Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: July 23, 2018

H. Andrews, Ed.
Cloudflare, Inc.
A. Wright, Ed.

January 19, 2018

JSON Hyper-Schema: A Vocabulary for Hypermedia Annotation of JSON
draft-handrews-json-schema-hyperschema-01

Abstract

JSON Schema is a JSON-based format for describing JSON data using various vocabularies. This document specifies a vocabulary for annotating JSON documents with hyperlinks. These hyperlinks include attributes describing how to manipulate and interact with remote resources through hypermedia environments such as HTTP, as well as determining whether the link is usable based on the instance value. The hyperlink serialization format described in this document is also usable independent of JSON Schema.

Note to Readers

The issues list for this draft can be found at <https://github.com/json-schema-org/json-schema-spec/issues>.

For additional information, see <http://json-schema.org/>.

To provide feedback, use this issue tracker, the communication methods listed on the homepage, or email the document editors.

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 July 23, 2018.

Table of Contents

1. Introduction

JSON Hyper-Schema is a JSON Schema vocabulary for annotating JSON documents with hyperlinks and instructions for processing and

manipulating remote JSON resources through hypermedia environments such as HTTP.

The term JSON Hyper-Schema is used to refer to a JSON Schema that uses these keywords. The term "hyper-schema" on its own refers to a JSON Hyper-Schema within the scope of this specification.

The primary mechanism introduced for specifying links is the Link Description Object (LDO), which is a serialization of the abstract link model defined in RFC 8288, section 2 [RFC8288].

This specification will use the concepts, syntax, and terminology defined by the JSON Schema core [json-schema] and JSON Schema validation [json-schema-validation] specifications. It is advised that readers have a copy of these specifications.

2. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

3. Overview

JSON Hyper-Schema makes it possible to build hypermedia systems from JSON documents by describing how to construct hyperlinks from instance data.

The combination of a JSON instance document and a valid application/schema+json hyper-schema for that instance behaves as a single hypermedia representation. By allowing this separation, hyper-schema-based systems can gracefully support applications that expect plain JSON, while providing full hypermedia capabilities for hyper-schema-aware applications and user agents.

User agents can detect the presence of hyper-schema by looking for the application/schema+json media type and a "$schema" value that indicates the presence of the hyper-schema vocabulary. A user agent can then use an implementation of JSON Hyper-Schema to provide an interface to the combination of the schema and instance documents as a single logical representation of a resource, just as with any single-document hypermedia representation format.

Hyper-schemas allow representations to take up fewer bytes on the wire, and distribute the burden of link construction from the server to each client. A user agent need not construct a link unless a client application requests that link. JSON Hyper-Schema can also be used on the server side to generate other link serializations or

representation formats at runtime, or pre-emptively follow links to facilitate server push usage.

Here is an example hyper-schema that adds a single link, with the IANA-registered link relation type "self", that is built from an instance with one known object field named "id":

```
{
    "type": "object",
    "properties": {
        "id": {
            "type": "number",
            "readOnly": true
        }
    },
    "links": [
        {
            "rel": "self",
            "href": "thing/{id}"
        }
    ]
}
```

If the instance is {"id": 1234}, and its base URI according to RFC 3986 section 5.1 [RFC3986], is "https://api.example.com/", then "https://api.example.com/thing/1234" is the resulting link's target URI.

3.1. Terminology

The terms "schema", "instance", and "meta-schema" are to be interpreted as defined in the JSON Schema core specification [json-schema].

The terms "applicable" and "attached" are to be interpreted as defined in Section 3 of the JSON Schema validation specification [json-schema-validation].

The terms "link", "link context" (or "context"), "link target" (or "target"), and "target attributes" are to be interpreted as defined in Section 2 of RFC 8288 [RFC8288].

The term "user agent" is to be interpreted as defined in Section 2.1 of RFC 7230 [RFC7230], generalized to apply to any protocol that may be used in a hypermedia system rather than specifically being an HTTP client.

This specification defines the following terms:

JSON Hyper-Schema A JSON Schema using the keywords defined by this specification.

hyper-schema Within this document, the term "hyper-schema" always refers to a JSON Hyper-Schema

link validity A valid link for an instance is one that is applicable to that instance and does not fail any requirement imposed by the keywords in the Link Description Object.

generic user agent A user agent which can be used to interact with any resource, from any server, from among the standardized link relations, media types, URI schemes, and protocols that it supports; though it may be extendible to specially handle particular profiles of media types.

client application An application which uses a hypermedia system for a specific purpose. Such an application may also be its own user agent, or it may be built on top of a generic user agent. A client application is programmed with knowledge of link relations, media types, URI schemes, protocols, and data structures that are specific to the application's domain.

client input Data provided through a user agent, and most often also through a client application. Such data may be requested from a user interactively, or provided before interaction in forms such as command-line arguments, configuration files, or hardcoded values in source code.

operation A specific use of a hyperlink, such as making a network request (for a URI with a scheme such as "http://" that indicates a protocol) or otherwise taking action based on a link (reading data from a "data:" URI, or constructing an email message based on a "mailto:" link). For protocols such as HTTP that support multiple methods, each method is considered to be a separate operation on the same link.

3.2. Functionality

A JSON Hyper-Schema implementation is able to take a hyper-schema, an instance, and in some cases client input, and produce a set of fully resolved valid links. As defined by RFC 8288, section 2 [RFC8288], a link consists of a context, a typed relation, a target, and optionally additional target attributes.

The relation type and target attributes are taken directly from each link's Link Description Object. The context and target identifiers

are constructed from some combination of URI Templates, instance data, and (in the case of the target identifier) client input.

The target is always fully identified by a URI. Due to the lack of a URI fragment identifier syntax for application/json and many other media types that can be used with JSON Hyper-Schema, the context may be only partially identified by a URI. In such cases, the remaining identification will be provided as a JSON Pointer.

A few IANA-registered link relation types are given specific semantics in a JSON Hyper-Schema document. A "self" link is used to interact with the resource that the instance document represents, while "collection" and "item" links identify resources for which collection-specific semantics can be assumed.

4. Meta-Schemas and Output Schema

The current URI for the JSON Hyper-Schema meta-schema is <http://json-schema.org/draft-07/hyper-schema#>.

The link description format (Section 6) can be used without JSON Schema, and use of this format can be declared by referencing the normative link description schema as the schema for the data structure that uses the links. The URI of the normative link description schema is: <http://json-schema.org/draft-07/links#>.

JSON Hyper-Schema implementations are free to provide output in any format. However, a specific format is defined for use in the conformance test suite, which is also used to illustrate points in the "Implementation Requirements" (Section 7), and to show the output generated by examples (Section 9). It is RECOMMENDED that implementations be capable of producing output in this format to facilitated testing. The URI of the JSON Schema describing the recommended output format is <http://json-schema.org/draft-07/hyper-schema-output#>.

5. Schema Keywords

Hyper-schema keywords from all schemas that are applicable to a position in an instance, as defined by Section 3 of JSON Schema validation [json-schema-validation], can be used with that instance.

When multiple subschemas are applicable to a given sub-instance, all "link" arrays MUST be combined, in any order, into a single set. Each object in the resulting set MUST retain its own list of applicable "base" values, in resolution order, from the same schema and any parent schemas.

As with all JSON Schema keywords, all keywords described in this section are optional. The minimal valid JSON Hyper-schema is the blank object.

5.1. base

If present, this keyword MUST be first resolved as a URI Template (Section 7.2), and then MUST be resolved as a URI Reference against the current URI base of the instance. The result MUST be set as the new URI base for the instance while processing the sub-schema containing "base" and all sub-schemas within it.

The process for resolving the "base" template can be different when being resolved for use with "anchor" than when being resolved for use with "href", which is explained in detail in the URI Templating section.

5.2. links

The "links" property of schemas is used to associate Link Description Objects with instances. The value of this property MUST be an array, and the items in the array must be Link Description Objects, as defined below.

6. Link Description Object

A Link Description Object (LDO) is a serialization of the abstract link model defined in RFC 8288, section 2 [RFC8288]. As described in that document, a link consists of a context, a relation type, a target, and optionally target attributes. JSON Hyper-Schema's LDO provides all of these, along with additional features using JSON Schema to describe input for use with the links in various ways.

Due to the use of URI Templates to identify link contexts and targets, as well as optional further use of client input when identifying targets, an LDO is a link template that may resolve to multiple links when used with a JSON instance document.

A specific use of an LDO, typically involving a request and response across a protocol, is referred to as an operation. For many protocols, multiple operations are possible on any given link. The protocol is indicated by the target's URI scheme. Note that not all URI schemes indicate a protocol that can be used for communications, and even resources with URI schemes that do indicate such protocols need not be available over that protocol.

A Link Description Object MUST be an object, and the "href" (Section 6.3.1) and "rel" (Section 6.2.1) properties MUST be present.

Each keyword is covered briefly in this section, with additional usage explanation and comprehensive examples given later in the document.

6.1. Link Context

In JSON Hyper-Schema, the link's context resource is, by default, the sub-instance to which it is attached (as defined by Section 3 of the JSON Schema validation specification [json-schema-validation]). This is often not the entire instance document. This default context can be changed using the keywords in this section.

