OpenAPI Specification v3.1.1

Version 3.1.1

Published 24 October 2024

▼ **More details about this document**

This version:

https://spec.openapis.org/oas/v3.1.1.html

Latest published version:

https://spec.openapis.org/oas/latest.html

Latest editor's draft:

https://github.com/OAI/OpenAPI-Specification/

Editors:

Darrel Miller
Henry Andrews
Jeremy Whitlock
Lorna Mitchell
Marsh Gardiner
Miguel Quintero
Mike Kistler
Ralf Handl
Ron Ratovsky

Former editors:

Mike Ralphson
Uri Sarid
Jason Harmon
Tony Tam

Other versions:

https://spec.openapis.org/oas/v3.1.0.html
https://spec.openapis.org/oas/v3.0.4.html
https://spec.openapis.org/oas/v3.0.3.html
https://spec.openapis.org/oas/v3.0.2.html
https://spec.openapis.org/oas/v3.0.1.html
https://spec.openapis.org/oas/v3.0.0.html
https://spec.openapis.org/oas/v2.0.html

Participate

GitHub OAI/OpenAPI-Specification
File a bug

Commit history
Pull requests

What is the OpenAPI Specification?

The OpenAPI Specification (OAS) defines a standard, programming language-agnostic interface description for HTTP APIs, which allows both humans and computers to discover and understand the capabilities of a service without requiring access to source code, additional documentation, or inspection of network traffic. When properly defined via OpenAPI, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Similar to what interface descriptions have done for lower-level programming, the OpenAPI Specification removes guesswork in calling a service.

Status of This Document

The source-of-truth for this specification is the HTML file referenced above as *This version*.

Table of Contents

1. OpenAPI Specification

1.1 Version 3.1.1

The key words "*MUST*", "*MUST NOT*", "*REQUIRED*", "*SHALL*", "*SHALL NOT*", "*SHOULD*", "*SHOULD NOT*", "*RECOMMENDED*", "*NOT RECOMMENDED*", "*MAY*", and "*OPTIONAL*" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Introduction

The OpenAPI Specification (OAS) defines a standard, language-agnostic interface to HTTP APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic.

An OpenAPI Description can then be used by documentation generation tools to display the API, code generation tools to generate servers and clients in various programming languages, testing tools, and many other use cases.

For examples of OpenAPI usage and additional documentation, please visit [OpenAPI-Learn].

For extension registries and other specifications published by the OpenAPI Initiative, as well as the authoritative rendering of this specification, please visit spec.openapis.org.

3. *Definitions*

3.1 *OpenAPI Description*

An OpenAPI Description (OAD) formally describes the surface of an API and its semantics. It is composed of an entry document, which must be an OpenAPI Document, and any/all of its referenced documents. An OAD uses and conforms to the OpenAPI Specification, and *MUST* contain at least one paths field, components field, or webhooks field.

3.2 *OpenAPI Document*

An OpenAPI Document is a single JSON or YAML document that conforms to the OpenAPI Specification. An OpenAPI Document compatible with OAS 3.*.* contains a required `openapi` field which designates the version of the OAS that it uses.

3.3 *Schema*

A "schema" is a formal description of syntax and structure. This document serves as the schema for the OpenAPI Specification format; a non-authoritative JSON Schema based on this document is also provided on spec.openapis.org for informational purposes. This specification also *uses* schemas in the form of the Schema Object.

3.4 *Object*

When capitalized, the word "Object" refers to any of the Objects that are named by section headings in this document.

§ 3.5 *Path Templating*

Path templating refers to the usage of template expressions, delimited by curly braces ({}), to mark a section of a URL path as replaceable using path parameters.

Each template expression in the path *MUST* correspond to a path parameter that is included in the Path Item itself and/or in each of the Path Item's Operations. An exception is if the path item is empty, for example due to ACL constraints, matching path parameters are not required.

The value for these path parameters *MUST NOT* contain any unescaped "generic syntax" characters described by [RFC3986] Section 3: forward slashes (/), question marks (?), or hashes (#).

§ 3.6 *Media Types*

Media type definitions are spread across several resources. The media type definitions *SHOULD* be in compliance with [RFC6838].

Some examples of possible media type definitions:

```
text/plain; charset=utf-8
application/json
application/vnd.github+json
application/vnd.github.v3+json
application/vnd.github.v3.raw+json
application/vnd.github.v3.text+json
application/vnd.github.v3.html+json
application/vnd.github.v3.full+json
application/vnd.github.v3.diff
application/vnd.github.v3.patch
```

§ 3.7 *HTTP Status Codes*

The HTTP Status Codes are used to indicate the status of the executed operation. Status codes *SHOULD* be selected from the available status codes registered in the IANA Status Code Registry.

§ 3.8 *Case Sensitivity*

As most field names and values in the OpenAPI Specification are case-sensitive, this document endeavors to call out any case-insensitive names and values. However, the case sensitivity of field names and values that map directly to HTTP concepts follow the case sensitivity rules of HTTP, even if this document does not make a note of every concept.

§ 3.9 *Undefined and Implementation-Defined Behavior*

This specification deems certain situations to have either *undefined* or *implementation-defined* behavior.

Behavior described as *undefined* is likely, at least in some circumstances, to result in outcomes that contradict the specification. This description is used when detecting the contradiction is impossible or impractical. Implementations *MAY* support undefined scenarios for historical reasons, including ambiguous text in prior versions of the specification. This support might produce correct outcomes in many cases, but relying on it is *NOT RECOMMENDED* as there is no guarantee that it will work across all tools or with future specification versions, even if those versions are otherwise strictly compatible with this one.

Behavior described as *implementation-defined* allows implementations to choose which of several different-but-compliant approaches to a requirement to implement. This documents ambiguous requirements that API description authors are *RECOMMENDED* to avoid in order to maximize interoperability. Unlike undefined behavior, it is safe to rely on implementation-defined behavior if *and only if* it can be guaranteed that all relevant tools support the same behavior.

§ 4. Specification

§ 4.1 Versions

The OpenAPI Specification is versioned using a `major.minor.patch` versioning scheme. The `major.minor` portion of the version string (for example `3.1`) *SHALL* designate the OAS feature set. *`.patch`* versions address errors in, or provide clarifications to, this document, not the feature set. Tooling which supports OAS 3.1 *SHOULD* be compatible with all OAS 3.1.* versions. The patch version *SHOULD NOT* be considered by tooling, making no distinction between `3.1.0` and `3.1.1` for example.

Occasionally, non-backwards compatible changes may be made in `minor` versions of the OAS where impact is believed to be low relative to the benefit provided.

§ 4.2 Format

An OpenAPI Document that conforms to the OpenAPI Specification is itself a JSON object, which may be represented either in JSON or YAML format.

For example, if a field has an array value, the JSON array representation will be used:

```
{
  "field": [1, 2, 3]
}
```

All field names in the specification are **case sensitive**. This includes all fields that are used as keys in a map, except where explicitly noted that keys are **case insensitive**.

The schema exposes two types of fields: *fixed fields*, which have a declared name, and *patterned fields*, which have a declared pattern for the field name.

Patterned fields *MUST* have unique names within the containing object.

In order to preserve the ability to round-trip between YAML and JSON formats, YAML version 1.2 is *RECOMMENDED* along with some additional constraints:

- Tags *MUST* be limited to those allowed by YAML's JSON schema ruleset, which defines a subset of the YAML syntax and is unrelated to JSON Schema.
- Keys used in YAML maps *MUST* be limited to a scalar string, as defined by the YAML Failsafe schema ruleset.

Note: While APIs may be described by OpenAPI Descriptions in either YAML or JSON format, the API request and response bodies and other content are not required to be JSON or YAML.

§ 4.3 OpenAPI Description Structure

An OpenAPI Description (OAD) *MAY* be made up of a single JSON or YAML document or be divided into multiple, connected parts at the discretion of the author. In the latter case, Reference Object, Path Item Object and Schema Object `$ref` fields, as well as the Link Object `operationRef` field, and the URI form of the Discriminator Object `mapping` field, are used to identify the referenced elements.

In a multi-document OAD, the document containing the OpenAPI Object where parsing begins is known as that OAD's **entry document**.

It is *RECOMMENDED* that the entry document of an OAD be named: `openapi.json` or `openapi.yaml`.

§ 4.3.1 Parsing Documents

In order to properly handle Schema Objects, OAS 3.1 inherits the parsing requirements of JSON Schema Specification Draft 2020-12, with appropriate modifications regarding base URIs as specified in Relative References In URIs.

This includes a requirement to parse complete documents before deeming a Schema Object reference to be unresolvable, in order to detect keywords that might provide the reference target or impact the determination of the appropriate base URI.

Implementations *MAY* support complete-document parsing in any of the following ways:

- Detecting OpenAPI or JSON Schema documents using media types
- Detecting OpenAPI documents through the root `openapi` field
- Detecting JSON Schema documents through detecting keywords or otherwise successfully parsing the document in accordance with the JSON Schema specification
- Detecting a document containing a referenceable Object at its root based on the expected type of the reference
- Allowing users to configure the type of documents that might be loaded due to a reference to a non-root Object

Implementations that parse referenced fragments of OpenAPI content without regard for the content of the rest of the containing document will miss keywords that change the meaning and behavior of the reference target. In particular, failing to take into account keywords that change the base URI introduces security risks by causing references to resolve to unintended URIs, with unpredictable results. While some implementations support this sort of parsing due to the requirements of past versions of this specification, in version 3.1, the result of parsing fragments in isolation is *undefined* and likely to contradict the requirements of this specification.

While it is possible to structure certain OpenAPI Descriptions to ensure that they will behave correctly when references are parsed as isolated fragments, depending on this is *NOT RECOMMENDED*. This specification does not explicitly enumerate the conditions under which such behavior is safe and provides no guarantee for continued safety in any future versions of the OAS.

A special case of parsing fragments of OAS content would be if such fragments are embedded in another format, referred to as an *embedding format* with respect to the OAS. Note that the OAS itself is an embedding format with respect to JSON Schema, which is embedded as Schema Objects. It is the responsibility of an embedding format to define how to parse embedded content, and OAS implementations that do not document support for an embedding format cannot be expected to parse embedded OAS content correctly.

§ 4.3.2 Structural Interoperability

JSON or YAML objects within an OAD are interpreted as specific Objects (such as Operation Objects, Response Objects, Reference Objects, etc.) based on their context. Depending on how references are arranged, a given JSON or YAML object can be interpreted in multiple different contexts:

- As the root object of the entry document, which is always interpreted as an OpenAPI Object
- As the Object type implied by its parent Object within the document
- As a reference target, with the Object type matching the reference source's context

If the same JSON/YAML object is parsed multiple times and the respective contexts require it to be parsed as *different* Object types, the resulting behavior is *implementation defined*, and *MAY* be treated as an error if detected. An example would be referencing an empty Schema Object under `#/components/schemas` where a Path Item Object is expected, as an empty object is valid for both types. For maximum interoperability, it is *RECOMMENDED* that OpenAPI Description authors avoid such scenarios.

§ 4.3.3 Resolving Implicit Connections

Several features of this specification require resolution of non-URI-based connections to some other part of the OpenAPI Description (OAD).

These connections are unambiguously resolved in single-document OADs, but the resolution process in multi-document OADs is *implementation-defined*, within the constraints described in this section. In some cases, an unambiguous URI-based alternative is available, and OAD authors are *RECOMMENDED* to always use the alternative:

Source	Target	Alternative
Security Requirement Object `{name}`	Security Scheme Object name under the Components Object	*n/a*

Source	Target	Alternative
Discriminator Object `mapping` *(implicit, or explicit name syntax)*	Schema Object name under the Components Object	`mapping` *(explicit URI syntax)*
Operation Object `tags`	Tag Object `name` (in the OpenAPI Object's `tags` array)	*n/a*
Link Object `operationId`	Path Item Object `operationId`	`operationRef`

A fifth implicit connection involves appending the templated URL paths of the Paths Object to the appropriate Server Object's `url` field. This is unambiguous because only the entry document's Paths Object contributes URLs to the described API.

It is *RECOMMENDED* to consider all Operation Objects from all parsed documents when resolving any Link Object `operationId`. This requires parsing all referenced documents prior to determining an `operationId` to be unresolvable.

The implicit connections in the Security Requirement Object and Discriminator Object rely on the *component name*, which is the name of the property holding the component in the appropriately typed sub-object of the Components Object. For example, the component name of the Schema Object at `#/components/schemas/Foo` is `Foo`. The implicit connection of `tags` in the Operation Object uses the `name` field of Tag Objects, which (like the Components Object) are found under the root OpenAPI Object. This means resolving component names and tag names both depend on starting from the correct OpenAPI Object.

For resolving component and tag name connections from a referenced (non-entry) document, it is *RECOMMENDED* that tools resolve from the entry document, rather than the current document. This allows Security Scheme Objects and Tag Objects to be defined next to the API's deployment information (the top-level array of Server Objects), and treated as an interface for referenced documents to access.

The interface approach can also work for Discriminator Objects and Schema Objects, but it is also possible to keep the Discriminator Object's behavior within a single document using the relative URI-reference syntax of `mapping`.

There are no URI-based alternatives for the Security Requirement Object or for the Operation Object's `tags` field. These limitations are expected to be addressed in a future release.

See Appendix F: Resolving Security Requirements in a Referenced Document for an example of the possible resolutions, including which one is recommended by this section. The behavior for

Discrimator Object non-URI mappings and for the Operation Object's `tags` field operate on the same principles.

Note that no aspect of implicit connection resolution changes how URIs are resolved, or restricts their possible targets.

§ 4.4 Data Types

Data types in the OAS are based on the types defined by the JSON Schema Validation Specification Draft 2020-12: "null", "boolean", "object", "array", "number", "string", or "integer". Models are defined using the Schema Object, which is a superset of the JSON Schema Specification Draft 2020-12.

JSON Schema keywords and `format` values operate on JSON "instances" which may be one of the six JSON data types, "null", "boolean", "object", "array", "number", or "string", with certain keywords and formats only applying to a specific type. For example, the `pattern` keyword and the `date-time` format only apply to strings, and treat any instance of the other five types as *automatically valid*. This means JSON Schema keywords and formats do **NOT** implicitly require the expected type. Use the `type` keyword to explicitly constrain the type.

Note that the `type` keyword allows `"integer"` as a value for convenience, but keyword and format applicability does not recognize integers as being of a distinct JSON type from other numbers because JSON itself does not make that distinction. Since there is no distinct JSON integer type, JSON Schema defines integers mathematically. This means that both `1` and `1.0` are equivalent, and are both considered to be integers.

§ 4.4.1 Data Type Format

As defined by the JSON Schema Validation specification, data types can have an optional modifier keyword: `format`. As described in that specification, `format` is treated as a non-validating annotation by default; the ability to validate `format` varies across implementations.

The OpenAPI Initiative also hosts a Format Registry for formats defined by OAS users and other specifications. Support for any registered format is strictly *OPTIONAL*, and support for one registered format does not imply support for any others.

Types that are not accompanied by a `format` keyword follow the type definition in the JSON Schema. Tools that do not recognize a specific `format` *MAY* default back to the `type` alone, as if the `format` is not specified. For the purpose of JSON Schema validation, each format should

specify the set of JSON data types for which it applies. In this registry, these types are shown in the "JSON Data Type" column.

The formats defined by the OAS are:

format	JSON Data Type	Comments
int32	number	signed 32 bits
int64	number	signed 64 bits (a.k.a long)
float	number	
double	number	
password	string	A hint to obscure the value.

As noted under Data Type, both `type: number` and `type: integer` are considered to be numbers in the data model.

§ 4.4.2 Working with Binary Data

The OAS can describe either *raw* or *encoded* binary data.

- **raw binary** is used where unencoded binary data is allowed, such as when sending a binary payload as the entire HTTP message body, or as part of a `multipart/*` payload that allows binary parts
- **encoded binary** is used where binary data is embedded in a text-only format such as `application/json` or `application/x-www-form-urlencoded` (either as a message body or in the URL query string).

In the following table showing how to use Schema Object keywords for binary data, we use `image/png` as an example binary media type. Any binary media type, including `application/octet-stream`, is sufficient to indicate binary content.

Keyword	Raw	Encoded	Comments
type	*omit*	string	raw binary is outside of type
contentMediaType	image/png	image/png	can sometimes be omitted if redundant (see below)

Keyword	Raw	Encoded	Comments
`contentEncoding`	*omit*	`base64` or `base64url`	other encodings are allowed

Note that the encoding indicated by `contentEncoding`, which inflates the size of data in order to represent it as 7-bit ASCII text, is unrelated to HTTP's `Content-Encoding` header, which indicates whether and how a message body has been compressed and is applied after all content serialization described in this section has occurred. Since HTTP allows unencoded binary message bodies, there is no standardized HTTP header for indicating base64 or similar encoding of an entire message body.

Using a `contentEncoding` of `base64url` ensures that URL encoding (as required in the query string and in message bodies of type `application/x-www-form-urlencoded`) does not need to further encode any part of the already-encoded binary data.

The `contentMediaType` keyword is redundant if the media type is already set:

- as the key for a MediaType Object
- in the `contentType` field of an Encoding Object

If the Schema Object will be processed by a non-OAS-aware JSON Schema implementation, it may be useful to include `contentMediaType` even if it is redundant. However, if `contentMediaType` contradicts a relevant Media Type Object or Encoding Object, then `contentMediaType` *SHALL* be ignored.

The `maxLength` keyword *MAY* be used to set an expected upper bound on the length of a streaming payload. The keyword can be applied to either string data, including encoded binary data, or to unencoded binary data. For unencoded binary, the length is the number of octets.

§ 4.4.2.1 Migrating binary descriptions from OAS 3.0

The following table shows how to migrate from OAS 3.0 binary data descriptions, continuing to use `image/png` as the example binary media type:

OAS < 3.1	OAS 3.1	Comments
`type: string` `format: binary`	`contentMediaType: image/png`	if redundant, can be omitted, often resulting in an empty Schema Object

OAS < 3.1	OAS 3.1	Comments

`type: string`
`format: byte`
	`type: string`
	`contentMediaType: image/png`
	`contentEncoding: base64`
		note that `base64url` can be used to avoid re-encoding the base64 string to be URL-safe

4.5 Rich Text Formatting

Throughout the specification `description` fields are noted as supporting [CommonMark] markdown formatting. Where OpenAPI tooling renders rich text it *MUST* support, at a minimum, markdown syntax as described by [CommonMark-0.27]. Tooling *MAY* choose to ignore some CommonMark or extension features to address security concerns.

