*)

Retrieve a given anchor from a resource which must already be crawled.

contents(*uri: str*) → *D*

Retrieve the (already crawled) contents identified by the given URI.

crawl() → *Registry*[*D*]

Crawl all added resources, discovering subresources.

with_resource(*uri: str*, *resource:* Resource*[*D*]*)

Add the given *Resource* to the registry, without crawling it.

with_resources(*pairs: Iterable[tuple[str,* Resource*[*D*]]]*) → *Registry*[*D*]

Add the given *Resource*s to the registry, without crawling them.

with_contents(*pairs: Iterable[tuple[str,* D*]]*, ***kwargs: Any*) → *Registry*[*D*]

Add the given contents to the registry, autodetecting when necessary.

combine(**registries:* Registry*[*D*]*) → *Registry*[*D*]

Combine together one or more other registries, producing a unified one.

resolver(*base_uri: str = ''*) → *Resolver*[*D*]

Return a *Resolver* which resolves references against this registry.

resolver_with_root(*resource:* Resource*[*D*]*) → *Resolver*[*D*]

Return a *Resolver* with a specific root resource.

class referencing.**Resource**(*contents:* D, *specification:* Specification*[*D*]*)

A document (deserialized JSON) with a concrete interpretation under a spec.

In other words, a Python object, along with an instance of *Specification* which describes how the document interacts with referencing – both internally (how it refers to other *Resource*s) and externally (how it should be identified such that it is referenceable by other documents).

contents: *D*

classmethod from_contents(*contents: ~referencing.typing.D*, *default_specification: type[~referencing._core.Specification[~referencing.typing.D]] | ~referencing._core.Specification[~referencing.typing.D] = <class 'referencing._core.Specification'>*) → *Resource*[*D*]

Create a resource guessing which specification applies to the contents.

Raises

CannotDetermineSpecification – if the given contents don't have any discernible information which could be used to guess which specification they identify as

classmethod opaque(*contents:* D) → *Resource*[*D*]

Create an opaque *Resource* – i.e. one with opaque specification.

See *Specification.OPAQUE* for details.

id() → str | None

Retrieve this resource's (specification-specific) identifier.

subresources() → Iterable[*Resource*[*D*]]

Retrieve this resource's subresources.

anchors() → Iterable[*Anchor*[*D*]]

Retrieve this resource's (specification-specific) identifier.

pointer(*pointer: str*, *resolver:* Resolver[D]) → *Resolved*[*D*]

Resolve the given JSON pointer.

Raises

exceptions.PointerToNowhere – if the pointer points to a location not present in the document

class referencing.**Specification**(*name: str*, *id_of: Callable[[D], URI | None]*, *subresources_of: Callable[[D], Iterable[D]]*, *maybe_in_subresource: _MaybeInSubresource[D]*, *anchors_in: Callable[[Specification[D], D], Iterable[AnchorType[D]]]*)

A specification which defines referencing behavior.

The various methods of a *Specification* allow for varying referencing behavior across JSON Schema specification versions, etc.

name: `str`

A short human-readable name for the specification, used for debugging.

id_of: `Callable[[D], str | None]`

Find the ID of a given document.

subresources_of: `Callable[[D], Iterable[D]]`

Retrieve the subresources of the given document (without traversing into the subresources themselves).

maybe_in_subresource: *_MaybeInSubresource*`[D]`

While resolving a JSON pointer, conditionally enter a subresource (if e.g. we have just entered a keyword whose value is a subresource)

OPAQUE: `ClassVar[Specification[Any]] = <Specification name='opaque'>`

An opaque specification where resources have no subresources nor internal identifiers.

detect() → *Specification*[*D*]

Attempt to discern which specification applies to the given contents.

May be called either as an instance method or as a class method, with slightly different behavior in the following case:

Recall that not all contents contains enough internal information about which specification it is written for – the JSON Schema `{}`, for instance, is valid under many different dialects and may be interpreted as any one of them.