Depending on the media type of the instance, it may or may not be possible to assign a URI to the exact default context resource. In particular, application/json does not define a URI fragment resolution syntax, so properties or array elements within a plain JSON document cannot be fully identified by a URI. When it is not possible to produce a complete URI, the position of the context SHOULD be conveyed by the URI of the instance document, together with a separate plain-string JSON Pointer.

Implementations MUST be able to construct the link context's URI, and (if necessary for full identification), a JSON Pointer in string representation form as per RFC 6901, section 5 [RFC6901] in place of a URI fragment. The process for constructing a URI based on a URI template is given in the URI Templating (Section 7.2) section.

6.1.1. anchor

This property sets the context URI of the link. The value of the property is a URI Template [RFC6570], and the resulting URI-reference [RFC3986] MUST be resolved against the base URI of the instance.

The URI is computed from the provided URI template using the same process described for the "href" (Section 6.3.1) property, with the exception that "hrefSchema" (Section 6.6.1) MUST NOT be applied. Unlike target URIs, context URIs do not accept user input.

6.1.2. anchorPointer

This property changes the point within the instance that is considered to be the context resource of the link. The value of the property MUST be a valid JSON Pointer in JSON String representation form, or a valid Relative JSON Pointer [relative-json-pointer] which is evaluated relative to the default context.

While an alternate context with a known URI is best set with the "anchor" (Section 6.1.1) keyword, the lack of a fragment identifier

syntax for application/json means that it is usually not possible to change the context within a JSON instance using a URI.

Even in "+json" media types that define JSON Pointer as a fragment identifier syntax, if the default context is nested within an array, it is not possible to obtain the index of the default context's position in that array in order to construct a pointer to another property in that same nested JSON object. This will be demonstrated in the examples.

The result of processing this keyword SHOULD be a URI fragment if the media type of the instance allows for such a fragment. Otherwise it MUST be a string-encoded JSON Pointer.

6.2. Link Relation Type

The link's relation type identifies its semantics. It is the primary means of conveying how an application can interact with a resource.

Relationship definitions are not normally media type dependent, and users are encouraged to utilize the most suitable existing accepted relation definitions.

6.2.1. rel

The value of this property MUST be a string, and MUST be a single Link Relation Type as defined in RFC 8288, Section 2.1.

This property is required.

6.2.2. "self" Links

A "self" link, as originally defined by Section 4.2.7.2 of RFC 4287 [RFC4287], indicates that the target URI identifies a resource equivalent to the link context. In JSON Hyper-Schema, a "self" link MUST be resolvable from the instance, and therefore "hrefSchema" MUST NOT be present.

Hyper-schema authors SHOULD use "templateRequired" to ensure that the "self" link has all instance data that is needed for use.

A hyper-schema implementation MUST recognize that a link with relation type "self" that has the entire current instance document as its context describes how a user agent can interact with the resource represented by that instance document.

6.2.3. "collection" and "item" Links

RFC 6573 [RFC6573] defines and registers the "item" and "collection" link relation types. JSON Hyper-Schema imposes additional semantics on collection resources indicated by these types.

Implementations MUST recognize the target of a "collection" link and the context of an "item" link as collections.

A well-known design pattern in hypermedia is to use a collection resource to create a member of the collection and give it a server-assigned URI. If the protocol indicated by the URI scheme defines a specific method that is suited to creating a resource with a server-assigned URI, then a collection resource, as identified by these link relation types, MUST NOT define semantics for that method that conflict with the semantics of creating a collection member. Collection resources MAY implement item creation via such a protocol method, and user agents MAY assume that any such operation, if it exists, has item creation semantics.

As such a method would correspond to JSON Hyper-Schema's data submission concept, the "submissionSchema" (Section 6.6.4.2) field for the link SHOULD be compatible with the schema of the representation of the collection's items, as indicated by the "item" link's target resource or the "self" link of the "collection" link's context resource.

6.2.4. Using Extension Relation Types

When no registered relation (aside from "related") applies, users are encouraged to mint their own extension relation types, as described in section 2.1.2 of RFC 8288 [RFC8288]. The simplest approaches for choosing link relation type URIs are to either use a URI scheme that is already in use to identify the system's primary resources, or to use a human-readable, non-dereferenceable URI scheme such as "tag", defined by RFC 4151 [RFC4151].

Extension relation type URIs need not be dereferenceable, even when using a scheme that allows it.

6.3. Link Target

The target URI template is used to identify the link's target, potentially making use of instance data. Additionally, with "hrefSchema" (Section 6.6.1), this template can identify a set of possible target resources to use based on client input. The full process of resolving the URI template, with or without client input, is covered in the URI Templating (Section 7.2) section.

6.3.1. href

The value of the "href" link description property is a template used to determine the target URI of the related resource. The value of the instance property MUST be resolved as a URI-reference [RFC3986] against the base URI of the instance.

This property is REQUIRED.

6.4. Adjusting URI Template Resolution

The keywords in this section are used when resolving all URI Templates involved in hyper-schema: "base", "anchor", and "href". See the URI Templating (Section 7.2) section for the complete template resolution algorithm.

Note that when resolving a "base" template, the attachment point from which resolution begins is the attachment point of the "href" or "anchor" keyword being resolved which requires "base" templates to be resolved, not the attachment point of the "base" keyword itself.

6.4.1. templatePointers

The value of the "templatePointers" link description property MUST be an object. Each property value in the object MUST be a valid JSON Pointer [RFC6901], or a valid Relative JSON Pointer [relative-json-pointer] which is evaluated relative to the attachment point of the link for which the template is being resolved.

For each property name in the object that matches a variable name in the template being resolved, the value of that property adjusts the starting position of variable resolution for that variable. Properties which do not match template variable names in the template being resolved MUST be ignored.

6.4.2. templateRequired

The value of this keyword MUST be an array, and the elements MUST be unique. Each element SHOULD match a variable in the link's URI Template, without percent-encoding. After completing the entire URI Template resolution process, if any variable that is present in this array does not have a value, the link MUST NOT be used.

6.5. Link Target Attributes

All properties in this section are advisory only. While keywords such as "title" and "description" are used primarily to present the link to users, those keywords that predict the nature of a link

interaction or response MUST NOT be considered authoritative. The runtime behavior of the target resource MUST be respected whenever it conflicts with the target attributes in the LDO.

6.5.1. title

This property defines a title for the link. The value MUST be a string.

User agents MAY use this title when presenting the link to the user.

6.5.2. description

This property provides additional information beyond what is present in the title. The value MUST be a string. While a title is preferably short, a description can be used to go into more detail about the purpose and usage of the link.

User agents MAY use this description when presenting the link to the user.

6.5.3. targetMediaType

The value of this property represents the media type RFC 2046 [RFC2046], that is expected to be returned when fetching this resource. This property value MAY be a media range instead, using the same pattern defined in RFC 7231, section 5.3.2 - HTTP "Accept" header [RFC7231].

This property is analogous to the "type" property of other link serialization formats. User agents MAY use this information to inform the interface they present to the user before the link is followed, but MUST NOT use this information in the interpretation of the resulting data. Instead, a user agent MUST use the media type given by the response for run-time interpretation. See the section on "Security Concerns" (Section 10) for a detailed examination of mis-use of "targetMediaType".

For protocols supporting content-negotiation, implementations MAY choose to describe possible target media types using protocol-specific information in "headerSchema" (Section 6.6.2). If both protocol-specific information and "targetMediaType" are present, then the value of "targetMediaType" MUST be compatible with the protocol-specific information, and SHOULD indicate the media type that will be returned in the absence of content negotiation.

When no such protocol-specific information is available, or when the implementation does not recognize the protocol involved, then the value SHOULD be taken to be "application/json".

6.5.4. targetSchema

This property provides a schema that is expected to describe the link target's representation. Depending on the protocol, the schema may or may not describe the request or response to any particular operation performed with the link. See the JSON Hyper-Schema and HTTP (Section 8) section for an in-depth discussion of how this keyword is used with HTTP.

6.5.5. targetHints

[[CREF1: This section attempts to strike a balance between comprehensiveness and flexibility by deferring most of its structure to the protocol indicated by the URI scheme. Note that a resource can be identified by a URI with a dereferenceable scheme, yet not be accessible over that protocol. While currently very loose, this section is expected to become more well-defined based on draft feedback, and may change significantly in future drafts.]]

The value of this property is advisory only. It represents information that is expected to be discoverable through interacting with the target resource, typically in the form of protocol-specific control information or meta-data such as headers returned in response to an HTTP HEAD or OPTIONS request. The protocol is determined by the "href" URI scheme, although note that resources are not guaranteed to be accessible over such a protocol.

The value of this property SHOULD be an object. The keys to this object SHOULD be lower-cased forms of the control data field names. Each value SHOULD be an array, in order to uniformly handle multi-valued fields. Multiple values MUST be presented as an array, and not as a single string.

Protocols with control information not suitable for representation as a JSON object MAY be represented by another data type, such as an array.

Values that cannot be understood as part of the indicated protocol MUST be ignored by a JSON Hyper-Schema implementation. Applications MAY make use of such values, but MUST NOT assume interoperability with other implementations.

Implementations MUST NOT assume that all discoverable information is accounted for in this object. Client applications MUST properly handle run-time responses that contradict this property's values.