While the framing of CommonMark 0.27 as a minimum requirement means that tooling *MAY* choose to implement extensions on top of it, note that any such extensions are by definition implementation-defined and will not be interoperable. OpenAPI Description authors *SHOULD* consider how text using such extensions will be rendered by tools that offer only the minimum support.

4.6 Relative References in API Description URIs

URIs used as references within an OpenAPI Description, or to external documentation or other supplementary information such as a license, are resolved as *identifiers*, and described by this specification as ***URIs***. As noted under Parsing Documents, this specification inherits JSON Schema Specification Draft 2020-12's requirements for loading documents and associating them with their expected URIs, which might not match their current location. This feature is used both for working in development or test environments without having to change the URIs, and for working within restrictive network configurations or security policies.

Note that some URI fields are named `url` for historical reasons, but the descriptive text for those fields uses the correct "URI" terminology.

Unless specified otherwise, all fields that are URIs *MAY* be relative references as defined by [RFC3986] Section 4.2.

Relative references in Schema Objects, including any that appear as `$id` values, use the nearest parent `$id` as a Base URI, as described by JSON Schema Specification Draft 2020-12.

Relative URI references in other Objects, and in Schema Objects where no parent schema contains an `$id`, *MUST* be resolved using the referring document's base URI, which is determined in accordance with [RFC3986] Section 5.1.2 – 5.1.4. In practice, this is usually the retrieval URI of the document, which *MAY* be determined based on either its current actual location or a user-supplied expected location.

If a URI contains a fragment identifier, then the fragment should be resolved per the fragment resolution mechanism of the referenced document. If the representation of the referenced document is JSON or YAML, then the fragment identifier *SHOULD* be interpreted as a JSON-Pointer as per [RFC6901].

Relative references in CommonMark hyperlinks are resolved in their rendered context, which might differ from the context of the API description.

4.7 Relative References in API URLs

API endpoints are by definition accessed as locations, and are described by this specification as ***URLs***.

Unless specified otherwise, all fields that are URLs *MAY* be relative references as defined by [RFC3986] Section 4.2. Unless specified otherwise, relative references are resolved using the URLs defined in the Server Object as a Base URL. Note that these themselves *MAY* be relative to the referring document.

4.8 Schema

This section describes the structure of the OpenAPI Description format. This text is the only normative description of the format. A JSON Schema is hosted on spec.openapis.org for informational purposes. If the JSON Schema differs from this section, then this section *MUST* be considered authoritative.

In the following description, if a field is not explicitly ***REQUIRED*** or described with a *MUST* or *SHALL*, it can be considered *OPTIONAL*.

4.8.1 OpenAPI Object

This is the root object of the OpenAPI Description.

§ 4.8.1.1 Fixed Fields

Field Name	Type	Description
openapi	`string`	***REQUIRED***. This string *MUST* be the version number of the OpenAPI Specification that the OpenAPI Document uses. The `openapi` field *SHOULD* be used by tooling to interpret the OpenAPI Document. This is *not* related to the API `info.version` string.
info	Info Object	***REQUIRED***. Provides metadata about the API. The metadata *MAY* be used by tooling as required.
jsonSchemaDialect	`string`	The default value for the `$schema` keyword within Schema Objects contained within this OAS document. This *MUST* be in the form of a URI.
servers	[Server Object]	An array of Server Objects, which provide connectivity information to a target server. If the `servers` field is not provided, or is an empty array, the default value would be a Server Object with a url value of `/`.
paths	Paths Object	The available paths and operations for the API.
webhooks	Map[`string`, Path Item Object]	The incoming webhooks that *MAY* be received as part of this API and that the API consumer *MAY* choose to implement. Closely related to the `callbacks` feature, this section describes requests initiated other than by an API call, for example by an out of band registration. The key name is a unique string to refer to each webhook, while the (optionally referenced) Path Item Object describes a request that may be initiated by the API provider and the expected responses. An example is available.
components	Components Object	An element to hold various Objects for the OpenAPI Description.

Field Name	Type	Description
security	[Security Requirement Object]	A declaration of which security mechanisms can be used across the API. The list of values includes alternative Security Requirement Objects that can be used. Only one of the Security Requirement Objects need to be satisfied to authorize a request. Individual operations can override this definition. The list can be incomplete, up to being empty or absent. To make security explicitly optional, an empty security requirement (`{}`) can be included in the array.
tags	[Tag Object]	A list of tags used by the OpenAPI Description with additional metadata. The order of the tags can be used to reflect on their order by the parsing tools. Not all tags that are used by the Operation Object must be declared. The tags that are not declared *MAY* be organized randomly or based on the tools' logic. Each tag name in the list *MUST* be unique.
externalDocs	External Documentation Object	Additional external documentation.

This object *MAY* be extended with Specification Extensions.

§ 4.8.2 Info Object

The object provides metadata about the API. The metadata *MAY* be used by the clients if needed, and *MAY* be presented in editing or documentation generation tools for convenience.

§ 4.8.2.1 Fixed Fields

Field Name	Type	Description
title	`string`	***REQUIRED***. The title of the API.

Field Name	Type	Description
summary	`string`	A short summary of the API.
description	`string`	A description of the API. [CommonMark] syntax *MAY* be used for rich text representation.
termsOfService	`string`	A URI for the Terms of Service for the API. This *MUST* be in the form of a URI.
contact	Contact Object	The contact information for the exposed API.
license	License Object	The license information for the exposed API.
version	`string`	***REQUIRED***. The version of the OpenAPI Document (which is distinct from the OpenAPI Specification version or the version of the API being described or the version of the OpenAPI Description).

This object *MAY* be extended with Specification Extensions.

§ 4.8.2.2 Info Object Example

```
{
  "title": "Example Pet Store App",
  "summary": "A pet store manager.",
  "description": "This is an example server for a pet store.",
  "termsOfService": "https://example.com/terms/",
  "contact": {
    "name": "API Support",
    "url": "https://www.example.com/support",
    "email": "support@example.com"
  },
  "license": {
    "name": "Apache 2.0",
    "url": "https://www.apache.org/licenses/LICENSE-2.0.html"
```

```
  },
  "version": "1.0.1"
}
```

```
title: Example Pet Store App
summary: A pet store manager.
description: This is an example server for a pet store.
termsOfService: https://example.com/terms/
contact:
  name: API Support
  url: https://www.example.com/support
  email: support@example.com
license:
  name: Apache 2.0
  url: https://www.apache.org/licenses/LICENSE-2.0.html
version: 1.0.1
```

§ 4.8.3 Contact Object

Contact information for the exposed API.

§ 4.8.3.1 Fixed Fields

Field Name	Type	Description
name	`string`	The identifying name of the contact person/organization.
url	`string`	The URI for the contact information. This *MUST* be in the form of a URI.
email	`string`	The email address of the contact person/organization. This *MUST* be in the form of an email address.

This object *MAY* be extended with Specification Extensions.

§ 4.8.3.2 Contact Object Example

```
{
  "name": "API Support",
  "url": "https://www.example.com/support",
  "email": "support@example.com"
}
```

```
name: API Support
url: https://www.example.com/support
email: support@example.com
```

§ 4.8.4 License Object

License information for the exposed API.

§ 4.8.4.1 Fixed Fields

Field Name	Type	Description
name	`string`	***REQUIRED***. The license name used for the API.
identifier	`string`	An [SPDX-Licenses] expression for the API. The `identifier` field is mutually exclusive of the `url` field.
url	`string`	A URI for the license used for the API. This *MUST* be in the form of a URI. The `url` field is mutually exclusive of the `identifier` field.

This object *MAY* be extended with Specification Extensions.

§ 4.8.4.2 License Object Example

```
{
  "name": "Apache 2.0",
  "identifier": "Apache-2.0"
}
```

```
name: Apache 2.0
identifier: Apache-2.0
```

§ 4.8.5 Server Object

An object representing a Server.

§ 4.8.5.1 Fixed Fields

Field Name	Type	Description
url	`string`	***REQUIRED***. A URL to the target host. This URL supports Server Variables and *MAY* be relative, to indicate that the host location is relative to the location where the document containing the Server Object is being served. Variable substitutions will be made when a variable is named in {braces}.
description	`string`	An optional string describing the host designated by the URL. [CommonMark] syntax *MAY* be used for rich text representation.
variables	Map[`string`, Server Variable Object]	A map between a variable name and its value. The value is used for substitution in the server's URL template.

This object *MAY* be extended with Specification Extensions.

§ 4.8.5.2 Server Object Example

A single server would be described as:

```
{
  "url": "https://development.gigantic-server.com/v1",
  "description": "Development server"
}
```

```
url: https://development.gigantic-server.com/v1
description: Development server
```

The following shows how multiple servers can be described, for example, at the OpenAPI Object's servers:

```
{
  "servers": [
    {
      "url": "https://development.gigantic-server.com/v1",
      "description": "Development server"
    },
    {
      "url": "https://staging.gigantic-server.com/v1",
      "description": "Staging server"
    },
    {
      "url": "https://api.gigantic-server.com/v1",
      "description": "Production server"
    }
  ]
}
```

```
servers:
  - url: https://development.gigantic-server.com/v1
    description: Development server
  - url: https://staging.gigantic-server.com/v1
    description: Staging server
  - url: https://api.gigantic-server.com/v1
    description: Production server
```

The following shows how variables can be used for a server configuration:

```
{
  "servers": [
    {
      "url": "https://{username}.gigantic-server.com:{port}/{basePath}",
      "description": "The production API server",
```

```
      "variables": {
        "username": {
          "default": "demo",
          "description": "A user-specific subdomain. Use `demo` for a free
        },
        "port": {
          "enum": ["8443", "443"],
          "default": "8443"
        },
        "basePath": {
          "default": "v2"
        }
      }
    }
  ]
}
```

```
servers:
  - url: https://{username}.gigantic-server.com:{port}/{basePath}
    description: The production API server
    variables:
      username:
        # note! no enum here means it is an open value
        default: demo
        description: A user-specific subdomain. Use `demo` for a free sanc
      port:
        enum:
          - '8443'
          - '443'
        default: '8443'
      basePath:
        # open meaning there is the opportunity to use special base paths
        default: v2
```

§ 4.8.6 Server Variable Object

An object representing a Server Variable for server URL template substitution.

§ 4.8.6.1 Fixed Fields

Field Name	Type	Description
enum	[`string`]	An enumeration of string values to be used if the substitution options are from a limited set. The array *MUST NOT* be empty.
default	`string`	***REQUIRED***. The default value to use for substitution, which *SHALL* be sent if an alternate value is *not* supplied. If the `enum` is defined, the value *MUST* exist in the enum's values. Note that this behavior is different from the Schema Object's `default` keyword, which documents the receiver's behavior rather than inserting the value into the data.
description	`string`	An optional description for the server variable. [CommonMark] syntax *MAY* be used for rich text representation.

This object *MAY* be extended with Specification Extensions.

§ 4.8.7 Components Object

Holds a set of reusable objects for different aspects of the OAS. All objects defined within the Components Object will have no effect on the API unless they are explicitly referenced from outside the Components Object.

§ 4.8.7.1 Fixed Fields

Field Name	Type	Description
schemas	Map[`string`, Schema Object]	An object to hold reusable Schema Objects.
responses	Map[`string`, Response Object | Reference Object]	An object to hold reusable Response Objects.
parameters	Map[`string`, Parameter Object | Reference Object]	An object to hold reusable Parameter Objects.

Field Name	Type	Description
examples	Map[`string`, Example Object | Reference Object]	An object to hold reusable Example Objects.
requestBodies	Map[`string`, Request Body Object | Reference Object]	An object to hold reusable Request Body Objects.
headers	Map[`string`, Header Object | Reference Object]	An object to hold reusable Header Objects.
securitySchemes	Map[`string`, Security Scheme Object | Reference Object]	An object to hold reusable Security Scheme Objects.
links	Map[`string`, Link Object | Reference Object]	An object to hold reusable Link Objects.
callbacks	Map[`string`, Callback Object | Reference Object]	An object to hold reusable Callback Objects.
pathItems	Map[`string`, Path Item Object]	An object to hold reusable Path Item Objects.

This object *MAY* be extended with Specification Extensions.

All the fixed fields declared above are objects that *MUST* use keys that match the regular expression: `^[a-zA-Z0-9\.\-_]+$`.

Field Name Examples:

```
User
User_1
User_Name
user-name
my.org.User
```

§ 4.8.7.2 Components Object Example

```
"components": {
  "schemas": {
    "GeneralError": {
      "type": "object",
      "properties": {
        "code": {
```

```
            "type": "integer",
            "format": "int32"
          },
          "message": {
            "type": "string"
          }
        }
      },
      "Category": {
        "type": "object",
        "properties": {
          "id": {
            "type": "integer",
            "format": "int64"
          },
          "name": {
            "type": "string"
          }
        }
      },
      "Tag": {
        "type": "object",
        "properties": {
          "id": {
            "type": "integer",
            "format": "int64"
          },
          "name": {
            "type": "string"
          }
        }
      }
    },
    "parameters": {
      "skipParam": {
        "name": "skip",
        "in": "query",
        "description": "number of items to skip",
        "required": true,
        "schema": {
          "type": "integer",
          "format": "int32"
        }
      },
      "limitParam": {
        "name": "limit",
```

```
      "in": "query",
      "description": "max records to return",
      "required": true,
      "schema" : {
        "type": "integer",
        "format": "int32"
      }
    }
  },
  "responses": {
    "NotFound": {
      "description": "Entity not found."
    },
    "IllegalInput": {
      "description": "Illegal input for operation."
    },
    "GeneralError": {
      "description": "General Error",
      "content": {
        "application/json": {
          "schema": {
            "$ref": "#/components/schemas/GeneralError"
          }
        }
      }
    }
  },
  "securitySchemes": {
    "api_key": {
      "type": "apiKey",
      "name": "api-key",
      "in": "header"
    },
    "petstore_auth": {
      "type": "oauth2",
      "flows": {
        "implicit": {
          "authorizationUrl": "https://example.org/api/oauth/dialog",
          "scopes": {
            "write:pets": "modify pets in your account",
            "read:pets": "read your pets"
          }
        }
      }
```

```
      }
    }
  }

  components:
    schemas:
      GeneralError:
        type: object
        properties:
          code:
            type: integer
            format: int32
          message:
            type: string
      Category:
        type: object
        properties:
          id:
            type: integer
            format: int64
          name:
            type: string
      Tag:
        type: object
        properties:
          id:
            type: integer
            format: int64
          name:
            type: string
    parameters:
      skipParam:
        name: skip
        in: query
        description: number of items to skip
        required: true
        schema:
          type: integer
          format: int32
      limitParam:
        name: limit
        in: query
        description: max records to return
        required: true
        schema:
          type: integer
          format: int32
```

```yaml
responses:
  NotFound:
    description: Entity not found.
  IllegalInput:
    description: Illegal input for operation.
  GeneralError:
    description: General Error
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/GeneralError'
securitySchemes:
  api_key:
    type: apiKey
    name: api-key
    in: header
  petstore_auth:
    type: oauth2
    flows:
      implicit:
        authorizationUrl: https://example.org/api/oauth/dialog
        scopes:
          write:pets: modify pets in your account
          read:pets: read your pets
```

§ 4.8.8 Paths Object

Holds the relative paths to the individual endpoints and their operations. The path is appended to the URL from the Server Object in order to construct the full URL. The Paths Object *MAY* be empty, due to Access Control List (ACL) constraints.

§ 4.8.8.1 Patterned Fields

Field Pattern	Type	Description
/{path}	Path Item Object	A relative path to an individual endpoint. The field name *MUST* begin with a forward slash (`/`). The path is **appended** (no relative URL resolution) to the expanded URL from the Server Object's `url` field in order to construct the full URL. Path templating is allowed. When matching URLs, concrete (non-templated) paths would be

Field Pattern	Type	Description
		matched before their templated counterparts. Templated paths with the same hierarchy but different templated names *MUST NOT* exist as they are identical. In case of ambiguous matching, it's up to the tooling to decide which one to use.

This object *MAY* be extended with Specification Extensions.

§ 4.8.8.2 Path Templating Matching

Assuming the following paths, the concrete definition, `/pets/mine`, will be matched first if used:

```
/pets/{petId}
/pets/mine
```

The following paths are considered identical and invalid:

```
/pets/{petId}
/pets/{name}
```

The following may lead to ambiguous resolution:

```
/{entity}/me
/books/{id}
```

§ 4.8.8.3 Paths Object Example

```
{
  "/pets": {
    "get": {
      "description": "Returns all pets from the system that the user has a
      "responses": {
        "200": {
          "description": "A list of pets.",
          "content": {
            "application/json": {
              "schema": {
                "type": "array",
                "items": {
                  "$ref": "#/components/schemas/pet"
```

```
                    }
                  }
                }
              }
            }
          }
        }
      }
    }

    /pets:
      get:
        description: Returns all pets from the system that the user has access
        responses:
          '200':
            description: A list of pets.
            content:
              application/json:
                schema:
                  type: array
                  items:
                    $ref: '#/components/schemas/pet'
```

§ 4.8.9 Path Item Object

Describes the operations available on a single path. A Path Item *MAY* be empty, due to ACL constraints. The path itself is still exposed to the documentation viewer but they will not know which operations and parameters are available.

§ 4.8.9.1 Fixed Fields

Field Name	Type	Description
$ref	`string`	Allows for a referenced definition of this path item. The value *MUST* be in the form of a URI, and the referenced structure *MUST* be in the form of a Path Item Object. In case a Path Item Object field appears both in the defined object and the referenced object, the behavior is undefined. See the rules for resolving Relative References.