When this method is used as an instance method (i.e. called on a specific specification), that specification is used as the default if the given contents are unidentifiable.

On the other hand when called as a class method, an error is raised.

To reiterate, `DRAFT202012.detect({})` will return `DRAFT202012` whereas the class method `Specification.detect({})` will raise an error.

(Note that of course `DRAFT202012.detect(...)` may return some other specification when given a schema which *does* identify as being for another version).

Raises

CannotDetermineSpecification – if the given contents don't have any discernible information which could be used to guess which specification they identify as

anchors_in(*contents:* D)

Retrieve the anchors contained in the given document.

create_resource(*contents:* D) → *Resource*[*D*]

Create a resource which is interpreted using this specification.

6.1 Private Objects

The following objects are private in the sense that constructing or importing them is not part of the *referencing* public API, as their name indicates (by virtue of beginning with an underscore).

They are however public in the sense that other public API functions may return objects of these types.

Plainly then, you may rely on their methods and attributes not changing in backwards incompatible ways once *referencing* itself is stable, but may not rely on importing or constructing them yourself.

class referencing._core.**Resolved**(*contents:* D, *resolver:* Resolver[D])

A reference resolved to its contents by a *Resolver*.

contents: *D*

resolver: *Resolver*[*D*]

class referencing._core.**Retrieved**(*value:* AnchorOrResource, *registry:* Registry[D])

A value retrieved from a *Registry*.

value: *AnchorOrResource*

registry: *Registry*[*D*]

class referencing._core.**AnchorOrResource**

An anchor or resource.

alias of TypeVar('AnchorOrResource', ~referencing.typing.Anchor[~typing.Any], ~referencing._core.Resource[~typing.Any])

class referencing._core.**Resolver**(*base_uri:* URI, *registry:* Registry[D], *previous:* List[URI] = List([])

A reference resolver.

Resolvers help resolve references (including relative ones) by pairing a fixed base URI with a *Registry*.

This object, under normal circumstances, is expected to be used by *implementers of libraries* built on top of *referencing* (e.g. JSON Schema implementations or other libraries resolving JSON references), not directly by end-users populating registries or while writing schemas or other resources.

References are resolved against the base URI, and the combined URI is then looked up within the registry.

The process of resolving a reference may itself involve calculating a *new* base URI for future reference resolution (e.g. if an intermediate resource sets a new base URI), or may involve encountering additional subresources and adding them to a new registry.

lookup(*ref: str*) → *Resolved*[*D*]

Resolve the given reference to the resource it points to.

Raises

- *exceptions.Unresolvable* – or a subclass thereof (see below) if the reference isn't resolvable
- *exceptions.NoSuchAnchor* – if the reference is to a URI where a resource exists but contains a plain name fragment which does not exist within the resource
- *exceptions.PointerToNowhere* – if the reference is to a URI where a resource exists but contains a JSON pointer to a location within the resource that does not exist

in_subresource(*subresource:* Resource[D]) → *Resolver*[*D*]

Create a resolver for a subresource (which may have a new base URI).

dynamic_scope() → Iterable[tuple[str, *Registry*[*D*]]]

In specs with such a notion, return the URIs in the dynamic scope.

class referencing._core.**_MaybeInSubresource**(**args*, ***kwargs*)

class referencing._core.**_Unset**

A sentinel object used internally to satisfy the type checker.

Neither accessing nor explicitly passing this object anywhere is public API, and it is only documented here at all to get Sphinx to not complain.