Client applications MUST NOT assume that an implementation will automatically take any action based on the value of this property.

See "JSON Hyper-Schema and HTTP" (Section 8) for guidance on using this keyword with HTTP and analogous protocols.

6.6. Link Input

There are four ways to use client input with a link, and each is addressed by a separate link description object keyword. When performing operations, user agents SHOULD ignore schemas that are not relevant to their semantics.

6.6.1. hrefSchema

The value of the "hrefSchema" link description property MUST be a valid JSON Schema. This schema is used to validate user input or other user agent data for filling out the URI Template in "href" (Section 6.3.1).

Omitting "hrefSchema" or setting the entire schema to "false" prevents any user agent data from being accepted.

Setting any subschema that applies to a particular variable to the JSON literal value "false" prevents any user agent data from being accepted for that single variable.

For template variables that can be resolved from the instance data, if the instance data is valid against all applicable subschemas in "hrefSchema", then it MUST be used to pre-populate the input data set for that variable.

Note that even when data is pre-populated from the instance, the validation schema for that variable in "hrefSchema" need not be identical to the validation schema(s) that apply to the instance data's location. This allows for different validation rules for user agent data, such as supporting spelled-out months for date-time input, but using the standard date-time format for storage.

After input is accepted, potentially overriding the pre-populated instance data, the resulting data set MUST successfully validate against the value of "hrefSchema". If it does not then the link MUST NOT be used. If it is valid, then the process given in the "URI Templating" section continues with this updated data set.

6.6.2. headerSchema

[[CREF2: As with "targetHints", this keyword is somewhat under-specified to encourage experimentation and feedback as we try to balance flexibility and clarity.]]

If present, this property is a schema for protocol-specific request headers or analogous control and meta-data. The value of this object MUST be a valid JSON Schema. The protocol is determined by the "href" URI scheme, although note that resources are not guaranteed to be accessible over such a protocol. The schema is advisory only; the target resource's behavior is not constrained by its presence.

The purpose of this keyword is to advertise target resource interaction features, and indicate to user agents and client applications what headers and header values are likely to be useful. User agents and client applications MAY use the schema to validate relevant headers, but MUST NOT assume that missing headers or values are forbidden from use. While schema authors MAY set "additionalProperties" to false, this is NOT RECOMMENDED and MUST NOT prevent client applications or user agents from supplying additional headers when requests are made.

The exact mapping of the JSON data model into the headers is protocol-dependent. However, in most cases this schema SHOULD specify a type of "object", and the property names SHOULD be lower-cased forms of the control data field names. See the "JSON Hyper-Schema and HTTP" (Section 8) section for detailed guidance on using this keyword with HTTP and analogous protocols.

"headerSchema" is applicable to any request method or command that the protocol supports. When generating a request, user agents and client applications SHOULD ignore schemas for headers that are not relevant to that request.

6.6.3. Manipulating the Target Resource Representation

In JSON Hyper-Schema, "targetSchema" (Section 6.5.4) supplies a non-authoritative description of the target resource's representation. A client application can use "targetSchema" to structure input for replacing or modifying the representation, or as the base representation for building a patch document based on a patch media type.

Alternatively, if "targetSchema" is absent or if the client application prefers to only use authoritative information, it can interact with the target resource to confirm or discover its representation structure.

"targetSchema" is not intended to describe link operation responses, except when the response semantics indicate that it is a representation of the target resource. In all cases, the schema indicated by the response itself is authoritative. See "JSON Hyper-Schema and HTTP" (Section 8) for detailed examples.

6.6.4. Submitting Data for Processing

The "submissionSchema" (Section 6.6.4.2) and "submissionMediaType" (Section 6.6.4.1) keywords describe the domain of the processing function implemented by the target resource. Otherwise, as noted above, the submission schema and media type are ignored for operations to which they are not relevant.

6.6.4.1. submissionMediaType

If present, this property indicates the media type format the client application and user agent should use for the request payload described by "submissionSchema" (Section 6.6.4.2).

Omitting this keyword has the same behavior as a value of application/json.

Note that "submissionMediaType" and "submissionSchema" are not restricted to HTTP URIs. [[CREF3: This statement might move to wherever the example ends up.]]

6.6.4.2. submissionSchema

This property contains a schema which defines the acceptable structure of the document to be encoded according to the "submissionMediaType" property and sent to the target resource for processing. This can be viewed as describing the domain of the processing function implemented by the target resource.

This is a separate concept from the "targetSchema" (Section 6.5.4) property, which describes the target information resource (including for replacing the contents of the resource in a PUT request), unlike "submissionSchema" which describes the user-submitted request data to be evaluated by the resource. "submissionSchema" is intended for use with requests that have payloads that are not necessarily defined in terms of the target representation.

Omitting "submissionSchema" has the same behavior as a value of "true".

7. Implementation Requirements

At a high level, a conforming implementation will meet the following requirements. Each of these requirements is covered in more detail in the individual keyword sections and keyword group overviews.

Note that the requirements around how an implementation MUST recognize "self", "collection", and "item" links are thoroughly covered in the link relation type (Section 6.2) section and are not repeated here.

While it is not a mandatory format for implementations, the output format used in the test suite summarizes what needs to be computed for each link before it can be used:

contextUri The fully resolved URI (with scheme) of the context resource. If the context is not the entire resource and there is a usable fragment identifier syntax, then the URI includes a fragment. Note that there is no such syntax for application/json.

contextPointer The JSON Pointer for the location within the instance of the context resource. If the instance media type supports JSON Pointers as fragment identifiers, this pointer will be the same as the one encoded in the fragment of the "contextUri" field.

rel The link relation type, as it appears in the LDO.

targetUri The fully resolved URI (with a scheme) of the target resource. If the link accepts input, this can only be produced once the input has been supplied.

hrefInputTemplates The list of partially resolved URI references for a link that accepts input. The first entry in the list is the partially resolved "href". The additional entries, if any, are the partially resolved "base" values ordered from the most immediate out to the root of the schema. Template variables that are pre-populated in the input are not resolved at this stage, as the pre-populated value can be overridden.

hrefPrepopulatedInput The data set that the user agent should use to prepopulate any input mechanism before accepting client input. If input is to be accepted but no fields are to be pre-populated, then this will be an empty object.

attachmentPointer The JSON Pointer for the location within the instance to which the link is attached. By default, "contextUri" and "attachmentUri" are the same, but "contextUri" can be changed by LDO keywords, while "attachmentUri" cannot.

Other LDO keywords that are not involved in producing the above information are included exactly as they appear when producing output for the test suite. Those fields will not be further discussed here unless specifically relevant.

7.1. Link Discovery and Look-Up

Before links can be used, they must be discovered by applying the hyper-schema to the instance and finding all applicable and valid links. Note that in addition to collecting valid links, any "base" (Section 5.1) values necessary to resolve each LDO's URI Templates must also be located and associated with the LDO through whatever mechanism is most useful for the implementation's URI Template resolution process.

And implementation MUST support looking up links by either their attachment pointer or context pointer, either by performing the look-up or by providing the set of all links with both pointers determined so that user agents can implement the look-up themselves.

When performing look-ups by context pointer, links that are attached to elements of the same array MUST be returned in the same order as the array elements to which they are attached.

7.2. URI Templating

Three hyper-schema keywords are URI Templates [RFC6570]: "base", "anchor", and "href". Each are resolved separately to URI-references, and then the anchor or href URI-reference is resolved against the base (which is itself resolved against earlier bases as needed, each of which was first resolved from a URI Template to a URI-reference).

All three keywords share the same algorithm for resolving variables from instance data, which makes use of the "templatePointers" and "templateRequired" keywords. When resolving "href", both it and any "base" templates needed for resolution to an absolute URI, the algorithm is modified to optionally accept user input based on the "hrefSchema" keyword.

For each URI Template (T), the following pseudocode describes an algorithm for resolving T into a URI-reference (R). For the purpose of this algorithm:

- "ldo.templatePointers" is an empty object if the keyword was not present and "ldo.templateRequired" is likewise an empty array.

- "attachmentPointer" is the absolute JSON Pointer for the attachment location of the LDO.

- "getApplicableSchemas()" returns an iterable set of all (sub)schemas that apply to the attachment point in the instance.

This algorithm should be applied first to either "href" or "anchor", and then as needed to each successive "base". The order is important, as it is not always possible to tell whether a template will resolve to a full URI or a URI-reference.