Field Name	Type	Description
		Note: *The behavior of* `$ref` *with adjacent properties is likely to change in future versions of this specification to bring it into closer alignment with the behavior of the Reference Object.*
summary	`string`	An optional string summary, intended to apply to all operations in this path.
description	`string`	An optional string description, intended to apply to all operations in this path. [CommonMark] syntax *MAY* be used for rich text representation.
get	Operation Object	A definition of a GET operation on this path.
put	Operation Object	A definition of a PUT operation on this path.
post	Operation Object	A definition of a POST operation on this path.
delete	Operation Object	A definition of a DELETE operation on this path.
options	Operation Object	A definition of a OPTIONS operation on this path.
head	Operation Object	A definition of a HEAD operation on this path.
patch	Operation Object	A definition of a PATCH operation on this path.
trace	Operation Object	A definition of a TRACE operation on this path.
servers	[Server Object]	An alternative `servers` array to service all operations in this path. If a `servers` array is specified at the OpenAPI Object level, it will be overridden by this value.
parameters	[Parameter Object |	A list of parameters that are applicable for all the operations described under this path. These parameters can be overridden at the operation level, but cannot be removed

Field Name	Type	Description
	Reference Object]	there. The list *MUST NOT* include duplicated parameters. A unique parameter is defined by a combination of a name and location. The list can use the Reference Object to link to parameters that are defined in the OpenAPI Object's `components.parameters`.

This object *MAY* be extended with Specification Extensions.

§ 4.8.9.2 Path Item Object Example

```
{
  "get": {
    "description": "Returns pets based on ID",
    "summary": "Find pets by ID",
    "operationId": "getPetsById",
    "responses": {
      "200": {
        "description": "pet response",
        "content": {
          "*/*": {
            "schema": {
              "type": "array",
              "items": {
                "$ref": "#/components/schemas/Pet"
              }
            }
          }
        }
      },
      "default": {
        "description": "error payload",
        "content": {
          "text/html": {
            "schema": {
              "$ref": "#/components/schemas/ErrorModel"
            }
          }
        }
      }
    }
```

```
    },
    "parameters": [
      {
        "name": "id",
        "in": "path",
        "description": "ID of pet to use",
        "required": true,
        "schema": {
          "type": "array",
          "items": {
            "type": "string"
          }
        },
        "style": "simple"
      }
    ]
  }
```

```
  get:
    description: Returns pets based on ID
    summary: Find pets by ID
    operationId: getPetsById
    responses:
      '200':
        description: pet response
        content:
          '*/*':
            schema:
              type: array
              items:
                $ref: '#/components/schemas/Pet'
      default:
        description: error payload
        content:
          text/html:
            schema:
              $ref: '#/components/schemas/ErrorModel'
  parameters:
    - name: id
      in: path
      description: ID of pet to use
      required: true
      schema:
        type: array
        items:
          type: string
      style: simple
```

§ 4.8.10 Operation Object

Describes a single API operation on a path.

§ 4.8.10.1 Fixed Fields

Field Name	Type	Description
tags	[`string`]	A list of tags for API documentation control. Tags can be used for logical grouping of operations by resources or any other qualifier.
summary	`string`	A short summary of what the operation does.
description	`string`	A verbose explanation of the operation behavior. [CommonMark] syntax *MAY* be used for rich text representation.
externalDocs	External Documentation Object	Additional external documentation for this operation.
operationId	`string`	Unique string used to identify the operation. The id *MUST* be unique among all operations described in the API. The operationId value is **case-sensitive**. Tools and libraries *MAY* use the operationId to uniquely identify an operation, therefore, it is *RECOMMENDED* to follow common programming naming conventions.
parameters	[Parameter Object | Reference Object]	A list of parameters that are applicable for this operation. If a parameter is already defined at the Path Item, the new definition will override it but can never remove it. The list *MUST NOT* include duplicated parameters. A unique parameter is defined by a combination of a name and location. The list can use the Reference Object to link to parameters that are defined in the OpenAPI Object's `components.parameters`.

Field Name	Type	Description
requestBody	Request Body Object | Reference Object	The request body applicable for this operation. The `requestBody` is fully supported in HTTP methods where the HTTP 1.1 specification [RFC7231] Section 4.3.1 has explicitly defined semantics for request bodies. In other cases where the HTTP spec is vague (such as GET, HEAD and DELETE), `requestBody` is permitted but does not have well-defined semantics and *SHOULD* be avoided if possible.
responses	Responses Object	The list of possible responses as they are returned from executing this operation.
callbacks	Map[`string`, Callback Object | Reference Object]	A map of possible out-of band callbacks related to the parent operation. The key is a unique identifier for the Callback Object. Each value in the map is a Callback Object that describes a request that may be initiated by the API provider and the expected responses.
deprecated	`boolean`	Declares this operation to be deprecated. Consumers *SHOULD* refrain from usage of the declared operation. Default value is `false`.
security	[Security Requirement Object]	A declaration of which security mechanisms can be used for this operation. The list of values includes alternative Security Requirement Objects that can be used. Only one of the Security Requirement Objects need to be satisfied to authorize a request. To make security optional, an empty security requirement (`{}`) can be included in the array. This definition overrides any declared top-level `security`. To remove a top-level security declaration, an empty array can be used.
servers	[Server Object]	An alternative `servers` array to service this operation. If a `servers` array is specified at the Path Item Object or OpenAPI Object level, it will be overridden by this value.

This object *MAY* be extended with Specification Extensions.

§ 4.8.10.2 Operation Object Example

```
{
  "tags": ["pet"],
  "summary": "Updates a pet in the store with form data",
  "operationId": "updatePetWithForm",
  "parameters": [
    {
      "name": "petId",
      "in": "path",
      "description": "ID of pet that needs to be updated",
      "required": true,
      "schema": {
        "type": "string"
      }
    }
  ],
  "requestBody": {
    "content": {
      "application/x-www-form-urlencoded": {
        "schema": {
          "type": "object",
          "properties": {
            "name": {
              "description": "Updated name of the pet",
              "type": "string"
            },
            "status": {
              "description": "Updated status of the pet",
              "type": "string"
            }
          },
          "required": ["status"]
        }
      }
    }
  },
  "responses": {
    "200": {
      "description": "Pet updated.",
      "content": {
        "application/json": {},
        "application/xml": {}
```

```
        }
      },
      "405": {
        "description": "Method Not Allowed",
        "content": {
          "application/json": {},
          "application/xml": {}
        }
      }
    },
    "security": [
      {
        "petstore_auth": ["write:pets", "read:pets"]
      }
    ]
  }
```

```yaml
  tags:
    - pet
  summary: Updates a pet in the store with form data
  operationId: updatePetWithForm
  parameters:
    - name: petId
      in: path
      description: ID of pet that needs to be updated
      required: true
      schema:
        type: string
  requestBody:
    content:
      application/x-www-form-urlencoded:
        schema:
          type: object
          properties:
            name:
              description: Updated name of the pet
              type: string
            status:
              description: Updated status of the pet
              type: string
          required:
            - status
  responses:
    '200':
      description: Pet updated.
      content:
        application/json: {}
```

```
        application/xml: {}
    '405':
      description: Method Not Allowed
      content:
        application/json: {}
        application/xml: {}
  security:
    - petstore_auth:
        - write:pets
        - read:pets
```

§ 4.8.11 External Documentation Object

Allows referencing an external resource for extended documentation.

§ 4.8.11.1 Fixed Fields

Field Name	Type	Description
description	`string`	A description of the target documentation. [CommonMark] syntax *MAY* be used for rich text representation.
url	`string`	***REQUIRED***. The URI for the target documentation. This *MUST* be in the form of a URI.

This object *MAY* be extended with Specification Extensions.

§ 4.8.11.2 External Documentation Object Example

```
{
  "description": "Find more info here",
  "url": "https://example.com"
}
```

```
description: Find more info here
url: https://example.com
```

§ 4.8.12 Parameter Object

Describes a single operation parameter.

A unique parameter is defined by a combination of a name and location.

See Appendix E for a detailed examination of percent-encoding concerns, including interactions with the `application/x-www-form-urlencoded` query string format.

§ 4.8.12.1 Parameter Locations

There are four possible parameter locations specified by the `in` field:

- path - Used together with Path Templating, where the parameter value is actually part of the operation's URL. This does not include the host or base path of the API. For example, in `/items/{itemId}`, the path parameter is `itemId`.
- query - Parameters that are appended to the URL. For example, in `/items?id=###`, the query parameter is `id`.
- header - Custom headers that are expected as part of the request. Note that [RFC7230] Section 3.2 states header names are case insensitive.
- cookie - Used to pass a specific cookie value to the API.

§ 4.8.12.2 Fixed Fields

The rules for serialization of the parameter are specified in one of two ways. Parameter Objects *MUST* include either a `content` field or a `schema` field, but not both. See Appendix B for a discussion of converting values of various types to string representations.

§ 4.8.12.2.1 Common Fixed Fields

These fields *MAY* be used with either `content` or `schema`.

Field Name	Type	Description
name	`string`	***REQUIRED***. The name of the parameter. Parameter names are *case sensitive*. • If `in` is `"path"`, the `name` field *MUST* correspond to a template expression occurring within the path field in the Paths Object. See Path Templating for further information. • If `in` is `"header"` and the `name` field is `"Accept"`, `"Content-Type"` or `"Authorization"`, the parameter definition *SHALL* be ignored. • For all other cases, the `name` corresponds to the parameter name used by the `in` field.
in	`string`	***REQUIRED***. The location of the parameter. Possible values are `"query"`, `"header"`, `"path"` or `"cookie"`.
description	`string`	A brief description of the parameter. This could contain examples of use. [CommonMark] syntax *MAY* be used for rich text representation.
required	`boolean`	Determines whether this parameter is mandatory. If the parameter location is `"path"`, this field is ***REQUIRED*** and its value *MUST* be `true`. Otherwise, the field *MAY* be included and its default value is `false`.
deprecated	`boolean`	Specifies that a parameter is deprecated and *SHOULD* be transitioned out of usage. Default value is `false`.
allowEmptyValue	`boolean`	If `true`, clients *MAY* pass a zero-length string value in place of parameters that would otherwise be omitted entirely, which the server *SHOULD* interpret as the parameter being unused. Default value is `false`. If `style` is used, and if behavior is *n/a* (cannot be serialized), the value of `allowEmptyValue` *SHALL* be ignored. Interactions between this field and the parameter's Schema Object are implementation-defined. This field is valid only for `query` parameters. Use of this field is *NOT RECOMMENDED*, and it is likely to be removed in a later revision.

This object *MAY* be extended with Specification Extensions.

Note that while `"Cookie"` as a `name` is not forbidden if `in` is `"header"`, the effect of defining a cookie parameter that way is undefined; use `in: "cookie"` instead.

§ 4.8.12.2.2 Fixed Fields for use with `schema`

For simpler scenarios, a `schema` and `style` can describe the structure and syntax of the parameter. When `example` or `examples` are provided in conjunction with the `schema` field, the example *SHOULD* match the specified schema and follow the prescribed serialization strategy for the parameter. The `example` and `examples` fields are mutually exclusive, and if either is present it *SHALL override* any `example` in the schema.

Serializing with `schema` is *NOT RECOMMENDED* for `in: "cookie"` parameters, `in: "header"` parameters that use HTTP header parameters (name=value pairs following a `;`) in their values, or `in: "header"` parameters where values might have non-URL-safe characters; see Appendix D for details.

Field Name	Type	Description
style	`string`	Describes how the parameter value will be serialized depending on the type of the parameter value. Default values (based on value of `in`): for `"query"` - `"form"`; for `"path"` - `"simple"`; for `"header"` - `"simple"`; for `"cookie"` - `"form"`.
explode	`boolean`	When this is true, parameter values of type `array` or `object` generate separate parameters for each value of the array or key-value pair of the map. For other types of parameters this field has no effect. When `style` is `"form"`, the default value is `true`. For all other styles, the default value is `false`. Note that despite `false` being the default for `deepObject`, the combination of `false` with `deepObject` is undefined.
allowReserved	`boolean`	When this is true, parameter values are serialized using reserved expansion, as defined by [RFC6570] Section 3.2.3, which allows RFC3986's reserved character set, as well as percent-encoded triples, to pass through unchanged, while still percent-encoding all other disallowed characters (including `%` outside of percent-

Field Name	Type	Description
		encoded triples). Applications are still responsible for percent-encoding reserved characters that are not allowed in the query string (`[`, `]`, `#`), or have a special meaning in `application/x-www-form-urlencoded` (`-`, `&`, `+`); see Appendices C and E for details. This field only applies to parameters with an `in` value of `query`. The default value is `false`.
schema	Schema Object	The schema defining the type used for the parameter.
example	Any	Example of the parameter's potential value; see Working With Examples.
examples	Map[`string`, Example Object | Reference Object]	Examples of the parameter's potential value; see Working With Examples.

See also Appendix C: Using RFC6570-Based Serialization for additional guidance.

§ 4.8.12.2.3 Fixed Fields for use with `content`

For more complex scenarios, the `content` field can define the media type and schema of the parameter, as well as give examples of its use. Using `content` with a `text/plain` media type is *RECOMMENDED* for `in: "header"` and `in: "cookie"` parameters where the `schema` strategy is not appropriate.

Field Name	Type	Description
content	Map[`string`, Media Type Object]	A map containing the representations for the parameter. The key is the media type and the value describes it. The map *MUST* only contain one entry.

§ 4.8.12.3 Style Values

In order to support common ways of serializing simple parameters, a set of `style` values are defined.

style	type	in	Comments
matrix	`primitive`, `array`, `object`	`path`	Path-style parameters defined by [RFC6570] Section 3.2.7
label	`primitive`, `array`, `object`	`path`	Label style parameters defined by [RFC6570] Section 3.2.5
simple	`primitive`, `array`, `object`	`path`, `header`	Simple style parameters defined by [RFC6570] Section 3.2.2. This option replaces `collectionFormat` with a `csv` value from OpenAPI 2.0.
form	`primitive`, `array`, `object`	`query`, `cookie`	Form style parameters defined by [RFC6570] Section 3.2.8. This option replaces `collectionFormat` with a `csv` (when `explode` is false) or `multi` (when `explode` is true) value from OpenAPI 2.0.
spaceDelimited	`array`, `object`	`query`	Space separated array values or object properties and values. This option replaces `collectionFormat` equal to `ssv` from OpenAPI 2.0.
pipeDelimited	`array`, `object`	`query`	Pipe separated array values or object properties and values. This option replaces `collectionFormat` equal to `pipes` from OpenAPI 2.0.
deepObject	`object`	`query`	Allows objects with scalar properties to be represented using form parameters. The representation of array or object properties is not defined.

See Appendix E for a discussion of percent-encoding, including when delimiters need to be percent-encoded and options for handling collisions with percent-encoded data.

§ 4.8.12.4 Style Examples

Assume a parameter named `color` has one of the following values:

```
string -> "blue"
array -> ["blue", "black", "brown"]
object -> { "R": 100, "G": 200, "B": 150 }
```

The following table shows examples, as would be shown with the `example` or `examples` keywords, of the different serializations for each value.

- The value *empty* denotes the empty string, and is unrelated to the `allowEmptyValue` field
- The behavior of combinations marked *n/a* is undefined
- The `undefined` column replaces the `empty` column in previous versions of this specification in order to better align with [RFC6570] Section 2.3 terminology, which describes certain values including but not limited to `null` as "undefined" values with special handling; notably, the empty string is *not* undefined
- For `form` and the non-RFC6570 query string styles `spaceDelimited`, `pipeDelimited`, and `deepObject`, each example is shown prefixed with `?` as if it were the only query parameter; see Appendix C for more information on constructing query strings from multiple parameters, and Appendix D for warnings regarding `form` and cookie parameters
- Note that the `?` prefix is not appropriate for serializing `application/x-www-form-urlencoded` HTTP message bodies, and *MUST* be stripped or (if constructing the string manually) not added when used in that context; see the Encoding Object for more information
- The examples are percent-encoded as required by RFC6570 and RFC3986; see Appendix E for a thorough discussion of percent-encoding concerns, including why unencoded `|` (`%7C`), `[` (`%5B`), and `]` (`%5D`) seem to work in some environments despite not being compliant.

style	explode	undefined	string	array
matrix	false	;color	;color=blue	;color=blue,black,brown
matrix	true	;color	;color=blue	;color=blue;color=black;colo
label	false	.	.blue	.blue,black,brown
label	true	.	.blue	.blue.black.brown

style	explode	undefined	string	array
simple	false	*empty*	blue	blue,black,brown
simple	true	*empty*	blue	blue,black,brown
form	false	?color=	?color=blue	?color=blue,black,brown
form	true	?color=	?color=blue	?color=blue&color=black&c
spaceDelimited	false	*n/a*	*n/a*	?color=blue%20black%20br
spaceDelimited	true	*n/a*	*n/a*	*n/a*
pipeDelimited	false	*n/a*	*n/a*	?color=blue%7Cblack%7Cbr
pipeDelimited	true	*n/a*	*n/a*	*n/a*
deepObject	false	*n/a*	*n/a*	*n/a*
deepObject	true	*n/a*	*n/a*	*n/a*

§ 4.8.12.5 Parameter Object Examples

A header parameter with an array of 64-bit integer numbers:

```
{
  "name": "token",
  "in": "header",
  "description": "token to be passed as a header",
  "required": true,
  "schema": {
    "type": "array",
    "items": {
      "type": "integer",
      "format": "int64"
    }
  },
  "style": "simple"
}
```

```
name: token
in: header
description: token to be passed as a header
required: true
```

```yaml
schema:
  type: array
  items:
    type: integer
    format: int64
style: simple
```

A path parameter of a string value:

```json
{
  "name": "username",
  "in": "path",
  "description": "username to fetch",
  "required": true,
  "schema": {
    "type": "string"
  }
}
```

```yaml
name: username
in: path
description: username to fetch
required: true
schema:
  type: string
```

An optional query parameter of a string value, allowing multiple values by repeating the query parameter:

```json
{
  "name": "id",
  "in": "query",
  "description": "ID of the object to fetch",
  "required": false,
  "schema": {
    "type": "array",
    "items": {
      "type": "string"
    }
  },
  "style": "form",
  "explode": true
}
```

```yaml
name: id
in: query
description: ID of the object to fetch
required: false
```

```yaml
schema:
  type: array
  items:
    type: string
style: form
explode: true
```

A free-form query parameter, allowing undefined parameters of a specific type:

```json
{
  "in": "query",
  "name": "freeForm",
  "schema": {
    "type": "object",
    "additionalProperties": {
      "type": "integer"
    }
  },
  "style": "form"
}
```

```yaml
in: query
name: freeForm
schema:
  type: object
  additionalProperties:
    type: integer
style: form
```

A complex parameter using `content` to define serialization:

```json
{
  "in": "query",
  "name": "coordinates",
  "content": {
    "application/json": {
      "schema": {
        "type": "object",
        "required": ["lat", "long"],
        "properties": {
          "lat": {
            "type": "number"
          },
          "long": {
            "type": "number"
          }
        }
```

```
      }
    }
  }
}

in: query
name: coordinates
content:
  application/json:
    schema:
      type: object
      required:
        - lat
        - long
      properties:
        lat:
          type: number
        long:
          type: number
```

§ 4.8.13 Request Body Object

Describes a single request body.