6.2 Submodules

6.2.1 referencing.jsonschema

Referencing implementations for JSON Schema specs (historic & current).

referencing.jsonschema.**ObjectSchema**

A JSON Schema which is a JSON object

alias of Mapping[str, Any]

referencing.jsonschema.**Schema**

A JSON Schema of any kind

alias of Union[bool, Mapping[str, Any]]

referencing.jsonschema.**SchemaResource**

A Resource whose contents are JSON Schemas

alias of *Resource*[Union[bool, Mapping[str, Any]]]

referencing.jsonschema.**SchemaRegistry**

A JSON Schema Registry

alias of *Registry*[Union[bool, Mapping[str, Any]]]

referencing.jsonschema.**EMPTY_REGISTRY: *Registry*[bool | Mapping[str, Any]] = <Registry (0 resources)>**

The empty JSON Schema Registry

exception referencing.jsonschema.**UnknownDialect**(*uri: str*)

A dialect identifier was found for a dialect unknown by this library.

If it's a custom ("unofficial") dialect, be sure you've registered it.

uri: *str*

referencing.jsonschema.**DRAFT202012 = <Specification name='draft2020-12'>**

JSON Schema draft 2020-12

referencing.jsonschema.**DRAFT201909 = <Specification name='draft2019-09'>**

JSON Schema draft 2019-09

referencing.jsonschema.**DRAFT7 = <Specification name='draft-07'>**

JSON Schema draft 7

referencing.jsonschema.**DRAFT6 = <Specification name='draft-06'>**

JSON Schema draft 6

referencing.jsonschema.**DRAFT4 = <Specification name='draft-04'>**

JSON Schema draft 4

referencing.jsonschema.**DRAFT3 = <Specification name='draft-03'>**

JSON Schema draft 3

referencing.jsonschema.**specification_with**(*dialect_id: str*, *default:* Specification*[Any]* | _Unset = *_Unset.SENTINEL*) → *Specification*[Any]

Retrieve the *Specification* with the given dialect identifier.

Raises

UnknownDialect – if the given `dialect_id` isn't known

class referencing.jsonschema.**DynamicAnchor**(*name: str*, *resource:* Resource*[bool | Mapping[str, Any]]*)

Dynamic anchors, introduced in draft 2020.

name: *str*

resource: *Resource*[bool | Mapping[str, Any]]

resolve(*resolver:* Resolver*[bool | Mapping[str, Any]]*) → *Resolved*[bool | Mapping[str, Any]]

Resolve this anchor dynamically.

referencing.jsonschema.**lookup_recursive_ref**(*resolver:* Resolver*[bool | Mapping[str, Any]]*) → *Resolved*[bool | Mapping[str, Any]]

Recursive references (via recursive anchors), present only in draft 2019.

As per the 2019 specification (§ 8.2.4.2.1), only the # recursive reference is supported (and is therefore assumed to be the relevant reference).

6.2.2 referencing.exceptions

Errors, oh no!

exception referencing.exceptions.**NoSuchResource**(*ref:* URI)

Bases: KeyError

The given URI is not present in a registry.

Unlike most exceptions, this class *is* intended to be publicly instantiable and *is* part of the public API of the package.

ref: *URI*

exception referencing.exceptions.**NoInternalID**(*resource:* Resource*[Any]*)

Bases: Exception

A resource has no internal ID, but one is needed.

E.g. in modern JSON Schema drafts, this is the $id keyword.

One might be needed if a resource was to-be added to a registry but no other URI is available, and the resource doesn't declare its canonical URI.

resource: *Resource*[Any]

exception referencing.exceptions.**Unretrievable**(*ref:* URI)

Bases: KeyError

The given URI is not present in a registry, and retrieving it failed.

ref: *URI*

exception referencing.exceptions.**CannotDetermineSpecification**(*contents: Any*)

Bases: Exception

Attempting to detect the appropriate *Specification* failed.