In English, the high-level algorithm is:

1. Populate template variable data from the instance

2. If input is desired, accept input

3. Check that all required variables have a value

4. Encode values into strings and fill out the template

This is the high-level algorithm as pseudocode. "T" comes from either "href" or "anchor" within the LDO, or from "base" in a containing schema. Pseudocode for each step follows. "initialTemplateKeyword" indicates which of the two started the process (since "base" is always resolved in order to finish resolving one or the other of those keywords).

```
templateData = populateDataFromInstance(T, ldo, instance)

if initialTemplateKeyword == "href" and ldo.hrefSchema exists:
    inputData = acceptInput(ldo, instance, templateData)
    for varname in inputData:
        templateData[varname] = inputData[varname]

for varname in ldo.templateRequired:
    if not exists templateData[varname]
        fatal("Missing required variable(s)")

templateData = stringEncode(templateData)
R = rfc6570ResolutionAlgorithm(T, templateData)
```

7.2.1. Populating Template Data From the Instance

This step looks at various locations in the instance for variable values. For each variable:

1. Use "templatePointers" to find a value if the variable appears in that keyword's value

2. Otherwise, look for a property name matching the variable in the instance location to which the link is attached

3. In either case, if there is a value at the location, put it in the template resolution data set

```
for varname in T:
    varname = rfc3986PercentDecode(varname)
    if varname in ldo.templatePointers:
        valuePointer = templatePointers[varname]
        if valuePointer is relative:
            valuePointer = resolveRelative(attachmentPointer,
                                           valuePointer)
    else
        valuePointer = attachmentPointer + "/" + varname

    value = instance.valueAt(valuePointer)
    if value is defined:
        templateData[varname] = value
```

7.2.2. Accepting Input for Template Data

This step is relatively complex, as there are several cases to support. Some variables will forbid input and some will allow it. Some will have initial values that need to be presented in the input interface, and some will not.

1. Determine which variables can accept input

2. Pre-populate the input data set if the template resolution data set has a value

3. Accept input (present a web form, make a callback, etc.)

4. Validate the input data set, (not the template resolution data set)

5. Put the input in the template resolution data set, overriding any existing values

"InputForm" represents whatever sort of input mechanism is appropriate. This may be a literal web form, or may be a more programmatic construct such as a callback function accepting specific fields and data types, with the given initial values, if any.

```
form = new InputForm()
for varname in T:
    useField = true
    useInitialData = true
    for schema in getApplicableSchemas(ldo.hrefSchema,
                                       "/" + varname):
        if schema is false:
            useField = false
            break

        if varname in templateData and
           not isValid(templateData[varname], schema)):
            useInitialData = false
            break

    if useField:
        if useInitialData:
            form.addInputFieldFor(varname, ldo.hrefSchema,
                                  templateData[varname])
        else:
            form.addInputFieldFor(varname, ldo.hrefSchema)

inputData = form.acceptInput()

if not isValid(inputData, hrefSchema):
    fatal("Input invalid, link is not usable")

return inputData:
```

7.2.3. Encoding Data as Strings

This section is straightforward, converting literals to their names as strings, and converting numbers to strings in the most obvious manner, and percent-encoding as needed for use in the URI.

```
for varname in templateData:
    value = templateData[varname]
    if value is true:
        templateData[varname] = "true"
    else if value is false:
        temlateData[varname] = "false"
    else if value is null:
        templateData[varname] = "null"
    else if value is a number:
        templateData[varname] =
            bestEffortOriginalJsonString(value)
    else:
        templateData[varname] = rfc3986PercentEncode(value)
```

In some software environments the original JSON representation of a number will not be available (there is no way to tell the difference between 1.0 and 1), so any reasonable representation should be used. Schema and API authors should bear this in mind, and use other types (such as string or boolean) if the exact representation is important. If the number was provide as input in the form of a string, the string used as input SHOULD be used.

7.3. Providing Access to LDO Keywords

For a given link, an implementation MUST make the values of all target attribute keywords directly available to the user agent. Implementations MAY provide additional interfaces for using this information, as discussed in each keyword's section.

For a given link, an implementation MUST make the value of each input schema keyword directly available to the user agent.

To encourage encapsulation of the URI Template resolution process, implementations MAY omit the LDO keywords that are used only to construct URIs. However, implementations MUST provide access to the link relation type.

Unrecognized keywords SHOULD be made available to the user agent, and MUST otherwise be ignored.

7.4. Requests

A hyper-schema implementation SHOULD provide access to all information needed to construct any valid request to the target resource.

The LDO can express all information needed to perform any operation on a link. This section explains what hyper-schema fields a user agent should examine to build requests from any combination of instance data and client input. A hyper-schema implementation is not itself expected to construct and send requests.

Target URI construction rules, including "hrefSchema" for accepting input, are identical for all possible requests.

Requests that do not carry a body payload do not require additional keyword support.

Requests that take a target representation as a payload SHOULD use the "targetSchema" and "targetMediaType" keywords for input description and payload validation. If a protocol allows an operation taking a payload that is based on the representation as modified by a media type (such as a patch media type), then such a media type SHOULD be indicated through "targetHints" in a protocol-specific manner.

Requests that take a payload that is not derived from the target resource's representation SHOULD use the "submissionSchema" and "submissionMediaType" keywords for input description and payload validation. Protocols used in hypermedia generally only support one such non-representation operation per link.

RPC systems that pipe many application operations with arbitrarily different request structures through a single hypermedia protocol operation are outside of the scope of a hypermedia format such as JSON Hyper-Schema.

7.5. Responses

As a hypermedia format, JSON Hyper-Schema is concerned with describing a resource, including describing its links in sufficient detail to make all valid requests. It is not concerned with directly describing all possible responses for those requests.

As in any hypermedia system, responses are expected to be self-describing. In the context of hyper-schema, this means that each response MUST link its own hyper-schema(s). While responses that consist of a representation of the target resource are expected to be valid against "targetSchema" and "targetMediaType", those keywords are advisory only and MUST be ignored if contradicted by the response itself.

Other responses, including error responses, complex redirections, and processing status representations SHOULD also link to their own

schemas and use appropriate media types (e.g. "application/problem+json" [RFC7807] for errors). Certain errors might not link a schema due to being generated by an intermediary that is not aware of hyper-schema, rather than by the origin.

User agents are expected to understand protocol status codes and response media types well enough to handle common situations, and provide enough information to client applications to handle domain-specific responses.

Statically mapping all possible responses and their schemas at design time is outside of the scope of JSON Hyper-Schema, but may be within the scope of other JSON Schema vocabularies which build on hyper-schema (see Appendix A.3).

7.6. Streaming Parsers

The requirements around discovering links based on their context, or using the context of links to identify collections, present unique challenges when used with streaming parsers. It is not possible to authoritatively fulfill these requirements without processing the entire schema and instance documents.

Such implementations MAY choose to return non-authoritative answers based on data processed to date. When offering this approach, implementations MUST be clear on the nature of the response, and MUST offer an option to block and wait until all data is processed and an authoritative answer can be returned.

8. JSON Hyper-Schema and HTTP

While JSON Hyper-Schema is a hypermedia format and therefore protocol-independent, it is expected that its most common use will be in HTTP systems, or systems using protocols such as CoAP that are explicitly analogous to HTTP.

This section provides guidance on how to use each common HTTP method with a link, and how collection resources impose additional constraints on HTTP POST. Additionally, guidance is provided on hinting at HTTP response header values and describing possible HTTP request headers that are relevant to the given resource.

Section 11 of the JSON Schema core specification [json-schema] provides guidance on linking instances in a hypermedia system to their schemas. This may be done with network-accessible schemas, or may simply identify schemas which were pre-packaged within the client application. JSON Hyper-Schema intentionally does not constrain this

mechanism, although it is RECOMMENDED that the techniques outlined in the core specification be used to whatever extent is possible.

8.1. One Link Per Target and Relation Type

Link Description Objects do not directly indicate what operations, such as HTTP methods, are supported by the target resource. Instead, operations should be inferred primarily from link relation types (Section 6.2.1) and URI schemes.

This means that for each target resource and link relation type pair, schema authors SHOULD only define a single LDO. While it is possible to use "allow" with "targetHints" to repeat a relation type and target pair with different HTTP methods marked as allowed, this is NOT RECOMMENDED and may not be well-supported by conforming implementations.

All information necessary to use each HTTP method can be conveyed in a single LDO as explained in this section. The "allow" field in "targetHints" is intended simply to hint at which operations are supported, not to separately define each operation.

Note, however, that a resource may always decline an operation at runtime, for instance due to authorization failure, or due to other application state that controls the operation's availability.

8.2. "targetSchema" and HTTP

"targetSchema" describes the resource on the target end of the link, while "targetMediaType" defines that resource's media type. With HTTP links, "headerSchema" can also be used to describe valid values for use in an "Accept" request header, which can support multiple media types or media ranges. When both ways of indicating the target media type are present, "targetMediaType" SHOULD indicate the default representation media type, while the schema for "accept" in "headerSchema" SHOULD include the default as well as any alternate media types or media ranges that can be requested.

Since the semantics of many HTTP methods are defined in terms of the target resource, "targetSchema" is used for requests and/or responses for several HTTP methods. In particular, "targetSchema" suggests what a client application can expect for the response to an HTTP GET or any response for which the "Content-Location" header is equal to the request URI, and what a client application should send if it replaces the resource in an HTTP PUT request. These correlations are defined by RFC 7231, section 4.3.1 - "GET", section 4.3.4 "PUT", and section 3.1.4.2, "Content-Location" [RFC7231].

Per RFC 5789 [RFC5789], the request structure for an HTTP PATCH is determined by the combination of "targetSchema" and the request media type, which is conveyed by the "Accept-Patch" header, which may be included in "targetHints". Media types that are suitable for PATCH-ing define a syntax for expressing changes to a document, which can be applied to the representation described by "targetSchema" to determine the set of syntactically valid request payloads. Often, the simplest way to validate a PATCH request is to apply it and validate the result as a normal representation.