§ 4.8.13.1 Fixed Fields

Field Name	Type	Description
description	`string`	A brief description of the request body. This could contain examples of use. [CommonMark] syntax *MAY* be used for rich text representation.
content	Map[`string`, Media Type Object]	***REQUIRED***. The content of the request body. The key is a media type or media type range, see [RFC7231] Appendix D, and the value describes it. For requests that match multiple keys, only the most specific key is applicable. e.g. `"text/plain"` overrides `"text/*"`
required	`boolean`	Determines if the request body is required in the request. Defaults to `false`.

This object *MAY* be extended with Specification Extensions.

§ 4.8.13.2 Request Body Examples

A request body with a referenced schema definition.

```
{
  "description": "user to add to the system",
  "content": {
    "application/json": {
      "schema": {
        "$ref": "#/components/schemas/User"
      },
      "examples": {
        "user": {
          "summary": "User Example",
          "externalValue": "https://foo.bar/examples/user-example.json"
        }
      }
    },
    "application/xml": {
      "schema": {
        "$ref": "#/components/schemas/User"
      },
      "examples": {
        "user": {
          "summary": "User example in XML",
          "externalValue": "https://foo.bar/examples/user-example.xml"
        }
      }
    },
    "text/plain": {
      "examples": {
        "user": {
          "summary": "User example in Plain text",
          "externalValue": "https://foo.bar/examples/user-example.txt"
        }
      }
    },
    "*/*": {
      "examples": {
        "user": {
          "summary": "User example in other format",
          "externalValue": "https://foo.bar/examples/user-example.whatever
```

```
        }
      }
    }
  }
}

description: user to add to the system
content:
  application/json:
    schema:
      $ref: '#/components/schemas/User'
    examples:
      user:
        summary: User example
        externalValue: https://foo.bar/examples/user-example.json
  application/xml:
    schema:
      $ref: '#/components/schemas/User'
    examples:
      user:
        summary: User example in XML
        externalValue: https://foo.bar/examples/user-example.xml
  text/plain:
    examples:
      user:
        summary: User example in plain text
        externalValue: https://foo.bar/examples/user-example.txt
  '*/*':
    examples:
      user:
        summary: User example in other format
        externalValue: https://foo.bar/examples/user-example.whatever
```

§ 4.8.14 Media Type Object

Each Media Type Object provides schema and examples for the media type identified by its key.

When `example` or `examples` are provided, the example *SHOULD* match the specified schema and be in the correct format as specified by the media type and its encoding. The `example` and `examples` fields are mutually exclusive, and if either is present it *SHALL override* any `example` in the schema. See Working With Examples for further guidance regarding the different ways of specifying examples, including non-JSON/YAML values.

§ 4.8.14.1 Fixed Fields

Field Name	Type	Description
schema	Schema Object	The schema defining the content of the request, response, parameter, or header.
example	Any	Example of the media type; see Working With Examples.
examples	Map[`string`, Example Object | Reference Object]	Examples of the media type; see Working With Examples.
encoding	Map[`string`, Encoding Object]	A map between a property name and its encoding information. The key, being the property name, *MUST* exist in the schema as a property. The `encoding` field *SHALL* only apply to Request Body Objects, and only when the media type is `multipart` or `application/x-www-form-urlencoded`. If no Encoding Object is provided for a property, the behavior is determined by the default values documented for the Encoding Object.

This object *MAY* be extended with Specification Extensions.

§ 4.8.14.2 Media Type Examples

```
{
  "application/json": {
    "schema": {
      "$ref": "#/components/schemas/Pet"
    },
    "examples": {
      "cat": {
        "summary": "An example of a cat",
        "value": {
          "name": "Fluffy",
          "petType": "Cat",
          "color": "White",
          "gender": "male",
```

```
            "breed": "Persian"
          }
        },
        "dog": {
          "summary": "An example of a dog with a cat's name",
          "value": {
            "name": "Puma",
            "petType": "Dog",
            "color": "Black",
            "gender": "Female",
            "breed": "Mixed"
          }
        },
        "frog": {
          "$ref": "#/components/examples/frog-example"
        }
      }
    }
  }

  application/json:
    schema:
      $ref: '#/components/schemas/Pet'
    examples:
      cat:
        summary: An example of a cat
        value:
          name: Fluffy
          petType: Cat
          color: White
          gender: male
          breed: Persian
      dog:
        summary: An example of a dog with a cat's name
        value:
          name: Puma
          petType: Dog
          color: Black
          gender: Female
          breed: Mixed
      frog:
        $ref: '#/components/examples/frog-example'
```

§ 4.8.14.3 Considerations for File Uploads

In contrast to OpenAPI 2.0, `file` input/output content in OAS 3.x is described with the same semantics as any other schema type.

In contrast to OAS 3.0, the `format` keyword has no effect on the content-encoding of the schema in OAS 3.1. Instead, JSON Schema's `contentEncoding` and `contentMediaType` keywords are used. See Working With Binary Data for how to model various scenarios with these keywords, and how to migrate from the previous `format` usage.

Examples:

Content transferred in binary (octet-stream) *MAY* omit `schema`:

```
# a PNG image as a binary file:
content:
  image/png: {}
```

```
# an arbitrary binary file:
content:
  application/octet-stream: {}
```

```
# arbitrary JSON without constraints beyond being syntactically valid:
content:
  application/json: {}
```

These examples apply to either input payloads of file uploads or response payloads.

A `requestBody` for submitting a file in a `POST` operation may look like the following example:

```
requestBody:
  content:
    application/octet-stream: {}
```

In addition, specific media types *MAY* be specified:

```
# multiple, specific media types may be specified:
requestBody:
  content:
    # a binary file of type png or jpeg
    image/jpeg: {}
    image/png: {}
```

To upload multiple files, a `multipart` media type *MUST* be used as shown under Example: Multipart Form with Multiple Files.

§ 4.8.14.4 Support for x-www-form-urlencoded Request Bodies

See Encoding the `x-www-form-urlencoded` Media Type for guidance and examples, both with and without the `encoding` field.

§ 4.8.14.5 Special Considerations for `multipart` Content

See Encoding `multipart` Media Types for further guidance and examples, both with and without the `encoding` field.

§ 4.8.15 Encoding Object

A single encoding definition applied to a single schema property. See Appendix B for a discussion of converting values of various types to string representations.

Properties are correlated with `multipart` parts using the `name` parameter of `Content-Disposition: form-data`, and with `application/x-www-form-urlencoded` using the query string parameter names. In both cases, their order is implementation-defined.

See Appendix E for a detailed examination of percent-encoding concerns for form media types.

§ 4.8.15.1 Fixed Fields

§ 4.8.15.1.1 Common Fixed Fields

These fields *MAY* be used either with or without the RFC6570-style serialization fields defined in the next section below.

Field Name	Type	Description
contentType	`string`	The `Content-Type` for encoding a specific property. The value is a comma-separated list, each

Field Name	Type	Description
		element of which is either a specific media type (e.g. `image/png`) or a wildcard media type (e.g. `image/*`). Default value depends on the property type as shown in the table below.
headers	Map[`string`, Header Object | Reference Object]	A map allowing additional information to be provided as headers. `Content-Type` is described separately and *SHALL* be ignored in this section. This field *SHALL* be ignored if the request body media type is not a `multipart`.

This object *MAY* be extended with Specification Extensions.

The default values for `contentType` are as follows, where an *n/a* in the `contentEncoding` column means that the presence or value of `contentEncoding` is irrelevant:

`type`	`contentEncoding`	Default `contentType`
absent	*n/a*	`application/octet-stream`
`string`	*present*	`application/octet-stream`
`string`	*absent*	`text/plain`
`number`, `integer`, or `boolean`	*n/a*	`text/plain`
`object`	*n/a*	`application/json`
`array`	*n/a*	according to the `type` of the `items` schema

Determining how to handle a `type` value of `null` depends on how `null` values are being serialized. If `null` values are entirely omitted, then the `contentType` is irrelevant. See Appendix B for a discussion of data type conversion options.

§ 4.8.15.1.2 Fixed Fields for RFC6570-style Serialization

Field Name	Type	Description
style	`string`	Describes how a specific property value will be serialized depending on its type. See Parameter Object for details on the `style` field. The behavior follows the same values as `query` parameters, including default values. Note that the initial `?` used in query strings is not used in `application/x-www-form-urlencoded` message bodies, and *MUST* be removed (if using an RFC6570 implementation) or simply not added (if constructing the string manually). This field *SHALL* be ignored if the request body media type is not `application/x-www-form-urlencoded` or `multipart/form-data`. If a value is explicitly defined, then the value of `contentType` (implicit or explicit) *SHALL* be ignored.
explode	`boolean`	When this is true, property values of type `array` or `object` generate separate parameters for each value of the array, or key-value-pair of the map. For other types of properties this field has no effect. When `style` is `"form"`, the default value is `true`. For all other styles, the default value is `false`. Note that despite `false` being the default for `deepObject`, the combination of `false` with `deepObject` is undefined. This field *SHALL* be ignored if the request body media type is not `application/x-www-form-urlencoded` or `multipart/form-data`. If a value is explicitly defined, then the value of `contentType` (implicit or explicit) *SHALL* be ignored.
allowReserved	`boolean`	When this is true, parameter values are serialized using reserved expansion, as defined by [RFC6570] Section 3.2.3, which allows RFC3986's reserved character set, as well as percent-encoded triples, to pass through unchanged, while still percent-encoding all other disallowed characters (including `%` outside of percent-encoded triples). Applications are still responsible for percent-encoding reserved characters that are not allowed in the query string (`[`, `]`, `#`), or have a special meaning in `application/x-`

Field Name	Type	Description
		`www-form-urlencoded` (`-`, `&`, `+`); see Appendices C and E for details. The default value is `false`. This field *SHALL* be ignored if the request body media type is not `application/x-www-form-urlencoded` or `multipart/form-data`. If a value is explicitly defined, then the value of `contentType` (implicit or explicit) *SHALL* be ignored.

See also Appendix C: Using RFC6570 Implementations for additional guidance, including on difficulties caused by the interaction between RFC6570's percent-encoding rules and the `multipart/form-data` media type.

Note that the presence of at least one of `style`, `explode`, or `allowReserved` with an explicit value is equivalent to using `schema` with `in: "query"` Parameter Objects. The absence of all three of those fields is the equivalent of using `content`, but with the media type specified in `contentType` rather than through a Media Type Object.

§ 4.8.15.2 Encoding the `x-www-form-urlencoded` Media Type

To submit content using form url encoding via [RFC1866], use the `application/x-www-form-urlencoded` media type in the Media Type Object under the Request Body Object. This configuration means that the request body *MUST* be encoded per [RFC1866] when passed to the server, after any complex objects have been serialized to a string representation.

See Appendix E for a detailed examination of percent-encoding concerns for form media types.

§ 4.8.15.2.1 Example: URL Encoded Form with JSON Values

When there is no `encoding` field, the serialization strategy is based on the Encoding Object's default values:

```
requestBody:
  content:
    application/x-www-form-urlencoded:
      schema:
        type: object
        properties:
```

```yaml
          id:
            type: string
            format: uuid
          address:
            # complex types are stringified to support RFC 1866
            type: object
            properties: {}
```

With this example, consider an `id` of `f81d4fae-7dec-11d0-a765-00a0c91e6bf6` and a US-style address (with ZIP+4) as follows:

```json
{
  "streetAddress": "123 Example Dr.",
  "city": "Somewhere",
  "state": "CA",
  "zip": "99999+1234"
}
```

Assuming the most compact representation of the JSON value (with unnecessary whitespace removed), we would expect to see the following request body, where space characters have been replaced with `+` and `+`, `"`, `{`, and `}` have been percent-encoded to `%2B`, `%22`, `%7B`, and `%7D`, respectively:

```
id=f81d4fae-7dec-11d0-a765-00a0c91e6bf6&address=%7B%22streetAddress%22:%22
```

Note that the `id` keyword is treated as `text/plain` per the Encoding Object's default behavior, and is serialized as-is. If it were treated as `application/json`, then the serialized value would be a JSON string including quotation marks, which would be percent-encoded as `%22`.

Here is the `id` parameter (without `address`) serialized as `application/json` instead of `text/plain`, and then encoded per RFC1866:

```
id=%22f81d4fae-7dec-11d0-a765-00a0c91e6bf6%22
```

§ 4.8.15.2.2 Example: URL Encoded Form with Binary Values

Note that `application/x-www-form-urlencoded` is a text format, which requires base64-encoding any binary data:

```yaml
requestBody:
  content:
    application/x-www-form-urlencoded:
      schema:
        type: object
```

```
        properties:
          name:
            type: string
          icon:
            # The default with "contentEncoding" is application/octet-stre
            # so we need to set image media type(s) in the Encoding Object
            type: string
            contentEncoding: base64url
  encoding:
    icon:
      contentType: image/png, image/jpeg
```

Given a name of `example` and a solid red 2x2-pixel PNG for `icon`, this would produce a request body of:

```
name=example&icon=iVBORw0KGgoAAAANSUhEUgAAAAIAAAACCAIAAAD91JpzAAAABGdBTUE/
```

Note that the `=` padding characters at the end need to be percent-encoded, even with the "URL safe" `contentEncoding: base64url`. Some base64-decoding implementations may be able to use the string without the padding per [RFC4648] Section 3.2. However, this is not guaranteed, so it may be more interoperable to keep the padding and rely on percent-decoding.

§ 4.8.15.3 Encoding `multipart` Media Types

It is common to use `multipart/form-data` as a `Content-Type` when transferring forms as request bodies. In contrast to OpenAPI 2.0, a `schema` is *REQUIRED* to define the input parameters to the operation when using `multipart` content. This supports complex structures as well as supporting mechanisms for multiple file uploads.

The `form-data` disposition and its `name` parameter are mandatory for `multipart/form-data` ([RFC7578] Section 4.2). Array properties are handled by applying the same `name` to multiple parts, as is recommended by [RFC7578] Section 4.3 for supplying multiple values per form field. See [RFC7578] Section 5 for guidance regarding non-ASCII part names.

Various other `multipart` types, most notable `multipart/mixed` ([RFC2046] Section 5.1.3) neither require nor forbid specific `Content-Disposition` values, which means care must be taken to ensure that any values used are supported by all relevant software. It is not currently possible to correlate schema properties with unnamed, ordered parts in media types such as `multipart/mixed`, but implementations *MAY* choose to support such types when `Content-Disposition: form-data` is used with a `name` parameter.

Note that there are significant restrictions on what headers can be used with `multipart` media types in general ([RFC2046] Section 5.1) and `multi-part/form-data` in particular ([RFC7578]

Section 4.8).

Note also that `Content-Transfer-Encoding` is deprecated for `multipart/form-data` ([RFC7578] Section 4.7) where binary data is supported, as it is in HTTP.

+Using `contentEncoding` for a multipart field is equivalent to specifying an Encoding Object with a `headers` field containing `Content-Transfer-Encoding` with a schema that requires the value used in `contentEncoding`. +If `contentEncoding` is used for a multipart field that has an Encoding Object with a `headers` field containing `Content-Transfer-Encoding` with a schema that disallows the value from `contentEncoding`, the result is undefined for serialization and parsing.

Note that as stated in Working with Binary Data, if the Encoding Object's `contentType`, whether set explicitly or implicitly through its default value rules, disagrees with the `contentMediaType` in a Schema Object, the `contentMediaType` *SHALL* be ignored. Because of this, and because the Encoding Object's `contentType` defaulting rules do not take the Schema Object's`contentMediaType` into account, the use of `contentMediaType` with an Encoding Object is *NOT RECOMMENDED*.

See Appendix E for a detailed examination of percent-encoding concerns for form media types.

§ 4.8.15.3.1 Example: Basic Multipart Form

When the `encoding` field is *not* used, the encoding is determined by the Encoding Object's defaults:

```yaml
requestBody:
  content:
    multipart/form-data:
      schema:
        type: object
        properties:
          id:
            # default for primitives without a special format is text/plai
            type: string
            format: uuid
          profileImage:
            # default for string with binary format is `application/octet-
            type: string
            format: binary
          addresses:
            # default for arrays is based on the type in the `items`
```

```
        # subschema, which is an object, so `application/json`
        type: array
        items:
          $ref: '#/components/schemas/Address'
```

§ 4.8.15.3.2 Example: Multipart Form with Encoding Objects

Using `encoding`, we can set more specific types for binary data, or non-JSON formats for complex values. We can also describe headers for each part:

```
requestBody:
  content:
    multipart/form-data:
      schema:
        type: object
        properties:
          id:
            # default is `text/plain`
            type: string
            format: uuid
          addresses:
            # default based on the `items` subschema would be
            # `application/json`, but we want these address objects
            # serialized as `application/xml` instead
            description: addresses in XML format
            type: array
            items:
              $ref: '#/components/schemas/Address'
          profileImage:
            # default is application/octet-stream, but we can declare
            # a more specific image type or types
            type: string
            format: binary
      encoding:
        addresses:
          # require XML Content-Type in utf-8 encoding
          # This is applied to each address part corresponding
          # to each address in he array
          contentType: application/xml; charset=utf-8
        profileImage:
          # only accept png or jpeg
          contentType: image/png, image/jpeg
          headers:
            X-Rate-Limit-Limit:
```

```
            description: The number of allowed requests in the current p
            schema:
              type: integer
```

§ 4.8.15.3.3 Example: Multipart Form with Multiple Files

In accordance with [RFC7578] Section 4.3, multiple files for a single form field are uploaded using the same name (`file` in this example) for each file's part:

```
requestBody:
  content:
    multipart/form-data:
      schema:
        properties:
          # The property name 'file' will be used for all files.
          file:
            type: array
            items: {}
```

As seen in the Encoding Object's `contentType` field documentation, the empty schema for `items` indicates a media type of `application/octet-stream`.