This happens if no discernible information is found in the contents of the new resource which would help identify it.

contents: Any

exception referencing.exceptions.**Unresolvable**(*ref:* URI)

Bases: Exception

A reference was unresolvable.

ref: *URI*

exception referencing.exceptions.**PointerToNowhere**(*ref:* URI, *resource:* Resource*[Any]*)

Bases: *Unresolvable*

A JSON Pointer leads to a part of a document that does not exist.

resource: *Resource*[Any]

exception referencing.exceptions.**NoSuchAnchor**(*ref:* URI, *resource:* Resource[Any], *anchor:* *str*)

Bases: *Unresolvable*

An anchor does not exist within a particular resource.

resource: *Resource*[Any]

anchor: str

exception referencing.exceptions.**InvalidAnchor**(*ref:* URI, *resource:* Resource[Any], *anchor:* *str*)

Bases: *Unresolvable*

An anchor which could never exist in a resource was dereferenced.

It is somehow syntactically invalid.

resource: *Resource*[Any]

anchor: str

6.2.3 referencing.retrieval

Helpers related to (dynamic) resource retrieval.

referencing.retrieval.**to_cached_resource**(*cache: Callable[[Retrieve[D]], Retrieve[D]] | None = None, loads: Callable[[_T], D] = <function loads>, from_contents: Callable[[D], Resource[D]] = <bound method Resource.from_contents of <class 'referencing._core.Resource'>>*) → Callable[[Callable[[*URI*], *_T*]], *Retrieve*[*D*]]

Create a retriever which caches its return values from a simpler callable.

Takes a function which returns things like serialized JSON (strings) and returns something suitable for passing to *Registry* as a retrieve function.

This decorator both reduces a small bit of boilerplate for a common case (deserializing JSON from strings and creating *Resource* objects from the result) as well as makes the probable need for caching a bit easier. Retrievers which otherwise do expensive operations (like hitting the network) might otherwise be called repeatedly.

Examples

```
from referencing import Registry
from referencing.typing import URI
import referencing.retrieval

@referencing.retrieval.to_cached_resource()
def retrieve(uri: URI):
    print(f"Retrieved {uri}")

    # Normally, go get some expensive JSON from the network, a file ...
    return '''
        {
            "$schema": "https://json-schema.org/draft/2020-12/schema",
```

(continues on next page)

(continued from previous page)

```
            "foo": "bar"
        }
    '''

one = Registry(retrieve=retrieve).get_or_retrieve("urn:example:foo")
print(one.value.contents["foo"])

# Retrieving the same URI again reuses the same value (and thus doesn't
# print another retrieval message here)
two = Registry(retrieve=retrieve).get_or_retrieve("urn:example:foo")
print(two.value.contents["foo"])
```

```
Retrieved urn:example:foo
bar
bar
```

class referencing.retrieval.**_T**

A serialized document (e.g. a JSON string)

alias of TypeVar('_T')

6.2.4 referencing.typing

Type-annotation related support for the referencing library.

referencing.typing.**URI**

A URI which identifies a *Resource*.

class referencing.typing.**D**

The type of documents within a registry.

alias of TypeVar('D')

class referencing.typing.**Retrieve**(**args*, ***kwargs*)

A retrieval callable, usable within a *Registry* for resource retrieval.

Does not make assumptions about where the resource might be coming from.

class referencing.typing.**Anchor**(**args*, ***kwargs*)

An anchor within a *Resource*.

Beyond "simple" anchors, some specifications like JSON Schema's 2020 version have dynamic anchors.

property name: str

Return the name of this anchor.

resolve(*resolver:* Resolver[D]) → *Resolved*[*D*]

Return the resource for this anchor.

CHAPTER

SEVEN

CHANGELOG

7.1 v0.34.0

- Also look inside `definitions` keywords even on newer dialects. The specification recommends doing so regardless of the rename to `$defs`.

7.2 v0.33.0

- Add a `referencing.jsonschema.SchemaResource` type alias to go along with the other JSON Schema specialized types.

7.3 v0.32.1

- Make `Specification.detect` raise a `CannotDetermineSpecification` error even if passed a mapping with a `$schema` key that doesn't match JSON Schema dialect semantics (e.g. a non-string).