8.3. HTTP POST and the "submission*" keywords

JSON Hyper-Schema allows for resources that process arbitrary data in addition to or instead of working with the target's representation. This arbitrary data is described by the "submissionSchema" and "submissionMediaType" keywords. In the case of HTTP, the POST method is the only one that handles such data. While there are certain conventions around using POST with collections, the semantics of a POST request are defined by the target resource, not HTTP.

In addition to the protocol-neutral "submission*" keywords (see Section 9.3 for a non-HTTP example), the "Accept-Post" header can be used to specify the necessary media type, and MAY be advertised via the "targetHints" field. [[CREF4: What happens if both are used? Also, "submissionSchema" is a MUST to support, while "targetHints" are at most a SHOULD. But forbidding the use of "Accept-Post" in "targetHints" seems incorrect.]]

Successful responses to POST other than a 201 or a 200 with "Content-Location" set likewise have no HTTP-defined semantics. As with all HTTP responses, any representation in the response should link to its own hyper-schema to indicate how it may be processed. As noted in Appendix A.2, connecting hyperlinks with all possible operation responses is not within the scope of JSON Hyper-Schema.

8.4. Optimizing HTTP Discoverability With "targetHints"

[[CREF5: It would be good to also include a section with CoAP examples.]]

JSON serializations of HTTP response header information SHOULD follow the guidelines established by the work in progress "A JSON Encoding for HTTP Header Field Values" [I-D.reschke-http-jfv]. Approaches shown in that document's examples SHOULD be applied to other similarly structured headers wherever possible.

Headers for all possible HTTP method responses all share "headerSchema". In particular, both headers that appear in a HEAD

response and those that appear in an OPTIONS response can appear. No distinction is made within "headerSchema" as to which method response contains which header.

It is RECOMMENDED that schema authors provide hints for the values of the following types of HTTP headers whenever applicable:

- Method allowance
- Method-specific request media types
- Authentication challenges

In general, headers that are likely to have different values at different times SHOULD NOT be included in "targetHints".

8.5. Advertising HTTP Features With "headerSchema"

Schemas SHOULD be written to describe JSON serializations that follow guidelines established by the work in progress "A JSON Encoding for HTTP Header Field Values" [I-D.reschke-http-jfv] Approaches shown in that document's examples SHOULD be applied to other similarly structured headers wherever possible.

It is RECOMMENDED that schema authors describe the available usage of the following types of HTTP headers whenever applicable:

- Content negotiation
- Authentication and authorization
- Range requests
- The "Prefer" header

Headers such as cache control and conditional request headers are generally implemented by intermediaries rather than the resource, and are therefore not generally useful to describe. While the resource must supply the information needed to use conditional requests, the runtime handling of such headers and related responses is not resource-specific.

8.6. Creating Resources Through Collections

When using HTTP, or a protocol such as CoAP that is explicitly analogous to HTTP, this is done by POST-ing a representation of the individual resource to be created to the collection resource. The

process for recognizing collection and item resources is described in Section 6.2.3.

8.7. Content Negotiation and Schema Evolution

JSON Hyper-Schema facilitates HTTP content negotiation, and allows for a hybrid of the proactive and reactive strategies. As mentioned above, a hyper-schema can include a schema for HTTP headers such as "Accept", "Accept-Charset", "Accept-Language", etc with the "headerSchema" keyword. A user agent or client application can use information in this schema, such as an enumerated list of supported languages, in lieu of making an initial request to start the reactive negotiation process.

In this way, the proactive content negotiation technique of setting these headers can be informed by server information about what values are possible, similar to examining a list of alternatives in reactive negotiation.

For media types that allow specifying a schema as a media type parameter, the "Accept" values sent in a request or advertised in "headerSchema" can include the URI(s) of the schema(s) to which the negotiated representation is expected to conform. One possible use for schema parameters in content negotiation is if the resource has conformed to several different schema versions over time. The client application can indicate what version(s) it understands in the "Accept" header in this way.

9. Examples

This section shows how the keywords that construct URIs and JSON Pointers are used. The results are shown in the format used by the test suite. [[CREF6: Need to post that and link it, but it should be pretty self-explanatory to those of you reviewing things at this stage.]]

Most other keywords are either straightforward ("title" and "description"), apply validation to specific sorts of input, requests, or responses, or have protocol-specific behavior. Examples demonstrating HTTP usage are available in an Appendix (Section 8).

9.1. Entry Point Links, No Templates

For this example, we will assume an example API with a documented entry point URI of https://example.com, which is an empty JSON object with a link to a schema. Here, the entry point has no data of its own and exists only to provide an initial set of links:

```
GET https://api.example.com HTTP/1.1

200 OK
Content-Type: application/json
Link: <https://schema.example.com/entry> rel=describedBy
{}
```

The linked hyper-schema defines the API's base URI and provides two links: an "about" link to API documentation, and a "self" link indicating that this is a schema for the base URI. In this case the base URI is also the entry point URI.

```
{
    "$id": "https://schema.example.com/entry",
    "$schema": "http://json-schema.org/draft-07/hyper-schema#",
    "base": "https://api.example.com/",
    "links": [
        {
            "rel": "self",
            "href": ""
        }, {
            "rel": "about",
            "href": "/docs"
        }
    ]
}
```

These are the simplest possible links, with only a relation type and an "href" with no template variables. They resolve as follows:

```
[
    {
        "contextUri": "https://api.example.com",
        "contextPointer": "",
        "rel": "self",
        "targetUri": "https://api.example.com",
        "attachmentPointer": ""
    },
    {
        "contextUri": "https://api.example.com",
        "contextPointer": "",
        "rel": "about",
        "targetUri": "https://api.example.com/docs",
        "attachmentPointer": ""
    }
]
```

The attachment pointer is the root pointer (the only possibility with an empty object for the instance). The context URI is the default, which is the requested document. Since application/json does not allow for fragments, the context pointer is necessary to fully describe the context. Its default behavior is to be the same as the attachment pointer.

9.2. Individually Identified Resources

Let's add "things" to our system, starting with an individual thing:

```
{
    "$id": "https://schema.example.com/thing",
    "$schema": "http://json-schema.org/draft-07/hyper-schema#",
    "base": "https://api.example.com/",
    "type": "object",
    "required": ["data"],
    "properties": {
        "id": {"$ref": "#/definitions/id"},
        "data": true
    },
    "links": [
        {
            "rel": "self",
            "href": "things/{id}",
            "templateRequired": ["id"],
            "targetSchema": {"$ref": "#"}
        }
    ],
    "definitions": {
        "id": {
            "type": "integer",
            "minimum": 1,
            "readOnly": true
        }
    }
}
```

Our "thing" has a server-assigned id, which is required in order to construct the "self" link. It also has a "data" field which can be of any type. The reason for the "definitions" section will be clear in the next example.

Note that "id" is not required by the validation schema, but is required by the self link. This makes sense: a "thing" only has a URI if it has been created, and the server has assigned an id. However, you can use this schema with an instance containing only the

data field, which allows you to validate "thing" instances that you are about to create.

Let's add a link to our entry point schema that lets you jump directly to a particular thing if you can supply it's id as input. To save space, only the new LDO is shown. Unlike "self" and "about", there is no IANA-registered relationship about hypothetical things, so an extension relationship is defined using the "tag:" URI scheme [RFC4151]:

```
{
    "rel": "tag:rel.example.com,2017:thing",
    "href": "things/{id}",
    "hrefSchema": {
        "required": ["id"],
        "properties": {
            "id": {"$ref": "thing#/definitions/id"}
        }
    },
    "targetSchema": {"$ref": "thing#"}
}
```

The "href" value here is the same, but everything else is different. Recall that the instance is an empty object, so "id" cannot be resolved from instance data. Instead it is required as client input. This LDO could also have used "templateRequired" but with "required" in "hrefSchema" it is not strictly necessary. Providing "templateRequired" without marking "id" as required in "hrefSchema" would lead to errors, as client input is the only possible source for resolving this link.

9.3. Submitting a Payload and Accepting URI Input

This example covers using the "submission" fields for non-representation input, as well as using them alongside of resolving the URI Template with input. Unlike HTML forms, which require either constructing a URI or sending a payload, but do not allow not both at once, JSON Hyper-Schema can describe both sorts of input for the same operation on the same link.

The "submissionSchema" and "submissionMediaType" fields are for describing payloads that are not representations of the target resource. When used with "http(s)://" URIs, they generally refer to a POST request payload, as seen in the appendix on HTTP usage (Section 8).

In this case, we use a "mailto:" URI, which, per RFC 6068, Section 3" [RFC6068], does not provide any operation for retrieving a resource.

It can only be used to construct a message for sending. Since there is no concept of a retrievable, replaceable, or deletable target resource, "targetSchema" and "targetMediaType" are not used. Non-representation payloads are described by "submissionSchema" and "submissionMediaType".

We use "submissionMediaType" to indicate a multipart/alternative payload format, providing two representations of the same data (HTML and plain text). Since a multipart/alternative message is an ordered sequence (the last part is the most preferred alternative), we model the sequence as an array in "submissionSchema". Since each part is itself a document with a media type, we model each item in the array as a string, using "contentMediaType" to indicate the format within the string.