§ 4.8.16 Responses Object

A container for the expected responses of an operation. The container maps a HTTP response code to the expected response.

The documentation is not necessarily expected to cover all possible HTTP response codes because they may not be known in advance. However, documentation is expected to cover a successful operation response and any known errors.

The `default` *MAY* be used as a default Response Object for all HTTP codes that are not covered individually by the Responses Object.

The Responses Object *MUST* contain at least one response code, and if only one response code is provided it *SHOULD* be the response for a successful operation call.

§ 4.8.16.1 Fixed Fields

Field Name	Type	Description
default	Response Object | Reference Object	The documentation of responses other than the ones declared for specific HTTP response codes. Use this field to cover undeclared responses.

§ 4.8.16.2 Patterned Fields

Field Pattern	Type	Description
HTTP Status Code	Response Object | Reference Object	Any HTTP status code can be used as the property name, but only one property per code, to describe the expected response for that HTTP status code. This field *MUST* be enclosed in quotation marks (for example, “200”) for compatibility between JSON and YAML. To define a range of response codes, this field *MAY* contain the uppercase wildcard character `X`. For example, `2XX` represents all response codes between `200` and `299`. Only the following range definitions are allowed: `1XX`, `2XX`, `3XX`, `4XX`, and `5XX`. If a response is defined using an explicit code, the explicit code definition takes precedence over the range definition for that code.

This object *MAY* be extended with Specification Extensions.

§ 4.8.16.3 Responses Object Example

A 200 response for a successful operation and a default response for others (implying an error):

```
{
  "200": {
    "description": "a pet to be returned",
    "content": {
      "application/json": {
```

```
          "schema": {
            "$ref": "#/components/schemas/Pet"
          }
        }
      }
    },
    "default": {
      "description": "Unexpected error",
      "content": {
        "application/json": {
          "schema": {
            "$ref": "#/components/schemas/ErrorModel"
          }
        }
      }
    }
  }
```

```
  '200':
    description: a pet to be returned
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Pet'
  default:
    description: Unexpected error
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/ErrorModel'
```

§ 4.8.17 Response Object

Describes a single response from an API operation, including design-time, static `links` to operations based on the response.

§ 4.8.17.1 Fixed Fields

Field Name	Type	Description
description	`string`	***REQUIRED***. A description of the response. [CommonMark] syntax *MAY* be used for rich text representation.
headers	Map[`string`, Header Object | Reference Object]	Maps a header name to its definition. [RFC7230] Section 3.2 states header names are case insensitive. If a response header is defined with the name `"Content-Type"`, it *SHALL* be ignored.
content	Map[`string`, Media Type Object]	A map containing descriptions of potential response payloads. The key is a media type or media type range, see [RFC7231] Appendix D, and the value describes it. For responses that match multiple keys, only the most specific key is applicable. e.g. `"text/plain"` overrides `"text/*"`
links	Map[`string`, Link Object | Reference Object]	A map of operations links that can be followed from the response. The key of the map is a short name for the link, following the naming constraints of the names for Component Objects.

This object *MAY* be extended with Specification Extensions.

§ 4.8.17.2 Response Object Examples

Response of an array of a complex type:

```
{
  "description": "A complex object array response",
  "content": {
    "application/json": {
      "schema": {
        "type": "array",
        "items": {
```

```
          "$ref": "#/components/schemas/VeryComplexType"
        }
      }
    }
  }
}
```

```
description: A complex object array response
content:
  application/json:
    schema:
      type: array
      items:
        $ref: '#/components/schemas/VeryComplexType'
```

Response with a string type:

```
{
  "description": "A simple string response",
  "content": {
    "text/plain": {
      "schema": {
        "type": "string"
      }
    }
  }
}
```

```
description: A simple string response
content:
  text/plain:
    schema:
      type: string
```

Plain text response with headers:

```
{
  "description": "A simple string response",
  "content": {
    "text/plain": {
      "schema": {
        "type": "string"
      },
      "example": "whoa!"
    }
  },
  "headers": {
    "X-Rate-Limit-Limit": {
```

```
      "description": "The number of allowed requests in the current perioc
      "schema": {
        "type": "integer"
      }
    },
    "X-Rate-Limit-Remaining": {
      "description": "The number of remaining requests in the current peri
      "schema": {
        "type": "integer"
      }
    },
    "X-Rate-Limit-Reset": {
      "description": "The number of seconds left in the current period",
      "schema": {
        "type": "integer"
      }
    }
  }
}
```

```
description: A simple string response
content:
  text/plain:
    schema:
      type: string
    example: 'whoa!'
headers:
  X-Rate-Limit-Limit:
    description: The number of allowed requests in the current period
    schema:
      type: integer
  X-Rate-Limit-Remaining:
    description: The number of remaining requests in the current period
    schema:
      type: integer
  X-Rate-Limit-Reset:
    description: The number of seconds left in the current period
    schema:
      type: integer
```

Response with no return value:

```
{
  "description": "object created"
}
```

```
description: object created
```

§ 4.8.18 Callback Object

A map of possible out-of band callbacks related to the parent operation. Each value in the map is a Path Item Object that describes a set of requests that may be initiated by the API provider and the expected responses. The key value used to identify the Path Item Object is an expression, evaluated at runtime, that identifies a URL to use for the callback operation.

To describe incoming requests from the API provider independent from another API call, use the `webhooks` field.

§ 4.8.18.1 Patterned Fields

Field Pattern	Type	Description
{expression}	Path Item Object	A Path Item Object used to define a callback request and expected responses. A complete example is available.

This object *MAY* be extended with Specification Extensions.

§ 4.8.18.2 Key Expression

The key that identifies the Path Item Object is a runtime expression that can be evaluated in the context of a runtime HTTP request/response to identify the URL to be used for the callback request. A simple example might be `$request.body#/url`. However, using a runtime expression the complete HTTP message can be accessed. This includes accessing any part of a body that a JSON Pointer [RFC6901] can reference.

For example, given the following HTTP request:

```
POST /subscribe/myevent?queryUrl=https://clientdomain.com/stillrunning HTT
Host: example.org
Content-Type: application/json
Content-Length: 188

{
  "failedUrl": "https://clientdomain.com/failed",
  "successUrls": [
```

```
    "https://clientdomain.com/fast",
    "https://clientdomain.com/medium",
    "https://clientdomain.com/slow"
  ]
}
```

resulting in:

```
201 Created
Location: https://example.org/subscription/1
```

The following examples show how the various expressions evaluate, assuming the callback operation has a path parameter named `eventType` and a query parameter named `queryUrl`.

Expression	Value
$url	https://example.org/subscribe/myevent?queryUrl=https://clientdomain.com/stillrunning
$method	POST
$request.path.eventType	myevent
$request.query.queryUrl	https://clientdomain.com/stillrunning
$request.header.content-type	application/json
$request.body#/failedUrl	https://clientdomain.com/failed
$request.body#/successUrls/1	https://clientdomain.com/medium
$response.header.Location	https://example.org/subscription/1

§ 4.8.18.3 Callback Object Examples

The following example uses the user provided `queryUrl` query string parameter to define the callback URL. This is similar to a webhook, but differs in that the callback only occurs because of the initial request that sent the `queryUrl`.

```
myCallback:
  '{$request.query.queryUrl}':
    post:
      requestBody:
        description: Callback payload
        content:
```

```
          application/json:
            schema:
              $ref: '#/components/schemas/SomePayload'
      responses:
        '200':
          description: callback successfully processed
```

The following example shows a callback where the server is hard-coded, but the query string parameters are populated from the `id` and `email` property in the request body.

```
transactionCallback:
  'http://notificationServer.com?transactionId={$request.body#/id}&email={
    post:
      requestBody:
        description: Callback payload
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/SomePayload'
      responses:
        '200':
          description: callback successfully processed
```

§ 4.8.19 Example Object

An object grouping an internal or external example value with basic `summary` and `description` metadata. This object is typically used in fields named `examples` (plural), and is a referenceable alternative to older `example` (singular) fields that do not support referencing or metadata.

Examples allow demonstration of the usage of properties, parameters and objects within OpenAPI.

§ 4.8.19.1 Fixed Fields

Field Name	Type	Description
summary	`string`	Short description for the example.
description	`string`	Long description for the example. [CommonMark] syntax *MAY* be used for rich text representation.
value	Any	Embedded literal example. The `value` field and `externalValue` field are mutually exclusive. To represent

Field Name	Type	Description
		examples of media types that cannot naturally represented in JSON or YAML, use a string value to contain the example, escaping where necessary.
externalValue	`string`	A URI that identifies the literal example. This provides the capability to reference examples that cannot easily be included in JSON or YAML documents. The `value` field and `externalValue` field are mutually exclusive. See the rules for resolving Relative References.

This object *MAY* be extended with Specification Extensions.

In all cases, the example value *SHOULD* be compatible with the schema of its associated value. Tooling implementations *MAY* choose to validate compatibility automatically, and reject the example value(s) if incompatible.

§ 4.8.19.2 Working with Examples

Example Objects can be used in both Parameter Objects and Media Type Objects. In both Objects, this is done through the `examples` (plural) field. However, there are several other ways to provide examples: The `example` (singular) field that is mutually exclusive with `examples` in both Objects, and two keywords (the deprecated singular `example` and the current plural `examples`, which takes an array of examples) in the Schema Object that appears in the `schema` field of both Objects. Each of these fields has slightly different considerations.

The Schema Object's fields are used to show example values without regard to how they might be formatted as parameters or within media type representations. The `examples` array is part of JSON Schema and is the preferred way to include examples in the Schema Object, while `example` is retained purely for compatibility with older versions of the OpenAPI Specification.

The mutually exclusive fields in the Parameter or Media Type Objects are used to show example values which *SHOULD* both match the schema and be formatted as they would appear as a serialized parameter or within a media type representation. The exact serialization and encoding is determined by various fields in the Parameter Object, or in the Media Type Object's Encoding Object. Because examples using these fields represent the final serialized form of the data, they *SHALL override* any `example` in the corresponding Schema Object.

The singular `example` field in the Parameter or Media Type Object is concise and convenient for simple examples, but does not offer any other advantages over using Example Objects under `examples`.

Some examples cannot be represented directly in JSON or YAML. For all three ways of providing examples, these can be shown as string values with any escaping necessary to make the string valid in the JSON or YAML format of documents that comprise the OpenAPI Description. With the Example Object, such values can alternatively be handled through the `externalValue` field.

§ 4.8.19.3 Example Object Examples

In a request body:

```yaml
requestBody:
  content:
    'application/json':
      schema:
        $ref: '#/components/schemas/Address'
      examples:
        foo:
          summary: A foo example
          value:
            foo: bar
        bar:
          summary: A bar example
          value:
            bar: baz
    application/xml:
      examples:
        xmlExample:
          summary: This is an example in XML
          externalValue: https://example.org/examples/address-example.xml
    text/plain:
      examples:
        textExample:
          summary: This is a text example
          externalValue: https://foo.bar/examples/address-example.txt
```

In a parameter:

```yaml
parameters:
  - name: zipCode
    in: query
    schema:
      type: string
      format: zip-code
```

```yaml
    examples:
      zip-example:
        $ref: '#/components/examples/zip-example'
```

In a response:

```yaml
responses:
  '200':
    description: your car appointment has been booked
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/SuccessResponse'
        examples:
          confirmation-success:
            $ref: '#/components/examples/confirmation-success'
```

Two different uses of JSON strings:

First, a request or response body that is just a JSON string (not an object containing a string):

```json
"application/json": {
  "schema": {
    "type": "string"
  },
  "examples": {
    "jsonBody": {
      "description": "A body of just the JSON string \"json\"",
      "value": "json"
    }
  }
}
```

```yaml
application/json:
  schema:
    type: string
  examples:
    jsonBody:
      description: 'A body of just the JSON string "json"'
      value: json
```

In the above example, we can just show the JSON string (or any JSON value) as-is, rather than stuffing a serialized JSON value into a JSON string, which would have looked like `"\"json\""`.

In contrast, a JSON string encoded inside of a URL-style form body:

```
"application/x-www-form-urlencoded": {
  "schema": {
    "type": "object",
    "properties": {
      "jsonValue": {
        "type": "string"
      }
    }
  },
  "encoding": {
    "jsonValue": {
      "contentType": "application/json"
    }
  },
  "examples": {
    "jsonFormValue": {
      "description": "The JSON string \"json\" as a form value",
      "value": "jsonValue=%22json%22"
    }
  }
}
```

```
application/x-www-form-urlencoded:
  schema:
    type: object
    properties:
      jsonValue:
        type: string
  encoding:
    jsonValue:
      contentType: application/json
  examples:
    jsonFormValue:
      description: 'The JSON string "json" as a form value'
      value: jsonValue=%22json%22
```

In this example, the JSON string had to be serialized before encoding it into the URL form value, so the example includes the quotation marks that are part of the JSON serialization, which are then URL percent-encoded.

§ 4.8.20 Link Object

The Link Object represents a possible design-time link for a response. The presence of a link does not guarantee the caller's ability to successfully invoke it, rather it provides a known relationship and traversal mechanism between responses and other operations.

Unlike *dynamic* links (i.e. links provided **in** the response payload), the OAS linking mechanism does not require link information in the runtime response.

For computing links and providing instructions to execute them, a runtime expression is used for accessing values in an operation and using them as parameters while invoking the linked operation.

§ 4.8.20.1 Fixed Fields

Field Name	Type	Description
operationRef	`string`	A URI reference to an OAS operation. This field is mutually exclusive of the `operationId` field, and *MUST* point to an Operation Object. Relative `operationRef` values *MAY* be used to locate an existing Operation Object in the OpenAPI Description.
operationId	`string`	The name of an *existing*, resolvable OAS operation, as defined with a unique `operationId`. This field is mutually exclusive of the `operationRef` field.
parameters	Map[`string`, Any | {expression}]	A map representing parameters to pass to an operation as specified with `operationId` or identified via `operationRef`. The key is the parameter name to be used (optionally qualified with the parameter location, e.g. `path.id` for an `id` parameter in the path), whereas the value can be a constant or an expression to be evaluated and passed to the linked operation.
requestBody	Any | {expression}	A literal value or {expression} to use as a request body when calling the target operation.
description	`string`	A description of the link. [CommonMark] syntax *MAY* be used for rich text representation.

Field Name	Type	Description
server	Server Object	A server object to be used by the target operation.

This object *MAY* be extended with Specification Extensions.

A linked operation *MUST* be identified using either an `operationRef` or `operationId`. The identified or reference operation *MUST* be unique, and in the case of an `operationId`, it *MUST* be resolved within the scope of the OpenAPI Description (OAD). Because of the potential for name clashes, the `operationRef` syntax is preferred for multi-document OADs. However, because use of an operation depends on its URL path template in the Paths Object, operations from any Path Item Object that is referenced multiple times within the OAD cannot be resolved unambiguously. In such ambiguous cases, the resulting behavior is implementation-defined and *MAY* result in an error.

Note that it is not possible to provide a constant value to `parameters` that matches the syntax of a runtime expression. It is possible to have ambiguous parameter names, e.g. `name: "id", in: "path"` and `name: "path.id", in: "query"`; this is *NOT RECOMMENDED* and the behavior is implementation-defined, however implementations *SHOULD* prefer the qualified interpretation (`path.id` as a path parameter), as the names can always be qualified to disambiguate them (e.g. using `query.path.id` for the query parameter).

§ 4.8.20.2 Examples

Computing a link from a request operation where the `$request.path.id` is used to pass a request parameter to the linked operation.

```yaml
paths:
  /users/{id}:
    parameters:
      - name: id
        in: path
        required: true
        description: the user identifier, as userId
        schema:
          type: string
    get:
      responses:
        '200':
          description: the user being returned
          content:
            application/json:
              schema:
```

```yaml
                type: object
                properties:
                  uuid: # the unique user id
                    type: string
                    format: uuid
        links:
          address:
            # the target link operationId
            operationId: getUserAddress
            parameters:
              # get the `id` field from the request path parameter named
              userid: $request.path.id
# the path item of the linked operation
/users/{userid}/address:
  parameters:
    - name: userid
      in: path
      required: true
      description: the user identifier, as userId
      schema:
        type: string
  # linked operation
  get:
    operationId: getUserAddress
    responses:
      '200':
        description: the user's address
```

When a runtime expression fails to evaluate, no parameter value is passed to the target operation.

Values from the response body can be used to drive a linked operation.

```yaml
links:
  address:
    operationId: getUserAddressByUUID
    parameters:
      # get the `uuid` field from the `uuid` field in the response body
      userUuid: $response.body#/uuid
```

Clients follow all links at their discretion. Neither permissions nor the capability to make a successful call to that link is guaranteed solely by the existence of a relationship.

§ 4.8.20.3 `operationRef` Examples

As references to `operationId` *MAY* NOT be possible (the `operationId` is an optional field in an Operation Object), references *MAY* also be made through a relative `operationRef`:

```
links:
  UserRepositories:
    # returns array of '#/components/schemas/repository'
    operationRef: '#/paths/~12.0~1repositories~1%7Busername%7D/get'
    parameters:
      username: $response.body#/username
```

or a URI `operationRef`:

```
links:
  UserRepositories:
    # returns array of '#/components/schemas/repository'
    operationRef: https://na2.gigantic-server.com/#/paths/~12.0~1repositor
    parameters:
      username: $response.body#/username
```

Note that in the use of `operationRef` the *escaped forward-slash* is necessary when using JSON Pointer, and it is necessary to URL-encode `{` and `}` as `%7B` and `%7D`, respectively, when using JSON Pointer as URI fragments.