7.4 v0.32.0

- Add `Specification.detect`, which essentially operates like `Resource.from_contents` without constructing a resource (i.e. it simply returns the detected specification).

7.5 v0.31.1

- No user facing changes.

7.6 v0.31.0

- Add `referencing.jsonschema.EMPTY_REGISTRY` (which simply has a convenient type annotation, but otherwise is just `Registry()`).

7.7 v0.30.2

- Minor docs improvement.

7.8 v0.30.1

- Ensure that an `sdist` contains the test suite JSON files.

7.9 v0.30.0

- Declare support for Python 3.12.

7.10 v0.29.3

- Documentation fix.

7.11 v0.29.2

- Improve the hashability of exceptions when they contain hashable data.

7.12 v0.29.1

- Minor docs improvement.

7.13 v0.29.0

- Add `referencing.retrieval.to_cached_resource`, a simple caching decorator useful when writing a retrieval function turning JSON text into resources without repeatedly hitting the network, filesystem, etc.

7.14 v0.28.6

- No user-facing changes.

7.15 v0.28.5

- Fix a type annotation and fill in some missing test coverage.

7.16 v0.28.4

- Fix a type annotation.

7.17 v0.28.3

- No user-facing changes.

7.18 v0.28.2

- Added some additional packaging trove classifiers.

7.19 v0.28.1

- More minor documentation improvements

7.20 v0.28.0

- Minor documentation improvement

7.21 v0.27.4

- Minor simplification to the docs structure.

7.22 v0.27.3

- Also strip fragments when using `__getitem__` on URIs with empty fragments.

7.23 v0.27.2

- Another fix for looking up anchors from non-canonical URIs, now when they're inside a subresource which has a relative `$id`.

7.24 v0.27.1

- Improve a small number of docstrings.

7.25 v0.27.0

- Support looking up anchors from non-canonical URIs. In other words, if you add a resource at the URI `http://example.com`, then looking up the anchor `http://example.com#foo` now works even if the resource has some internal `$id` saying its canonical URI is `http://somethingelse.example.com`.

7.26 v0.26.4

- Further API documentation.

7.27 v0.26.3

- Add some documentation on `referencing` public and non-public API.

7.28 v0.26.2

- Also suggest a proper JSON Pointer for users who accidentally use `#/` and intend to refer to the entire resource.

7.29 v0.26.1

- No changes.

7.30 v0.26.0

- Attempt to suggest a correction if someone uses '#foo/bar', which is neither a valid plain name anchor (as it contains a slash) nor a valid JSON pointer (as it doesn't start with a slash)

7.31 v0.25.3

- Normalize the ID of JSON Schema resources with empty fragments (by removing the fragment). Having a schema with an ID with empty fragment is discouraged, and newer versions of the spec may flat-out make it an error, but older meta-schemas indeed used IDs with empty fragments, so some extra normalization was needed and useful here even beyond what was previously done. TBD on whether this is exactly right if/when another referencing spec defines differing behavior.

7.32 v0.25.2

- Minor tweaks to the package keywords and description.

7.33 v0.25.1

- Minor internal tweaks to the docs configuration.

7.34 v0.25.0

- Bump the minimum version of `rpds.py` used, enabling registries to be used from multiple threads.

7.35 v0.24.4

- Fix handling of IDs with empty fragments (which are equivalent to URIs with no fragment)

7.36 v0.24.3

- Further intro documentation

7.37 v0.24.2

- Fix handling of `additionalProperties` with boolean value on Draft 4 (where the boolean isn't a schema, it's a special allowed value)

7.38 v0.24.1

- Add a bit of intro documentation

7.39 v0.24.0

- `pyrsistent` was replaced with `rpds.py` (Python bindings to the Rust rpds crate), which seems to be quite a bit faster. No user-facing changes really should be expected here.

PYTHON MODULE INDEX

r

INDEX

Symbols

A

C

D

E

F

G

I

L

M

N