Note that media types such as multipart/form-data, which associate a name with each part and are not ordered, should be modeled as JSON objects rather than arrays.

Note that some lines are wrapped to fit this document's width restrictions.

```
{
    "$id": "https://schema.example.com/interesting-stuff",
    "$schema": "http://json-schema.org/draft-07/hyper-schema#",
    "required": ["stuffWorthEmailingAbout", "email", "title"],
    "properties": {
        "title": {
            "type": "string"
        },
        "stuffWorthEmailingAbout": {
            "type": "string"
        },
        "email": {
            "type": "string",
            "format": "email"
        },
        "cc": false
    },
    "links": [
        {
            "rel": "author",
            "href": "mailto:{email}?subject={title}{&cc}",
            "templateRequired": ["email"],
            "hrefSchema": {
                "required": ["title"],
                "properties": {
                    "title": {
```

```
                        "type": "string"
                    },
                    "cc": {
                        "type": "string",
                        "format": "email"
                    },
                    "email": false
                }
            },
            "submissionMediaType":
                    "multipart/alternative; boundary=ab2",
            "submissionSchema": {
                "type": "array",
                "items": [
                    {
                        "type": "string",
                        "contentMediaType":
                                "text/plain; charset=utf8"
                    },
                    {
                        "type": "string",
                        "contentMediaType": "text/html"
                    }
                ],
                "minItems": 2
            }
        }
    ]
}
```

For the URI parameters, each of the three demonstrates a different way of resolving the input:

email: This variable's presence in "templateRequired" means that it must be resolved for the template to be used. Since the "false" schema assigned to it in "hrefSchema" excludes it from the input data set, it must be resolved from the instance.

title: The instance field matching this variable is required, and it is also allowed in the input data. So its instance value is used to pre-populate the input data set before accepting client input. The client application can opt to leave the instance value in place. Since this field is required in "hrefSchema", the client application cannot delete it (although it could set it to an empty string).

cc: The "false" schema set for this in the main schema prevents this field from having an instance value. If it is present at all, it

must come from client input. As it is not required in "hrefSchema", it may not be used at all.

So, given the following instance retrieved from "https://api.example.com/stuff":

```
{
    "title": "The Awesome Thing",
    "stuffWorthEmailingAbout": "Lots of text here...",
    "email": "someone@exapmle.com"
}
```

We can partially resolve the link as follows, before asking the client application for input.

```
{
    "contextUri": "https://api.example.com/stuff",
    "contextPointer": "",
    "rel": "author",
    "hrefInputTemplates": [
      "mailto:someone@example.com?subject={title}{&cc}"
    ],
    "hrefPrepopulatedInput": {
        "title": "The Really Awesome Thing"
    },
    "attachmentPointer": ""
}
```

Notice the "href*" keywords in place of "targetUri". These are three possible kinds of "targetUri" values covering different sorts of input. Here are examples of each:

No additional or changed input: "mailto:someone@example.com?subject=The%20Awesome%20Thing"

Change "title" to "your work": "mailto:someone@example.com?subject=your%20work"

Change title and add a "cc" of "other@elsewhere.org": "mailto:someone@example.com?subject=your%20work&cc=other@elsewhere.org"

9.4. "anchor", "base" and URI Template Resolution

A link is a typed connection from a context resource to a target resource. Older link serializations support a "rev" keyword that takes a link relation type as "rel" does, but reverses the semantics. This has long been deprecated, so JSON Hyper-Schema does not support

it. Instead, "anchor"'s ability to change the context URI can be used to reverse the direction of a link. It can also be used to describe a link between two resources, neither of which is the current resource.

As an example, there is an IANA-registered "up" relation, but there is no "down". In an HTTP Link header, you could implement "down" as ""rev": "up"".

First let's look at how this could be done in HTTP, showing a "self" link and two semantically identical links, one with "rev": "up" and the other using "anchor" with "rel": "up" (line wrapped due to formatting limitations).

```
GET https://api.example.com/trees/1/nodes/123 HTTP/1.1

200 OK
Content-Type: application/json
Link: <https://api.example.com/trees/1/nodes/123> rel=self
Link: <https://api.example.com/trees/1/nodes/123> rel=up
        anchor=<https://api.example.com/trees/1/nodes/456>
Link: <https://api.example.com/trees/1/nodes/456> rev=up
{
    "id": 123,
    "treeId": 1,
    "childIds": [456]
}
```

Note that the "rel=up" link has a target URI identical to the "rel=self" link, and sets "anchor" (which identifies the link's context) to the child's URI. This sort of reversed link is easily detectable by tools when a "self" link is also present.

The following hyper-schema, applied to the instance in the response above, would produce the same "self" link and "up" link with "anchor". It also shows the use of a templated "base" URI, plus both absolute and relative JSON Pointers in "templatePointers".

```
{
    "$id": "https://schema.example.com/tree-node",
    "$schema": "http://json-schema.org/draft-07/hyper-schema#",
    "base": "trees/{treeId}/",
    "properties": {
        "id": {"type": "integer"},
        "treeId": {"type": "integer"},
        "childIds": {
            "type": "array",
            "items": {
                "type": "integer",
                "links": [
                    {
                        "anchor": "nodes/{thisNodeId}",
                        "rel": "up",
                        "href": "nodes/{childId}",
                        "templatePointers": {
                            "thisNodeId": "/id",
                            "childId": "0"
                        }
                    }
                ]
            }
        }
    },
    "links": [
        {
            "rel": "self",
            "href": "nodes/{id}"
        }
    ]
}
```

The "base" template is evaluated identically for both the target ("href") and context ("anchor") URIs.

Note the two different sorts of templatePointers used. "thisNodeId" is mapped to an absolute JSON Pointer, "/id", while "childId" is mapped to a relative pointer, "0", which indicates the value of the current item. Absolute JSON Pointers do not support any kind of wildcarding, so there is no way to specify a concept like "current item" without a relative JSON Pointer.

9.5. Collections

In many systems, individual resources are grouped into collections. Those collections also often provide a way to create individual item resources with server-assigned identifiers.

For this example, we will re-use the individual thing schema as shown in an earlier section. It is repeated here for convenience, with an added "collection" link with a "targetSchema" reference pointing to the collection schema we will introduce next.

```
{
    "$id": "https://schema.example.com/thing",
    "$schema": "http://json-schema.org/draft-07/hyper-schema#",
    "base": "https://api.example.com/",
    "type": "object",
    "required": ["data"],
    "properties": {
        "id": {"$ref": "#/definitions/id"},
        "data": true
    },
    "links": [
        {
            "rel": "self",
            "href": "things/{id}",
            "templateRequired": ["id"],
            "targetSchema": {"$ref": "#"}
        }, {
            "rel": "collection",
            "href": "/things",
            "targetSchema": {"$ref": "thing-collection#"},
            "submissionSchema": {"$ref": "#"}
        }
    ],
    "definitions": {
        "id": {
            "type": "integer",
            "minimum": 1,
            "readOnly": true
        }
    }
}
```

The "collection" link is the same for all items, so there are no URI Template variables. The "submissionSchema" is that of the item itself. As described in Section 6.2.3, if a "collection" link supports a submission mechanism (POST in HTTP) then it MUST implement item creation semantics. Therefore "submissionSchema" is the schema for creating a "thing" via this link.

Now we want to describe collections of "thing"s. This schema describes a collection where each item representation is identical to the individual "thing" representation. While many collection representations only include subset of the item representations, this example uses the entirety to minimize the number of schemas involved. The actual collection items appear as an array within an object, as we will add more fields to the object in the next example.

```
{
    "$id": "https://schema.example.com/thing-collection",
    "$schema": "http://json-schema.org/draft-07/hyper-schema#",
    "base": "https://api.example.com/",
    "type": "object",
    "required": ["elements"],
    "properties": {
        "elements": {
            "type": "array",
            "items": {
                "allOf": [{"$ref": "thing#"}],
                "links": [
                    {
                        "anchorPointer": "",
                        "rel": "item",
                        "href": "things/{id}",
                        "templateRequired": ["id"],
                        "targetSchema": {"$ref": "thing#"}
                    }
                ]
            }
        }
    },
    "links": [
        {
            "rel": "self",
            "href": "things",
            "targetSchema": {"$ref": "#"},
            "submissionSchema": {"$ref": "thing"}
        }
    ]
}
```

Here is a simple two-element collection instance:

```
{
    "elements": [
        {"id": 12345, "data": {}},
        {"id": 67890, "data": {}}
    ]
}
```

Here are all of the links that apply to this instance, including those that are defined in the referenced individual "thing" schema:

```
[
    {
        "contextUri": "https://api.example.com/things",
        "contextPointer": "",
        "rel": "self",
        "targetUri": "https://api.example.com/things",
        "attachmentPointer": ""
    },
    {
        "contextUri": "https://api.example.com/things",
        "contextPointer": "/elements/0",
        "rel": "self",
        "targetUri": "https://api.example.com/things/12345",
        "attachmentPointer": "/elements/0"
    },
    {
        "contextUri": "https://api.example.com/things",
        "contextPointer": "/elements/1",
        "rel": "self",
        "targetUri": "https://api.example.com/things/67890",
        "attachmentPointer": "/elements/1"
    },
    {
        "contextUri": "https://api.example.com/things",
        "contextPointer": "",
        "rel": "item",
        "targetUri": "https://api.example.com/things/12345",
        "attachmentPointer": "/elements/0"
    },
    {
        "contextUri": "https://api.example.com/things",
        "contextPointer": "",
        "rel": "item",
        "targetUri": "https://api.example.com/things/67890",
        "attachmentPointer": "/elements/1"
    },
```

```
    {
        "contextUri": "https://api.example.com/things",
        "contextPointer": "/elements/0",
        "rel": "collection",
        "targetUri": "https://api.example.com/things",
        "attachmentPointer": "/elements/0"
    },
    {
        "contextUri": "https://api.example.com/things",
        "contextPointer": "/elements/1",
        "rel": "collection",
        "targetUri": "https://api.example.com/things",
        "attachmentPointer": "/elements/1"
    }
]
```

In all cases, the context URI is shown for an instance of media type application/json, which does not support fragments. If the instance media type was application/instance+json, which supports JSON Pointer fragments, then the context URIs would contain fragments identical to the context pointer field. For application/json and other media types without fragments, it is critically important to consider the context pointer as well as the context URI.