§ 4.8.20.4 Runtime Expressions

Runtime expressions allow defining values based on information that will only be available within the HTTP message in an actual API call. This mechanism is used by Link Objects and Callback Objects.

The runtime expression is defined by the following [ABNF] syntax

```
expression = "$url" / "$method" / "$statusCode" / "$request." source /
source     = header-reference / query-reference / path-reference / boc
header-reference = "header." token
query-reference  = "query." name
path-reference   = "path." name
body-reference   = "body" ["#" json-pointer ]
json-pointer    = *( "/" reference-token )
reference-token = *( unescaped / escaped )
```

```
unescaped       = %x00-2E / %x30-7D / %x7F-10FFFF
                ; %x2F ('/') and %x7E ('~') are excluded from 'unescap
escaped         = "~" ( "0" / "1" )
                ; representing '~' and '/', respectively
name = *( CHAR )
token = 1*tchar
tchar = "!" / "#" / "$" / "%" / "&" / "'" / "*" / "+" / "-" / "."
      / "^" / "_" / "`" / "|" / "~" / DIGIT / ALPHA
```

Here, `json-pointer` is taken from [RFC6901], `char` from [RFC7159] Section 7 and `token` from [RFC7230] Section 3.2.6.

The `name` identifier is case-sensitive, whereas `token` is not.

The table below provides examples of runtime expressions and examples of their use in a value:

§ 4.8.20.5 Examples

Source Location	**example expression**	**notes**
HTTP Method	`$method`	The allowable values for the `$method` will be those for the HTTP operation.
Requested media type	`$request.header.accept`	
Request parameter	`$request.path.id`	Request parameters *MUST* be declared in the `parameters` section of the parent operation or they cannot be evaluated. This includes request headers.
Request body property	`$request.body#/user/uuid`	In operations which accept payloads, references may be made to portions of the `requestBody` or the entire body.
Request URL	`$url`	
Response value	`$response.body#/status`	In operations which return payloads, references may be made to portions of the response body or the entire body.

Source Location	example expression	notes
Response header	`$response.header.Server`	Single header values only are available

Runtime expressions preserve the type of the referenced value. Expressions can be embedded into string values by surrounding the expression with `{}` curly braces.

§ 4.8.21 Header Object

Describes a single header for HTTP responses and for individual parts in `multipart` representations; see the relevant Response Object and Encoding Object documentation for restrictions on which headers can be described.

The Header Object follows the structure of the Parameter Object, including determining its serialization strategy based on whether `schema` or `content` is present, with the following changes:

1. `name` *MUST NOT* be specified, it is given in the corresponding `headers` map.
2. `in` *MUST NOT* be specified, it is implicitly in `header`.
3. All traits that are affected by the location *MUST* be applicable to a location of `header` (for example, `style`). This means that `allowEmptyValue` and `allowReserved` *MUST NOT* be used, and `style`, if used, *MUST* be limited to `"simple"`.

§ 4.8.21.1 Fixed Fields

§ 4.8.21.1.1 Common Fixed Fields

These fields *MAY* be used with either `content` or `schema`.

Field Name	Type	Description
description	`string`	A brief description of the header. This could contain examples of use. [CommonMark] syntax *MAY* be used for rich text

Field Name	Type	Description
		representation.
required	`boolean`	Determines whether this header is mandatory. The default value is `false`.
deprecated	`boolean`	Specifies that the header is deprecated and *SHOULD* be transitioned out of usage. Default value is `false`.

This object *MAY* be extended with Specification Extensions.

§ 4.8.21.1.2 Fixed Fields for use with `schema`

For simpler scenarios, a `schema` and `style` can describe the structure and syntax of the header. When `example` or `examples` are provided in conjunction with the `schema` field, the example *MUST* follow the prescribed serialization strategy for the header.

Serializing with `schema` is *NOT RECOMMENDED* for headers with parameters (name=value pairs following a `;`) in their values, or where values might have non-URL-safe characters; see Appendix D for details.

When `example` or `examples` are provided in conjunction with the `schema` field, the example *SHOULD* match the specified schema and follow the prescribed serialization strategy for the header. The `example` and `examples` fields are mutually exclusive, and if either is present it *SHALL override* any `example` in the schema.

Field Name	Type	Description
style	`string`	Describes how the header value will be serialized. The default (and only legal value for headers) is `"simple"`.
explode	`boolean`	When this is true, header values of type `array` or `object` generate a single header whose value is a comma-separated list of the array items or key-value pairs of the map, see Style Examples. For other data types this field has no effect. The default value is `false`.
schema	Schema Object | Reference Object	The schema defining the type used for the header.

Field Name	Type	Description
example	Any	Example of the header's potential value; see Working With Examples.
examples	Map[`string`, Example Object | Reference Object]	Examples of the header's potential value; see Working With Examples.

See also Appendix C: Using RFC6570-Based Serialization for additional guidance.

§ 4.8.21.1.3 Fixed Fields for use with `content`

For more complex scenarios, the `content` field can define the media type and schema of the header, as well as give examples of its use. Using `content` with a `text/plain` media type is *RECOMMENDED* for headers where the `schema` strategy is not appropriate.

Field Name	Type	Description
content	Map[`string`, Media Type Object]	A map containing the representations for the header. The key is the media type and the value describes it. The map *MUST* only contain one entry.

§ 4.8.21.2 Header Object Example

A simple header of type `integer`:

```
"X-Rate-Limit-Limit": {
  "description": "The number of allowed requests in the current period",
  "schema": {
    "type": "integer"
  }
}
```

```
X-Rate-Limit-Limit:
  description: The number of allowed requests in the current period
  schema:
```

```
    type: integer
```

Requiring that a strong `ETag` header (with a value starting with `"` rather than `W/`) is present. Note the use of `content`, because using `schema` and `style` would require the `"` to be percent-encoded as `%22`:

```
"ETag": {
  "required": true,
  "content": {
    "text/plain": {
      "schema": {
        "type": "string",
        "pattern": "^\""
      }
    }
  }
}
```

```
ETag:
  required: true
  content:
    text/plain:
      schema:
        type: string
        pattern: ^"
```

§ 4.8.22 Tag Object

Adds metadata to a single tag that is used by the Operation Object. It is not mandatory to have a Tag Object per tag defined in the Operation Object instances.

§ 4.8.22.1 Fixed Fields

Field Name	Type	Description
name	`string`	***REQUIRED***. The name of the tag.
description	`string`	A description for the tag. [CommonMark] syntax *MAY* be used for rich text representation.
externalDocs	External Documentation	Additional external documentation for this tag.

Field Name	Type	Description
	Object	

This object *MAY* be extended with Specification Extensions.

§ 4.8.22.2 Tag Object Example

```
{
  "name": "pet",
  "description": "Pets operations"
}
```

```
name: pet
description: Pets operations
```

§ 4.8.23 Reference Object

A simple object to allow referencing other components in the OpenAPI Description, internally and externally.

The `$ref` string value contains a URI [RFC3986], which identifies the value being referenced.

See the rules for resolving Relative References.

§ 4.8.23.1 Fixed Fields

Field Name	Type	Description
$ref	`string`	***REQUIRED***. The reference identifier. This *MUST* be in the form of a URI.
summary	`string`	A short summary which by default *SHOULD* override that of the referenced component. If the referenced object-type does not allow a `summary` field, then this field has no effect.
description	`string`	A description which by default *SHOULD* override that of the referenced component. [CommonMark] syntax *MAY* be used for

Field Name	Type	Description
		rich text representation. If the referenced object-type does not allow a `description` field, then this field has no effect.

This object cannot be extended with additional properties, and any properties added *SHALL* be ignored.

Note that this restriction on additional properties is a difference between Reference Objects and Schema Objects that contain a `$ref` keyword.

§ 4.8.23.2 Reference Object Example

```
{
  "$ref": "#/components/schemas/Pet"
}
```

```
$ref: '#/components/schemas/Pet'
```

§ 4.8.23.3 Relative Schema Document Example

```
{
  "$ref": "Pet.json"
}
```

```
$ref: Pet.yaml
```

§ 4.8.23.4 Relative Documents with Embedded Schema Example

```
{
  "$ref": "definitions.json#/Pet"
}
```

```
$ref: definitions.yaml#/Pet
```

§ 4.8.24 Schema Object

The Schema Object allows the definition of input and output data types. These types can be objects, but also primitives and arrays. This object is a superset of the JSON Schema Specification Draft 2020-12. The empty schema (which allows any instance to validate) *MAY* be represented by the boolean value `true` and a schema which allows no instance to validate *MAY* be represented by the boolean value `false`.

For more information about the keywords, see JSON Schema Core and JSON Schema Validation.

Unless stated otherwise, the keyword definitions follow those of JSON Schema and do not add any additional semantics; this includes keywords such as `$schema`, `$id`, `$ref`, and `$dynamicRef` being URIs rather than URLs. Where JSON Schema indicates that behavior is defined by the application (e.g. for annotations), OAS also defers the definition of semantics to the application consuming the OpenAPI document.

§ 4.8.24.1 JSON Schema Keywords

The OpenAPI Schema Object dialect is defined as requiring the OAS base vocabulary, in addition to the vocabularies as specified in the JSON Schema Specification Draft 2020-12 general purpose meta-schema.

The OpenAPI Schema Object dialect for this version of the specification is identified by the URI `https://spec.openapis.org/oas/3.1/dialect/base` (the “OAS dialect schema id”).

The following keywords are taken from the JSON Schema specification but their definitions have been extended by the OAS:

- description - [CommonMark] syntax *MAY* be used for rich text representation.
- format - See Data Type Formats for further details. While relying on JSON Schema’s defined formats, the OAS offers a few additional predefined formats.

In addition to the JSON Schema keywords comprising the OAS dialect, the Schema Object supports keywords from any other vocabularies, or entirely arbitrary properties.

JSON Schema implementations *MAY* choose to treat keywords defined by the OpenAPI Specification’s base vocabulary as unknown keywords, due to its inclusion in the OAS dialect with a `$vocabulary` value of `false`. The OAS base vocabulary is comprised of the following keywords:

§ 4.8.24.2 Fixed Fields

Field Name	Type	Description
discriminator	Discriminator Object	Adds support for polymorphism. The discriminator is used to determine which of a set of schemas a payload is expected to satisfy. See Composition and Inheritance for more details.
xml	XML Object	This *MAY* be used only on property schemas. It has no effect on root schemas. Adds additional metadata to describe the XML representation of this property.
externalDocs	External Documentation Object	Additional external documentation for this schema.
example	Any	A free-form field to include an example of an instance for this schema. To represent examples that cannot be naturally represented in JSON or YAML, a string value can be used to contain the example with escaping where necessary. **Deprecated:** The `example` field has been deprecated in favor of the JSON Schema `examples` keyword. Use of `example` is discouraged, and later versions of this specification may remove it.

This object *MAY* be extended with Specification Extensions, though as noted, additional properties *MAY* omit the `x-` prefix within this object.

§ 4.8.24.3 Extended Validation with Annotations

JSON Schema Draft 2020-12 supports collecting annotations, including treating unrecognized keywords as annotations. OAS implementations *MAY* use such annotations, including extensions not recognized as part of a declared JSON Schema vocabulary, as the basis for further validation. Note that JSON Schema Draft 2020-12 does not require an `x-` prefix for extensions.

§ 4.8.24.3.1 Non-validating constraint keywords

The `format` keyword (when using default format-annotation vocabulary) and the `contentMediaType`, `contentEncoding`, and `contentSchema` keywords define constraints on the data, but are treated as annotations instead of being validated directly. Extended validation is one way that these constraints *MAY* be enforced.

§ 4.8.24.3.2 Validating `readOnly` and `writeOnly`

The `readOnly` and `writeOnly` keywords are annotations, as JSON Schema is not aware of how the data it is validating is being used. Validation of these keywords *MAY* be done by checking the annotation, the read or write direction, and (if relevant) the current value of the field. JSON Schema Validation Draft 2020-12 §9.4 defines the expectations of these keywords, including that a resource (described as the "owning authority") *MAY* either ignore a `readOnly` field or treat it as an error.

Fields that are both required and read-only are an example of when it is beneficial to ignore a `readOnly: true` constraint in a PUT, particularly if the value has not been changed. This allows correctly requiring the field on a GET and still using the same representation and schema with PUT. Even when read-only fields are not required, stripping them is burdensome for clients, particularly when the JSON data is complex or deeply nested.

Note that the behavior of `readOnly` in particular differs from that specified by version 3.0 of this specification.

§ 4.8.24.4 Data Modeling Techniques

§ 4.8.24.4.1 Composition and Inheritance (Polymorphism)

The OpenAPI Specification allows combining and extending model definitions using the `allOf` keyword of JSON Schema, in effect offering model composition. `allOf` takes an array of object definitions that are validated *independently* but together compose a single object.

While composition offers model extensibility, it does not imply a hierarchy between the models. To support polymorphism, the OpenAPI Specification adds the `discriminator` field. When used,

the `discriminator` indicates the name of the property that hints which schema definition is expected to validate the structure of the model. As such, the `discriminator` field *MUST* be a required field. There are two ways to define the value of a discriminator for an inheriting instance.

- Use the schema name.
- Override the schema name by overriding the property with a new value. If a new value exists, this takes precedence over the schema name.

§ 4.8.24.4.2 Generic (Template) Data Structures

Implementations *MAY* support defining generic or template data structures using JSON Schema's dynamic referencing feature:

- `$dynamicAnchor` identifies a set of possible schemas (including a default placeholder schema) to which a `$dynamicRef` can resolve
- `$dynamicRef` resolves to the first matching `$dynamicAnchor` encountered on its path from the schema entry point to the reference, as described in the JSON Schema specification

An example is included in the "Schema Object Examples" section below, and further information can be found on the Learn OpenAPI site's "Dynamic References" page.

§ 4.8.24.4.3 Annotated Enumerations

The Schema Object's `enum` keyword does not allow associating descriptions or other information with individual values.

Implementations *MAY* support recognizing a `oneOf` or `anyOf` where each subschema in the keyword's array consists of a `const` keyword and annotations such as `title` or `description` as an enumerated type with additional information. The exact behavior of this pattern beyond what is required by JSON Schema is implementation-defined.

§ 4.8.24.4.4 XML Modeling

The xml field allows extra definitions when translating the JSON definition to XML. The XML Object contains additional information about the available options.

§ 4.8.24.5 Specifying Schema Dialects

It is important for tooling to be able to determine which dialect or meta-schema any given resource wishes to be processed with: JSON Schema Core, JSON Schema Validation, OpenAPI Schema dialect, or some custom meta-schema.

The `$schema` keyword *MAY* be present in any Schema Object that is a schema resource root, and if present *MUST* be used to determine which dialect should be used when processing the schema. This allows use of Schema Objects which comply with other drafts of JSON Schema than the default Draft 2020-12 support. Tooling *MUST* support the OAS dialect schema id, and *MAY* support additional values of `$schema`.

To allow use of a different default `$schema` value for all Schema Objects contained within an OAS document, a `jsonSchemaDialect` value may be set within the OpenAPI Object. If this default is not set, then the OAS dialect schema id *MUST* be used for these Schema Objects. The value of `$schema` within a resource root Schema Object always overrides any default.

For standalone JSON Schema documents that do not set `$schema`, or for Schema Objects in OpenAPI description documents that are *not* complete documents, the dialect *SHOULD* be assumed to be the OAS dialect. However, for maximum interoperability, it is *RECOMMENDED* that OpenAPI description authors explicitly set the dialect through `$schema` in such documents.

§ 4.8.24.6 Schema Object Examples

§ 4.8.24.6.1 Primitive Example

```json
{
  "type": "string",
  "format": "email"
}
```

```yaml
type: string
format: email
```

§ 4.8.24.6.2 Simple Model

```json
{
  "type": "object",
  "required": ["name"],
  "properties": {
    "name": {
      "type": "string"
    },
    "address": {
      "$ref": "#/components/schemas/Address"
    },
    "age": {
      "type": "integer",
      "format": "int32",
      "minimum": 0
    }
  }
}
```

```yaml
type: object
required:
  - name
properties:
  name:
    type: string
  address:
    $ref: '#/components/schemas/Address'
  age:
    type: integer
    format: int32
    minimum: 0
```

§ 4.8.24.6.3 Model with Map/Dictionary Properties

For a simple string to string mapping:

```json
{
  "type": "object",
  "additionalProperties": {
```

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

```
type: object
additionalProperties:
  type: string
```

For a string to model mapping:

```
{
  "type": "object",
  "additionalProperties": {
    "$ref": "#/components/schemas/ComplexModel"
  }
}
```

```
type: object
additionalProperties:
  $ref: '#/components/schemas/ComplexModel'
```

§ 4.8.24.6.4 Model with Annotated Enumeration

```
{
  "oneOf": [
    {
      "const": "RGB",
      "title": "Red, Green, Blue",
      "description": "Specify colors with the red, green, and blue additiv
    },
    {
      "const": "CMYK",
      "title": "Cyan, Magenta, Yellow, Black",
      "description": "Specify colors with the cyan, magenta, yellow, and b
    }
  ]
}
```

```
oneOf:
  - const: RGB
    title: Red, Green, Blue
    description: Specify colors with the red, green, and blue additive col
  - const: CMYK
    title: Cyan, Magenta, Yellow, Black
    description: Specify colors with the cyan, magenta, yellow, and black
```

§ 4.8.24.6.5 Model with Example

```json
{
  "type": "object",
  "properties": {
    "id": {
      "type": "integer",
      "format": "int64"
    },
    "name": {
      "type": "string"
    }
  },
  "required": ["name"],
  "examples": [
    {
      "name": "Puma",
      "id": 1
    }
  ]
}
```

```yaml
type: object
properties:
  id:
    type: integer
    format: int64
  name:
    type: string
required:
  - name
examples:
  - name: Puma
    id: 1
```

§ 4.8.24.6.6 Models with Composition

```json
{
  "components": {
    "schemas": {
```

```
        "ErrorModel": {
          "type": "object",
          "required": ["message", "code"],
          "properties": {
            "message": {
              "type": "string"
            },
            "code": {
              "type": "integer",
              "minimum": 100,
              "maximum": 600
            }
          }
        },
        "ExtendedErrorModel": {
          "allOf": [
            {
              "$ref": "#/components/schemas/ErrorModel"
            },
            {
              "type": "object",
              "required": ["rootCause"],
              "properties": {
                "rootCause": {
                  "type": "string"
                }
              }
            }
          ]
        }
      }
    }
  }
```

```
components:
  schemas:
    ErrorModel:
      type: object
      required:
        - message
        - code
      properties:
        message:
          type: string
        code:
          type: integer
          minimum: 100
```

```yaml
          maximum: 600
    ExtendedErrorModel:
      allOf:
        - $ref: '#/components/schemas/ErrorModel'
        - type: object
          required:
            - rootCause
          properties:
            rootCause:
              type: string
```

§ 4.8.24.6.7 Models with Polymorphism Support

```json
{
  "components": {
    "schemas": {
      "Pet": {
        "type": "object",
        "discriminator": {
          "propertyName": "petType"
        },
        "properties": {
          "name": {
            "type": "string"
          },
          "petType": {
            "type": "string"
          }
        },
        "required": ["name", "petType"]
      },
      "Cat": {
        "description": "A representation of a cat. Note that `Cat` will be
        "allOf": [
          {
            "$ref": "#/components/schemas/Pet"
          },
          {
            "type": "object",
            "properties": {
              "huntingSkill": {
                "type": "string",
                "description": "The measured skill for hunting",
                "default": "lazy",
```

```
                    "enum": ["clueless", "lazy", "adventurous", "aggressive"]
                  }
                },
                "required": ["huntingSkill"]
              }
            ]
          },
          "Dog": {
            "description": "A representation of a dog. Note that `Dog` will be
            "allOf": [
              {
                "$ref": "#/components/schemas/Pet"
              },
              {
                "type": "object",
                "properties": {
                  "packSize": {
                    "type": "integer",
                    "format": "int32",
                    "description": "the size of the pack the dog is from",
                    "default": 0,
                    "minimum": 0
                  }
                },
                "required": ["packSize"]
              }
            ]
          }
        }
      }
    }
```

```yaml
components:
  schemas:
    Pet:
      type: object
      discriminator:
        propertyName: petType
      properties:
        name:
          type: string
        petType:
          type: string
      required:
        - name
        - petType
    Cat: # "Cat" will be used as the discriminating value
```

```yaml
    description: A representation of a cat
    allOf:
      - $ref: '#/components/schemas/Pet'
      - type: object
        properties:
          huntingSkill:
            type: string
            description: The measured skill for hunting
            enum:
              - clueless
              - lazy
              - adventurous
              - aggressive
        required:
          - huntingSkill
  Dog: # "Dog" will be used as the discriminating value
    description: A representation of a dog
    allOf:
      - $ref: '#/components/schemas/Pet'
      - type: object
        properties:
          packSize:
            type: integer
            format: int32
            description: the size of the pack the dog is from
            default: 0
            minimum: 0
        required:
          - packSize
```

§ 4.8.24.6.8 Generic Data Structure Model

```yaml
{
  "components": {
    "schemas": {
      "genericArrayComponent": {
        "$id": "fully_generic_array",
        "type": "array",
        "items": {
          "$dynamicRef": "#generic-array"
        },
        "$defs": {
          "allowAll": {
            "$dynamicAnchor": "generic-array"
```

```
      }
    }
  },
  "numberArray": {
    "$id": "array_of_numbers",
    "$ref": "fully_generic_array",
    "$defs": {
      "numbersOnly": {
        "$dynamicAnchor": "generic-array",
        "type": "number"
      }
    }
  },
  "stringArray": {
    "$id": "array_of_strings",
    "$ref": "fully_generic_array",
    "$defs": {
      "stringsOnly": {
        "$dynamicAnchor": "generic-array",
        "type": "string"
      }
    }
  },
  "objWithTypedArray": {
    "$id": "obj_with_typed_array",
    "type": "object",
    "required": ["dataType", "data"],
    "properties": {
      "dataType": {
        "enum": ["string", "number"]
      }
    },
    "oneOf": [{
      "properties": {
        "dataType": {"const": "string"},
        "data": {"$ref": "array_of_strings"}
      }
    }, {
      "properties": {
        "dataType": {"const": "number"},
        "data": {"$ref": "array_of_numbers"}
      }
    }]
  }
```

```
    }
  }
}

components:
  schemas:
    genericArrayComponent:
      $id: fully_generic_array
      type: array
      items:
        $dynamicRef: '#generic-array'
      $defs:
        allowAll:
          $dynamicAnchor: generic-array
    numberArray:
      $id: array_of_numbers
      $ref: fully_generic_array
      $defs:
        numbersOnly:
          $dynamicAnchor: generic-array
          type: number
    stringArray:
      $id: array_of_strings
      $ref: fully_generic_array
      $defs:
        stringsOnly:
          $dynamicAnchor: generic-array
          type: string
    objWithTypedArray:
      $id: obj_with_typed_array
      type: object
      required:
      - dataType
      - data
      properties:
        dataType:
          enum:
          - string
          - number
      oneOf:
      - properties:
          dataType:
            const: string
          data:
            $ref: array_of_strings
      - properties:
          dataType:
```

```
      const: number
    data:
      $ref: array_of_numbers
```

§ 4.8.25 Discriminator Object

When request bodies or response payloads may be one of a number of different schemas, a Discriminator Object gives a hint about the expected schema of the document. This hint can be used to aid in serialization, deserialization, and validation. The Discriminator Object does this by implicitly or explicitly associating the possible values of a named property with alternative schemas.

Note that `discriminator` *MUST NOT* change the validation outcome of the schema.

§ 4.8.25.1 Fixed Fields

Field Name	Type	Description
propertyName	`string`	***REQUIRED***. The name of the property in the payload that will hold the discriminating value. This property *SHOULD* be required in the payload schema, as the behavior when the property is absent is undefined.
mapping	Map[`string`, `string`]	An object to hold mappings between payload values and schema names or URI references.

This object *MAY* be extended with Specification Extensions.

§ 4.8.25.2 Conditions for Using the Discriminator Object

The Discriminator Object is legal only when using one of the composite keywords `oneOf`, `anyOf`, `allOf`.

In both the `oneOf` and `anyOf` use cases, where those keywords are adjacent to `discriminator`, all possible schemas *MUST* be listed explicitly.

To avoid redundancy, the discriminator *MAY* be added to a parent schema definition, and all schemas building on the parent schema via an `allOf` construct may be used as an alternate

schema.

The `allOf` form of `discriminator` is *only* useful for non-validation use cases; validation with the parent schema with this form of `discriminator` *does not* perform a search for child schemas or use them in validation in any way. This is because `discriminator` cannot change the validation outcome, and no standard JSON Schema keyword connects the parent schema to the child schemas.

The behavior of any configuration of `oneOf`, `anyOf`, `allOf` and `discriminator` that is not described above is undefined.

§ 4.8.25.3 Options for Mapping Values to Schemas

The value of the property named in `propertyName` is used as the name of the associated schema under the Components Object, *unless* a `mapping` is present for that value. The `mapping` entry maps a specific property value to either a different schema component name, or to a schema identified by a URI. When using implicit or explicit schema component names, inline `oneOf` or `anyOf` subschemas are not considered. The behavior of a `mapping` value that is both a valid schema name and a valid relative URI reference is implementation-defined, but it is *RECOMMENDED* that it be treated as a schema name. To ensure that an ambiguous value (e.g. `"foo"`) is treated as a relative URI reference by all implementations, authors *MUST* prefix it with the `"."` path segment (e.g. `"./foo"`).

Mapping keys *MUST* be string values, but tooling *MAY* convert response values to strings for comparison. However, the exact nature of such conversions are implementation-defined.

§ 4.8.25.4 Examples

For these examples, assume all schemas are in the entry document of the OAD; for handling of `discriminator` in referenced documents see Resolving Implicit Connections.

In OAS 3.x, a response payload *MAY* be described to be exactly one of any number of types:

```yaml
MyResponseType:
  oneOf:
    - $ref: '#/components/schemas/Cat'
    - $ref: '#/components/schemas/Dog'
    - $ref: '#/components/schemas/Lizard'
```

which means the payload *MUST*, by validation, match exactly one of the schemas described by `Cat`, `Dog`, or `Lizard`. Deserialization of a `oneOf` can be a costly operation, as it requires determining which schema matches the payload and thus should be used in deserialization. This problem also exists for `anyOf` schemas. A `discriminator` *MAY* be used as a "hint" to improve the efficiency of selection of the matching schema. The `discriminator` field cannot change the validation result of the `oneOf`, it can only help make the deserialization more efficient and provide better error messaging. We can specify the exact field that tells us which schema is expected to match the instance:

```yaml
MyResponseType:
  oneOf:
    - $ref: '#/components/schemas/Cat'
    - $ref: '#/components/schemas/Dog'
    - $ref: '#/components/schemas/Lizard'
  discriminator:
    propertyName: petType
```

The expectation now is that a property with name `petType` *MUST* be present in the response payload, and the value will correspond to the name of a schema defined in the OpenAPI Description. Thus the response payload:

```json
{
  "id": 12345,
  "petType": "Cat"
}
```

will indicate that the `Cat` schema is expected to match this payload.

In scenarios where the value of the `discriminator` field does not match the schema name or implicit mapping is not possible, an optional `mapping` definition *MAY* be used:

```yaml
MyResponseType:
  oneOf:
    - $ref: '#/components/schemas/Cat'
    - $ref: '#/components/schemas/Dog'
    - $ref: '#/components/schemas/Lizard'
    - $ref: https://gigantic-server.com/schemas/Monster/schema.json
  discriminator:
    propertyName: petType
    mapping:
      dog: '#/components/schemas/Dog'
      monster: https://gigantic-server.com/schemas/Monster/schema.json
```

Here the discriminating value of `dog` will map to the schema `#/components/schemas/Dog`, rather than the default (implicit) value of `#/components/schemas/dog`. If the discriminating

value does not match an implicit or explicit mapping, no schema can be determined and validation *SHOULD* fail.

When used in conjunction with the `anyOf` construct, the use of the discriminator can avoid ambiguity for serializers/deserializers where multiple schemas may satisfy a single payload.

This example shows the `allOf` usage, which avoids needing to reference all child schemas in the parent:

```yaml
components:
  schemas:
    Pet:
      type: object
      required:
        - petType
      properties:
        petType:
          type: string
      discriminator:
        propertyName: petType
        mapping:
          dog: Dog
    Cat:
      allOf:
        - $ref: '#/components/schemas/Pet'
        - type: object
          # all other properties specific to a `Cat`
          properties:
            name:
              type: string
    Dog:
      allOf:
        - $ref: '#/components/schemas/Pet'
        - type: object
          # all other properties specific to a `Dog`
          properties:
            bark:
              type: string
    Lizard:
      allOf:
        - $ref: '#/components/schemas/Pet'
        - type: object
          # all other properties specific to a `Lizard`
          properties:
            lovesRocks:
              type: boolean
```

Validated against the `Pet` schema, a payload like this:

```
{
  "petType": "Cat",
  "name": "Misty"
}
```

will indicate that the `#/components/schemas/Cat` schema is expected to match. Likewise this payload:

```
{
  "petType": "dog",
  "bark": "soft"
}
```

will map to `#/components/schemas/Dog` because the `dog` entry in the `mapping` element maps to `Dog` which is the schema name for `#/components/schemas/Dog`.

§ 4.8.26 XML Object

A metadata object that allows for more fine-tuned XML model definitions.

When using arrays, XML element names are *not* inferred (for singular/plural forms) and the `name` field *SHOULD* be used to add that information. See examples for expected behavior.

§ 4.8.26.1 Fixed Fields

Field Name	Type	Description
name	`string`	Replaces the name of the element/attribute used for the described schema property. When defined within `items`, it will affect the name of the individual XML elements within the list. When defined alongside `type` being `"array"` (outside the `items`), it will affect the wrapping element if and only if `wrapped` is `true`. If `wrapped` is `false`, it will be ignored.
namespace	`string`	The URI of the namespace definition. Value *MUST* be in the form of a non-relative URI.
prefix	`string`	The prefix to be used for the name.

Field Name	Type	Description
attribute	`boolean`	Declares whether the property definition translates to an attribute instead of an element. Default value is `false`.
wrapped	`boolean`	*MAY* be used only for an array definition. Signifies whether the array is wrapped (for example, `<books><book/><book/></books>`) or unwrapped (`<book/><book/>`). Default value is `false`. The definition takes effect only when defined alongside `type` being `"array"` (outside the `items`).

This object *MAY* be extended with Specification Extensions.

The `namespace` field is intended to match the syntax of XML namespaces, although there are a few caveats:

- Versions 3.1.0, 3.0.3, and earlier of this specification erroneously used the term “absolute URI” instead of “non-relative URI”, so authors using namespaces that include a fragment should check tooling support carefully.
- XML allows but discourages relative URI-references, while this specification outright forbids them.
- XML 1.1 allows IRIs ([RFC3987]) as namespaces, and specifies that namespaces are compared without any encoding or decoding, which means that IRIs encoded to meet this specification’s URI syntax requirement cannot be compared to IRIs as-is.

§ 4.8.26.2 XML Object Examples

Each of the following examples represent the value of the `properties` keyword in a Schema Object that is omitted for brevity. The JSON and YAML representations of the `properties` value are followed by an example XML representation produced for the single property shown.

§ 4.8.26.2.1 No XML Element

Basic string property:

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

```
animals:
  type: string
```

```
<animals>...</animals>
```

Basic string array property (`wrapped` is `false` by default):

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

```
animals:
  type: array
  items:
    type: string
```

```
<animals>...</animals>
<animals>...</animals>
<animals>...</animals>
```

§ 4.8.26.2.2 XML Name Replacement

```
{
  "animals": {
    "type": "string",
    "xml": {
```

```
      "name": "animal"
    }
  }
}
```

```
animals:
  type: string
  xml:
    name: animal
```

```
<animal>...</animal>
```

§ 4.8.26.2.3 XML Attribute, Prefix and Namespace

In this example, a full model definition is shown.

```
{
  "Person": {
    "type": "object",
    "properties": {
      "id": {
        "type": "integer",
        "format": "int32",
        "xml": {
          "attribute": true
        }
      },
      "name": {
        "type": "string",
        "xml": {
          "namespace": "https://example.com/schema/sample",
          "prefix": "sample"
        }
      }
    }
  }
}
```

```
Person:
  type: object
  properties:
    id:
      type: integer
      format: int32
      xml:
```

```
        attribute: true
    name:
      type: string
      xml:
        namespace: https://example.com/schema/sample
        prefix: sample
```

```
<Person id="123">
    <sample:name xmlns:sample="https://example.com/schema/sample">example<
</Person>
```

§ 4.8.26.2.4 XML Arrays

Changing the element names:

```
{
  "animals": {
    "type": "array",
    "items": {
      "type": "string",
      "xml": {
        "name": "animal"
      }
    }
  }
}
```

```
animals:
  type: array
  items:
    type: string
    xml:
      name: animal
```

```
<animal>value</animal>
<animal>value</animal>
```

The external `name` field has no effect on the XML:

```
{
  "animals": {
    "type": "array",
    "items": {
      "type": "string",
      "xml": {
```

```
        "name": "animal"
      }
    },
    "xml": {
      "name": "aliens"
    }
  }
}
```

```
animals:
  type: array
  items:
    type: string
    xml:
      name: animal
  xml:
    name: aliens
```

```
<animal>value</animal>
<animal>value</animal>
```

Even when the array is wrapped, if a name is not explicitly defined, the same name will be used both internally and externally:

```
{
  "animals": {
    "type": "array",
    "items": {
      "type": "string"
    },
    "xml": {
      "wrapped": true
    }
  }
}
```

```
animals:
  type: array
  items:
    type: string
  xml:
    wrapped: true
```

```
<animals>
  <animals>value</animals>
  <animals>value</animals>
</animals>
```

To overcome the naming problem in the example above, the following definition can be used:

```
{
  "animals": {
    "type": "array",
    "items": {
      "type": "string",
      "xml": {
        "name": "animal"
      }
    },
    "xml": {
      "wrapped": true
    }
  }
}
```

```
animals:
  type: array
  items:
    type: string
    xml:
      name: animal
  xml:
    wrapped: true
```

```
<animals>
  <animal>value</animal>
  <animal>value</animal>
</animals>
```

Affecting both internal and external names:

```
{
  "animals": {
    "type": "array",
    "items": {
      "type": "string",
      "xml": {
        "name": "animal"
      }
    },
    "xml": {
      "name": "aliens",
      "wrapped": true
```

```
    }
  }
}
```

```
animals:
  type: array
  items:
    type: string
    xml:
      name: animal
  xml:
    name: aliens
    wrapped: true
```

```
<aliens>
  <animal>value</animal>
  <animal>value</animal>
</aliens>
```

If we change the external element but not the internal ones:

```
{
  "animals": {
    "type": "array",
    "items": {
      "type": "string"
    },
    "xml": {
      "name": "aliens",
      "wrapped": true
    }
  }
}
```

```
animals:
  type: array
  items:
    type: string
  xml:
    name: aliens
    wrapped: true
```

```
<aliens>
  <aliens>value</aliens>
  <aliens>value</aliens>
</aliens>
```

§ 4.8.27 Security Scheme Object

Defines a security scheme that can be used by the operations.

Supported schemes are HTTP authentication, an API key (either as a header, a cookie parameter or as a query parameter), mutual TLS (use of a client certificate), OAuth2's common flows (implicit, password, client credentials and authorization code) as defined in [RFC6749], and [OpenID-Connect-Core]. Please note that as of 2020, the implicit flow is about to be deprecated by OAuth 2.0 Security Best Current Practice. Recommended for most use cases is Authorization Code Grant flow with PKCE.