There are three "self" links, one for the collection, and one for each item in the "elements" array. The item "self" links are defined in the individual "thing" schema which is referenced with "$ref". The three links can be distinguished by their context or attachment pointers. We will revisit the "submissionSchema" of the collection's "self" link in Section 9.5.2.

There are two "item" links, one for each item in the "elements" array. Unlike the "self" links, these are defined only in the collection schema. Each of them have the same target URI as the corresponding "self" link that shares the same attachment pointer. However, each has a different context pointer. The context of the "self" link is the entry in "elements", while the context of the "item" link is always the entire collection regardless of the specific item.

Finally, there are two "collection" links, one for each item in "elements". In the individual item schema, these produce links with the item resource as the context. When referenced from the collection schema, the context is the location in the "elements" array of the relevant "thing", rather than that "thing"'s own separate resource URI.

The collection links have identical target URIs as there is only one relevant collection URI. While calculating both links as part of a full set of constructed links may not seem useful, when constructing links on an as-needed basis, this arrangement means that there is a "collection" link definition close at hand no matter which "elements" entry you are processing.

9.5.1. Pagination

Here we add pagination to our collection. There is a "meta" section to hold the information about current, next, and previous pages. Most of the schema is the same as in the previous section and has been omitted. Only new fields and new or (in the case of the main "self" link) changed links are shown in full.

```
{
    "properties": {
        "elements": {
            ...
        },
        "meta": {
            "type": "object",
            "properties": {
                "prev": {"$ref": "#/definitions/pagination"},
                "current": {"$ref": "#/definitions/pagination"},
                "next": {"$ref": "#/definitions/pagination"}
            }
        }
    },
    "links": [
        {
            "rel": "self",
            "href": "things{?offset,limit}",
            "templateRequired": ["offset", "limit"],
            "templatePointers": {
                "offset": "/meta/current/offset",
                "limit": "/meta/current/limit"
            },
            "targetSchema": {"$ref": "#"}
        }, {
            "rel": "prev",
            "href": "things{?offset,limit}",
            "templateRequired": ["offset", "limit"],
            "templatePointers": {
                "offset": "/meta/prev/offset",
                "limit": "/meta/prev/limit"
            },
            "targetSchema": {"$ref": "#"}
```

```
            }, {
                "rel": "next",
                "href": "things{?offset,limit}",
                "templateRequired": ["offset", "limit"],
                "templatePointers": {
                    "offset": "/meta/next/offset",
                    "limit": "/meta/next/limit"
                },
                "targetSchema": {"$ref": "#"}
            }
        ],
        "definitions": {
            "pagination": {
                "type": "object",
                "properties": {
                    "offset": {
                        "type": "integer",
                        "minimum": 0,
                        "default": 0
                    },
                    "limit": {
                        "type": "integer",
                        "minimum": 1,
                        "maximum": 100,
                        "default": 10
                    }
                }
            }
        }
    }
```

Notice that the "self" link includes the pagination query that produced the exact representation, rather than being a generic link to the collection allowing selecting the page via input.

Given this instance:

```
{
    "elements": [
        {"id": 12345, "data": {}},
        {"id": 67890, "data": {}}
    ],
    "meta": {
        "current": {
            "offset": 0,
            "limit": 2
        },
        "next": {
            "offset": 3,
            "limit": 2
        }
    }
}
```

Here are all of the links that apply to this instance that either did not appear in the previous example or have been changed with pagination added.

```
[
    {
        "contextUri": "https://api.example.com/things",
        "contextPointer": "",
        "rel": "self",
        "targetUri":
            "https://api.example.com/things?offset=20,limit=2",
        "attachmentPointer": ""
    },
    {
        "contextUri": "https://api.example.com/things",
        "contextPointer": "",
        "rel": "next",
        "targetUri":
            "https://api.example.com/things?offset=22,limit=2",
        "attachmentPointer": ""
    }
]
```

Note that there is no "prev" link in the output, as we are looking at the first page. The lack of a "prev" field under "meta", together with the "prev" link's "templateRequired" values, means that the link is not usable with this particular instance.

[[CREF7: It's not clear how pagination should work with the link from the "collection" links in the individual "thing" schema. Technically, a link from an item to a paginated or filtered collection should go to a page/filter that contains the item (in this case the "thing") that is the link context. See GitHub issue #421 for more discussion.]]

Let's add a link for this collection to the entry point schema (Section 9.1), including pagination input in order to allow client applications to jump directly to a specific page. Recall that the entry point schema consists only of links, therefore we only show the newly added link:

```
{
    "rel": "tag:rel.example.com,2017:thing-collection",
    "href": "/things{?offset,limit}",
    "hrefSchema": {
        "$ref": "thing-collection#/definitions/pagination"
    },
    "submissionSchema": {
        "$ref": "thing#"
    },
    "targetSchema": {
        "$ref": "thing-collection#"
    }
}
```

Now we see the pagination parameters being accepted as input, so we can jump to any page within the collection. The link relation type is a custom one as the generic "collection" link can only be used with an item as its context, not an entry point or other resource.

9.5.2. Creating the First Item

When we do not have any "thing"s, we do not have any resources with a relevant "collection" link. Therefore we cannot use a "collection" link's submission keywords to create the first "thing"; hyper-schemas must be evaluated with respect to an instance. Since the "elements" array in the collection instance would be empty, it cannot provide us with a collection link either.

However, our entry point link can take us to the empty collection, and we can use the presence of "item" links in the hyper-schema to recognize that it is a collection. Since the context of the "item" link is the collection, we simply look for a "self" link with the same context, which we can then treat as collection for the purposes of a creation operation.

Presumably, our custom link relation type in the entry point schema was sufficient to ensure that we have found the right collection. A client application that recognizes that custom link relation type may know that it can immediately assume that the target is a collection, but a generic user agent cannot do so. Despite the presence of a "-collection" suffix in our example, a generic user agent would have no way of knowing whether that substring indicates a hypermedia resource collection, or some other sort of collection.

Once we have recognized the "self" link as being for the correct collection, we can use its "submissionSchema" and/or "submissionMediaType" keywords to perform an item creation operation. [[CREF8: This works perfectly if the collection is unfiltered and unpaginated. However, one should generally POST to a collection that will contain the created resource, and a "self" link MUST include any filters, pagination, or other query parameters. Is it still valid to POST to such a "self" link even if the resulting item would not match the filter or appear within that page? See GitHub issue #421 for further discussion.]] [[CREF9: Draft-04 of Hyper-Schema defined a "create" link relation that had the schema, rather than the instance, as its context. This did not fit into the instance-based link model, and incorrectly used an operation name for a link relation type. However, defining a more correctly designed link from the schema to the collection instance may be one possible approach to solving this. Again, see GitHub issue #421 for more details.]]

10. Security Considerations

JSON Hyper-Schema defines a vocabulary for JSON Schema core and concerns all the security considerations listed there. As a link serialization format, the security considerations of RFC 8288 Web Linking [RFC8288] also apply, with appropriate adjustments (e.g. "anchor" as an LDO keyword rather than an HTTP Link header attribute).

10.1. Target Attributes

As stated in Section 6.5, all LDO keywords describing the target resource are advisory and MUST NOT be used in place of the authoritative information supplied by the target resource in response to an operation. Target resource responses SHOULD indicate their own hyper-schema, which is authoritative.

If the hyper-schema in the target response matches (by "$id") the hyper-schema in which the current LDO was found, then the target attributes MAY be considered authoritative. [[CREF10: Need to add something about the risks of spoofing by "$id", but given that other

parts of the specification discourage always re-downloading the linked schema, the risk mitigation options are unclear.]]

User agents or client applications MUST NOT use the value of "targetSchema" to aid in the interpretation of the data received in response to following the link, as this leaves "safe" data open to re-interpretation.

When choosing how to interpret data, the type information provided by the server (or inferred from the filename, or any other usual method) MUST be the only consideration, and the "targetMediaType" property of the link MUST NOT be used. User agents MAY use this information to determine how they represent the link or where to display it (for example hover-text, opening in a new tab). If user agents decide to pass the link to an external program, they SHOULD first verify that the data is of a type that would normally be passed to that external program.

This is to guard against re-interpretation of "safe" data, similar to the precautions for "targetSchema".

Protocol meta-data values conveyed in "targetHints" MUST NOT be considered authoritative. Any security considerations defined by the protocol that may apply based on incorrect assumptions about meta-data values apply.

Even when no protocol security considerations are directly applicable, implementations MUST be prepared to handle responses that do not match the link's "targetHints" values.

10.2. "self" Links

When link relation of "self" is used to denote a full representation of an object, the user agent SHOULD NOT consider the representation to be the authoritative representation of the resource denoted by the target URI if the target URI is not equivalent to or a sub-path of the URI used to request the resource representation which contains the target URI with the "self" link. [[CREF11: It is no longer entirely clear what was intended by the "sub-path" option in this paragraph. It may have been intended to allow "self" links for embedded item representations in a collection, which usually have target URIs that are sub-paths of that collection's URI, to be considered authoritative. However, this is simply a common design convention and does not appear to be based in RFC 3986 or any other guidance on URI usage. See GitHub issue #485 for further discussion.]]

11. Acknowledgments

Thanks to Gary Court, Francis Galiegue, Kris Zyp, and Geraint Luff for their work on the initial drafts of JSON Schema.

Thanks to Jason Desrosiers, Daniel Perrett, Erik Wilde, Ben Hutton, Evgeny Poberezkin, Brad Bowman, Gowry Sankar, Donald Pipowitch, Dave Finlay, and Denis Laxalde for their submissions and patches to the document.

12. References

12.1. Normative References

[json-schema]
Wright, A. and H. Andrews, "JSON Schema: A Media Type for Describing JSON Documents", draft-handrews-json-schema-00 (work in progress), November 2017.

[json-schema-validation]
Wright, A., Andrews, H., and G. Luff, "JSON Schema Validation: A Vocabulary for Structural Validation of JSON", draft-handrews-json-schema-validation-00 (work in progress), November 2017.

[relative-json-pointer]
Luff, G. and H. Andrews, "Relative JSON Pointers", draft-handrews-relative-json-pointer-01 (work in progress), January 2018.

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

[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/info/rfc3986>.

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

[RFC6570] Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and D. Orchard, "URI Template", RFC 6570, DOI 10.17487/RFC6570, March 2012, <https://www.rfc-editor.org/info/rfc6570>.

[RFC6573] Amundsen, M., "The Item and Collection Link Relations", RFC 6573, DOI 10.17487/RFC6573, April 2012, <https://www.rfc-editor.org/info/rfc6573>.

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

[RFC8288] Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017, <https://www.rfc-editor.org/info/rfc8288>.

12.2. Informative References

[I-D.reschke-http-jfv] Reschke, J., "A JSON Encoding for HTTP Header Field Values", draft-reschke-http-jfv-06 (work in progress), June 2017.

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, DOI 10.17487/RFC2046, November 1996, <https://www.rfc-editor.org/info/rfc2046>.

[RFC4151] Kindberg, T. and S. Hawke, "The 'tag' URI Scheme", RFC 4151, DOI 10.17487/RFC4151, October 2005, <https://www.rfc-editor.org/info/rfc4151>.

[RFC5789] Dusseault, L. and J. Snell, "PATCH Method for HTTP", RFC 5789, DOI 10.17487/RFC5789, March 2010, <https://www.rfc-editor.org/info/rfc5789>.

[RFC6068] Duerst, M., Masinter, L., and J. Zawinski, "The 'mailto' URI Scheme", RFC 6068, DOI 10.17487/RFC6068, October 2010, <https://www.rfc-editor.org/info/rfc6068>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

[RFC7807] Nottingham, M. and E. Wilde, "Problem Details for HTTP APIs", RFC 7807, DOI 10.17487/RFC7807, March 2016, <https://www.rfc-editor.org/info/rfc7807>.

Appendix A. Using JSON Hyper-Schema in APIs

Hypermedia APIs, which follow the constraints of the REST architectural style, enable the creation of generic user agents. Such a user agent has no application-specific knowledge. Rather, it understands pre-defined media types, URI schemes, protocols, and link relations, often by recognizing these and coordinating the use of existing software that implements support for them. Client applications can then be built on top of such a user agent, focusing on their own semantics and logic rather than the mechanics of the interactions.

Hyper-schema is only concerned with one resource and set of associated links at a time. Just as a web browser works with only one HTML page at a time, with no concept of whether or how that page functions as part of a "site", a hyper-schema-aware user agent works with one resource at a time, without any concept of whether or how that resource fits into an API.

Therefore, hyper-schema is suitable for use within an API, but is not suitable for the description of APIs as complete entities in their own right. There is no way to describe concepts at the API scope, rather than the resource and link scope, and such descriptions are outside of the boundaries of JSON Hyper-Schema.

A.1. Resource Evolution With Hyper-Schema

Since a given JSON Hyper-Schema is used with a single resource at a single point in time, it has no inherent notion of versioning. However, a given resource can change which schema or schemas it uses over time, and the URIs of these schemas can be used to indicate versioning information. When used with a media type that supports indicating a schema with a media type parameter, these versioned schema URIs can be used in content negotiation.

A resource can indicate that it is an instance of multiple schemas, which allows supporting multiple compatible versions simultaneously. A client application can then make use of the hyper-schema that it recognizes, and ignore newer or older versions.

A.2. Responses and Errors

Because a hyper-schema represents a single resource at a time, it does not provide for an enumeration of all possible responses to protocol operations performed with links. Each response, including errors, is considered its own (possibly anonymous) resource, and should identify its own hyper-schema, and optionally use an appropriate media type such as RFC 7807's "application/problem+json"

[RFC7807], to allow the user agent or client application to interpret any information that is provided beyond the protocol's own status reporting.

A.3. Static Analysis of an API's Hyper-Schemas

It is possible to statically analyze a set of hyper-schemas without instance data in order to generate output such as documentation or code. However, the full feature set of both validation and hyper-schema cannot be accessed without runtime instance data.

This is an intentional design choice to provide the maximum runtime flexibility for hypermedia systems. JSON Schema as a media type allows for establishing additional vocabularies for static analysis and content generation, which are not addressed in this specification. Additionally, individual systems may restrict their usage to subsets that can be analyzed statically if full design-time description is a goal. [[CREF12: Vocabularies for API documentation and other purposes have been proposed, and contributions are welcome at https://github.com/json-schema-org/json-schema-vocabularies]]

Appendix B. ChangeLog

[[CREF13: This section to be removed before leaving Internet-Draft status.]]

draft-handrews-json-schema-hyperschema-01

* This draft is purely a bug fix with no functional changes
* Fixed erroneous meta-schema URI (draft-07, not draft-07-wip)
* Removed stray "work in progress" language left over from review period
* Fixed missing trailing "/" in various "base" examples
* Fixed incorrect draft name in changelog (luff-*-00, not -01)
* Update relative pointer ref to handrews-*-01, also purely a bug fix

draft-handrews-json-schema-hyperschema-00

* Top to bottom reorganization and rewrite
* Group keywords per RFC 8288 context/relation/target/target attributes

* Additional keyword groups for template resolution and describing input
* Clarify implementation requirements with a suggested output format
* Expanded overview to provide context
* Consolidated examples into their own section, illustrate real-world patterns
* Consolidated HTTP guidance in its own section
* Added a subsection on static analysis of hyper-schemas
* Consolidated security concerns in their own section
* Added an appendix on usage in APIs
* Moved "readOnly" to the validation specification
* Moved "media" to validation as "contentMediaType"/"contentEncoding"
* Renamed "submissionEncType" to "submissionMediaType"
* Renamed "mediaType" to "targetMediaType"
* Added "anchor" and "anchorPointer"
* Added "templatePointers" and "templateRequired"
* Clarified how "hrefSchema" is used
* Added "targetHints" and "headerSchema"
* Added guidance on "self", "collection" and "item" link usage
* Added "description" as an LDO keyword
* Added "$comment" in LDOs to match the schema keyword

draft-wright-json-schema-hyperschema-01

* Fixed examples
* Added "hrefSchema" for user input to "href" URI Templates

* Removed URI Template pre-processing
* Clarified how links and data submission work
* Clarified how validation keywords apply hyper-schema keywords and links
* Clarified HTTP use with "targetSchema"
* Renamed "schema" to "submissionSchema"
* Renamed "encType" to "submissionEncType"
* Removed "method"

draft-wright-json-schema-hyperschema-00

* "rel" is now optional
* rel="self" no longer changes URI base
* Added "base" keyword to change instance URI base
* Removed "root" link relation
* Removed "create" link relation
* Removed "full" link relation
* Removed "instances" link relation
* Removed special behavior for "describedBy" link relation
* Removed "pathStart" keyword
* Removed "fragmentResolution" keyword
* Updated references to JSON Pointer, HTML
* Changed behavior of "method" property to align with hypermedia best current practices

draft-luff-json-hyper-schema-00

* Split from main specification.

Authors' Addresses

Henry Andrews (editor)
Cloudflare, Inc.
San Francisco, CA
USA

EMail: henry@cloudflare.com

Austin Wright (editor)

EMail: aaa@bzfx.net