§ 4.8.27.1 Fixed Fields

Field Name	Type	Applies To	Description
type	`string`	Any	***REQUIRED***. The type of the security scheme. Valid values are `"apiKey"`, `"http"`, `"mutualTLS"`, `"oauth2"`, `"openIdConnect"`.
description	`string`	Any	A description for security scheme. [CommonMark] syntax *MAY* be used for rich text representation.
name	`string`	`apiKey`	***REQUIRED***. The name of the header, query or cookie parameter to be used.
in	`string`	`apiKey`	***REQUIRED***. The location of the API key. Valid values are `"query"`, `"header"`, or `"cookie"`.
scheme	`string`	`http`	***REQUIRED***. The name of the HTTP Authentication scheme to be used in the Authorization header as defined in [RFC7235] Section 5.1. The values used *SHOULD* be registered in the IANA Authentication Scheme registry.

Field Name	Type	Applies To	Description
			The value is case-insensitive, as defined in [RFC7235] Section 2.1.
bearerFormat	`string`	`http` (`"bearer"`)	A hint to the client to identify how the bearer token is formatted. Bearer tokens are usually generated by an authorization server, so this information is primarily for documentation purposes.
flows	OAuth Flows Object	`oauth2`	***REQUIRED***. An object containing configuration information for the flow types supported.
openIdConnectUrl	`string`	`openIdConnect`	***REQUIRED***. Well-known URL to discover the [OpenID-Connect-Discovery] provider metadata.

This object *MAY* be extended with Specification Extensions.

§ 4.8.27.2 Security Scheme Object Examples

§ 4.8.27.2.1 Basic Authentication Example

```
{
  "type": "http",
  "scheme": "basic"
}
```

```
type: http
scheme: basic
```

§ 4.8.27.2.2 API Key Example

```
{
  "type": "apiKey",
  "name": "api-key",
  "in": "header"
}
```

```
type: apiKey
name: api-key
in: header
```

§ 4.8.27.2.3 JWT Bearer Example

```
{
  "type": "http",
  "scheme": "bearer",
  "bearerFormat": "JWT"
}
```

```
type: http
scheme: bearer
bearerFormat: JWT
```

§ 4.8.27.2.4 MutualTLS Example

```
{
  "type": "mutualTLS",
  "description": "Cert must be signed by example.com CA"
}
```

```
type: mutualTLS
description: Cert must be signed by example.com CA
```

§ 4.8.27.2.5 Implicit OAuth2 Example

```json
{
  "type": "oauth2",
  "flows": {
    "implicit": {
      "authorizationUrl": "https://example.com/api/oauth/dialog",
      "scopes": {
        "write:pets": "modify pets in your account",
        "read:pets": "read your pets"
      }
    }
  }
}
```

```yaml
type: oauth2
flows:
  implicit:
    authorizationUrl: https://example.com/api/oauth/dialog
    scopes:
      write:pets: modify pets in your account
      read:pets: read your pets
```

§ 4.8.28 OAuth Flows Object

Allows configuration of the supported OAuth Flows.

§ 4.8.28.1 Fixed Fields

Field Name	Type	Description
implicit	OAuth Flow Object	Configuration for the OAuth Implicit flow
password	OAuth Flow Object	Configuration for the OAuth Resource Owner Password flow

Field Name	Type	Description
clientCredentials	OAuth Flow Object	Configuration for the OAuth Client Credentials flow. Previously called `application` in OpenAPI 2.0.
authorizationCode	OAuth Flow Object	Configuration for the OAuth Authorization Code flow. Previously called `accessCode` in OpenAPI 2.0.

This object *MAY* be extended with Specification Extensions.

§ 4.8.29 OAuth Flow Object

Configuration details for a supported OAuth Flow

§ 4.8.29.1 Fixed Fields

Field Name	Type	Applies To	Description
authorizationUrl	`string`	`oauth2` (`"implicit"`, `"authorizationCode"`)	***REQUIRED***. The authorization URL to be used for this flow. This *MUST* be in the form of a URL. The OAuth2 standard requires the use of TLS.
tokenUrl	`string`	`oauth2` (`"password"`, `"clientCredentials"`, `"authorizationCode"`)	***REQUIRED***. The token URL to be used for this flow. This *MUST* be in the form of a URL. The OAuth2 standard requires the use of TLS.
refreshUrl	`string`	`oauth2`	The URL to be used for obtaining refresh tokens. This *MUST* be in the

Field Name	Type	Applies To	Description
			form of a URL. The OAuth2 standard requires the use of TLS.
scopes	Map[`string`, `string`]	`oauth2`	***REQUIRED***. The available scopes for the OAuth2 security scheme. A map between the scope name and a short description for it. The map *MAY* be empty.

This object *MAY* be extended with Specification Extensions.

§ 4.8.29.2 OAuth Flow Object Example

```
{
  "type": "oauth2",
  "flows": {
    "implicit": {
      "authorizationUrl": "https://example.com/api/oauth/dialog",
      "scopes": {
        "write:pets": "modify pets in your account",
        "read:pets": "read your pets"
      }
    },
    "authorizationCode": {
      "authorizationUrl": "https://example.com/api/oauth/dialog",
      "tokenUrl": "https://example.com/api/oauth/token",
      "scopes": {
        "write:pets": "modify pets in your account",
        "read:pets": "read your pets"
      }
```

```
    }
  }
}
```

```
type: oauth2
flows:
  implicit:
    authorizationUrl: https://example.com/api/oauth/dialog
    scopes:
      write:pets: modify pets in your account
      read:pets: read your pets
  authorizationCode:
    authorizationUrl: https://example.com/api/oauth/dialog
    tokenUrl: https://example.com/api/oauth/token
    scopes:
      write:pets: modify pets in your account
      read:pets: read your pets
```

§ 4.8.30 Security Requirement Object

Lists the required security schemes to execute this operation. The name used for each property *MUST* correspond to a security scheme declared in the Security Schemes under the Components Object.

A Security Requirement Object *MAY* refer to multiple security schemes in which case all schemes *MUST* be satisfied for a request to be authorized. This enables support for scenarios where multiple query parameters or HTTP headers are required to convey security information.

When the `security` field is defined on the OpenAPI Object or Operation Object and contains multiple Security Requirement Objects, only one of the entries in the list needs to be satisfied to authorize the request. This enables support for scenarios where the API allows multiple, independent security schemes.

An empty Security Requirement Object (`{}`) indicates anonymous access is supported.

§ 4.8.30.1 Patterned Fields

Field Pattern	Type	Description
{name}	[`string`]	Each name *MUST* correspond to a security scheme which is declared in the Security Schemes under the Components Object. If

Field Pattern	Type	Description
		the security scheme is of type `"oauth2"` or `"openIdConnect"`, then the value is a list of scope names required for the execution, and the list *MAY* be empty if authorization does not require a specified scope. For other security scheme types, the array *MAY* contain a list of role names which are required for the execution, but are not otherwise defined or exchanged in-band.

§ 4.8.30.2 Security Requirement Object Examples

See also Appendix F: Resolving Security Requirements in a Referenced Document for an example using Security Requirement Objects in multi-document OpenAPI Descriptions.

§ 4.8.30.2.1 Non-OAuth2 Security Requirement

```
{
  "api_key": []
}
```

```
api_key: []
```

§ 4.8.30.2.2 OAuth2 Security Requirement

```
{
  "petstore_auth": ["write:pets", "read:pets"]
}
```

```
petstore_auth:
  - write:pets
  - read:pets
```

§ 4.8.30.2.3 Optional OAuth2 Security

Optional OAuth2 security as would be defined in an OpenAPI Object or an Operation Object:

```
{
  "security": [
    {},
    {
      "petstore_auth": ["write:pets", "read:pets"]
    }
  ]
}
```

```
security:
  - {}
  - petstore_auth:
      - write:pets
      - read:pets
```

§ 4.9 Specification Extensions

While the OpenAPI Specification tries to accommodate most use cases, additional data can be added to extend the specification at certain points.

The extensions properties are implemented as patterned fields that are always prefixed by `x-`.

Field Pattern	Type	Description
^x-	Any	Allows extensions to the OpenAPI Schema. The field name *MUST* begin with `x-`, for example, `x-internal-id`. Field names beginning `x-oai-` and `x-oas-` are reserved for uses defined by the OpenAPI Initiative. The value can be any valid JSON value (`null`, a primitive, an array, or an object.)

The OpenAPI Initiative maintains several extension registries, including registries for individual extension keywords and extension keyword namespaces.

Extensions are one of the best ways to prove the viability of proposed additions to the specification. It is therefore *RECOMMENDED* that implementations be designed for extensibility

to support community experimentation.

Support for any one extension is *OPTIONAL*, and support for one extension does not imply support for others.

4.10 Security Filtering

Some objects in the OpenAPI Specification *MAY* be declared and remain empty, or be completely removed, even though they are inherently the core of the API documentation.

The reasoning is to allow an additional layer of access control over the documentation. While not part of the specification itself, certain libraries *MAY* choose to allow access to parts of the documentation based on some form of authentication/authorization.

Two examples of this:

1. The Paths Object *MAY* be present but empty. It may be counterintuitive, but this may tell the viewer that they got to the right place, but can't access any documentation. They would still have access to at least the Info Object which may contain additional information regarding authentication.
2. The Path Item Object *MAY* be empty. In this case, the viewer will be aware that the path exists, but will not be able to see any of its operations or parameters. This is different from hiding the path itself from the Paths Object, because the user will be aware of its existence. This allows the documentation provider to finely control what the viewer can see.

5. Security Considerations

5.1 OpenAPI Description Formats

OpenAPI Descriptions use a combination of JSON, YAML, and JSON Schema, and therefore share their security considerations:

- JSON
- YAML
- JSON Schema Core
- JSON Schema Validation

5.2 Tooling and Usage Scenarios

In addition, OpenAPI Descriptions are processed by a wide variety of tooling for numerous different purposes, such as client code generation, documentation generation, server side routing, and API testing. OpenAPI Description authors must consider the risks of the scenarios where the OpenAPI Description may be used.

5.3 Security Schemes

An OpenAPI Description describes the security schemes used to protect the resources it defines. The security schemes available offer varying degrees of protection. Factors such as the sensitivity of the data and the potential impact of a security breach should guide the selection of security schemes for the API resources. Some security schemes, such as basic auth and OAuth Implicit flow, are supported for compatibility with existing APIs. However, their inclusion in OpenAPI does not constitute an endorsement of their use, particularly for highly sensitive data or operations.

5.4 Handling External Resources

OpenAPI Descriptions may contain references to external resources that may be dereferenced automatically by consuming tools. External resources may be hosted on different domains that may be untrusted.

5.5 Handling Reference Cycles

References in an OpenAPI Description may cause a cycle. Tooling must detect and handle cycles to prevent resource exhaustion.

5.6 Markdown and HTML Sanitization

Certain fields allow the use of Markdown which can contain HTML including script. It is the responsibility of tooling to appropriately sanitize the Markdown.

A. Appendix A: Revision History

Version	Date	Notes
3.1.1	2024-10-24	Patch release of the OpenAPI Specification 3.1.1
3.1.0	2021-02-15	Release of the OpenAPI Specification 3.1.0
3.1.0-rc1	2020-10-08	rc1 of the 3.1 specification
3.1.0-rc0	2020-06-18	rc0 of the 3.1 specification
3.0.4	2024-10-24	Patch release of the OpenAPI Specification 3.0.4
3.0.3	2020-02-20	Patch release of the OpenAPI Specification 3.0.3
3.0.2	2018-10-08	Patch release of the OpenAPI Specification 3.0.2
3.0.1	2017-12-06	Patch release of the OpenAPI Specification 3.0.1
3.0.0	2017-07-26	Release of the OpenAPI Specification 3.0.0
3.0.0-rc2	2017-06-16	rc2 of the 3.0 specification
3.0.0-rc1	2017-04-27	rc1 of the 3.0 specification
3.0.0-rc0	2017-02-28	Implementer's Draft of the 3.0 specification
2.0	2015-12-31	Donation of Swagger 2.0 to the OpenAPI Initiative
2.0	2014-09-08	Release of Swagger 2.0
1.2	2014-03-14	Initial release of the formal document.
1.1	2012-08-22	Release of Swagger 1.1
1.0	2011-08-10	First release of the Swagger Specification

B. Appendix B: Data Type Conversion

Serializing typed data to plain text, which can occur in `text/plain` message bodies or `multipart` parts, as well as in the `application/x-www-form-urlencoded` format in either URL query strings or message bodies, involves significant implementation- or application-defined behavior.

Schema Objects validate data based on the JSON Schema data model, which only recognizes four primitive data types: strings (which are only broadly interoperable as UTF-8), numbers, booleans, and `null`. Notably, integers are not a distinct type from other numbers, with `type: "integer"` being a convenience defined mathematically, rather than based on the presence or absence of a decimal point in any string representation.

The Parameter Object, Header Object, and Encoding Object offer features to control how to arrange values from array or object types. They can also be used to control how strings are further encoded to avoid reserved or illegal characters. However, there is no general-purpose specification for converting schema-validated non-UTF-8 primitive data types (or entire arrays or objects) to strings.

Two cases do offer standards-based guidance:

- [RFC3987] Section 3.1 provides guidance for converting non-Unicode strings to UTF-8, particularly in the context of URIs (and by extension, the form media types which use the same encoding rules)
- [RFC6570] Section 2.3 specifies which values, including but not limited to `null`, are considered *undefined* and therefore treated specially in the expansion process when serializing based on that specification

Implementations of RFC6570 often have their own conventions for converting non-string values, but these are implementation-specific and not defined by the RFC itself. This is one reason for the OpenAPI Specification to leave these conversions as implementation-defined: It allows using RFC6570 implementations regardless of how they choose to perform the conversions.

To control the serialization of numbers, booleans, and `null` (or other values RFC6570 deems to be undefined) more precisely, schemas can be defined as `type: "string"` and constrained using `pattern`, `enum`, `format`, and other keywords to communicate how applications must pre-convert their data prior to schema validation. The resulting strings would not require any further type conversion.

The `format` keyword can assist in serialization. Some formats (such as `date-time`) are unambiguous, while others (such as `decimal` in the Format Registry) are less clear. However, care must be taken with `format` to ensure that the specific formats are supported by all relevant tools as unrecognized formats are ignored.

Requiring input as pre-formatted, schema-validated strings also improves round-trip interoperability as not all programming languages and environments support the same data types.

C. Appendix C: Using RFC6570-Based Serialization

Serialization is defined in terms of [RFC6570] URI Templates in three scenarios:

Object	Condition
Parameter Object	When `schema` is present
Header Object	When `schema` is present
Encoding Object	When encoding for `application/x-www-form-urlencoded` and any of `style`, `explode`, or `allowReserved` are used

Implementations of this specification *MAY* use an implementation of RFC6570 to perform variable expansion, however, some caveats apply.

Note that when using `style: "form"` RFC6570 expansion to produce an `application/x-www-form-urlencoded` HTTP message body, it is necessary to remove the `?` prefix that is produced to satisfy the URI query string syntax.

When using `style` and similar keywords to produce a `multipart/form-data` body, the query string names are placed in the `name` parameter of the `Content-Disposition` part header, and the values are placed in the corresponding part body; the `?`, `=`, and `&` characters are not used. Note that while [RFC7578] allows using [RFC3986] percent-encoding in "file names", it does not otherwise address the use of percent-encoding within the format. RFC7578 discusses character set and encoding issues for `multipart/form-data` in detail, and it is *RECOMMENDED* that OpenAPI Description authors read this guidance carefully before deciding to use RFC6570-based serialization with this media type.

Note also that not all RFC6570 implementations support all four levels of operators, all of which are needed to fully support the OpenAPI Specification's usage. Using an implementation with a lower level of support will require additional manual construction of URI Templates to work around the limitations.

C.1 Equivalences Between Fields and RFC6570 Operators

Certain field values translate to RFC6570 operators (or lack thereof):

field	value	equivalent
style	`"simple"`	*n/a*
style	`"matrix"`	`;` prefix operator
style	`"label"`	`.` prefix operator
style	`"form"`	`?` prefix operator
allowReserved	`false`	*n/a*
allowReserved	`true`	`+` prefix operator
explode	`false`	*n/a*
explode	`true`	`*` modifier suffix

Multiple `style: "form"` parameters are equivalent to a single RFC6570 variable list using the `?` prefix operator:

```yaml
parameters:
- name: foo
  in: query
  schema:
    type: object
  explode: true
- name: bar
  in: query
  schema:
    type: string
```

This example is equivalent to RFC6570's `{?foo*,bar}`, and **NOT** `{?foo*}{&bar}`. The latter is problematic because if `foo` is not defined, the result will be an invalid URI. The `&` prefix operator has no equivalent in the Parameter Object.

Note that RFC6570 does not specify behavior for compound values beyond the single level addressed by `explode`. The result of using objects or arrays where no behavior is clearly specified for them is implementation-defined.

§ C.2 Delimiters in Parameter Values

Delimiters used by RFC6570 expansion, such as the `,` used to join arrays or object values with `style: "simple"`, are all automatically percent-encoded as long as `allowReserved` is `false`.

Note that since RFC6570 does not define a way to parse variables based on a URI Template, users must take care to first split values by delimiter before percent-decoding values that might contain the delimiter character.

When `allowReserved` is `true`, both percent-encoding (prior to joining values with a delimiter) and percent-decoding (after splitting on the delimiter) must be done manually at the correct time.

See Appendix E for additional guidance on handling delimiters for `style` values with no RFC6570 equivalent that already need to be percent-encoded when used as delimiters.

C.3 Non-RFC6570 Field Values and Combinations

Configurations with no direct [RFC6570] equivalent *SHOULD* also be handled according to RFC6570. Implementations *MAY* create a properly delimited URI Template with variables for individual names and values using RFC6570 regular or reserved expansion (based on `allowReserved`).

This includes:

- the styles `pipeDelimited`, `spaceDelimited`, and `deepObject`, which have no equivalents at all
- the combination of the style `form` with `allowReserved: true`, which is not allowed because only one prefix operator can be used at a time
- any parameter name that is not a legal RFC6570 variable name

The Parameter Object's `name` field has a much more permissive syntax than RFC6570 variable name syntax. A parameter name that includes characters outside of the allowed RFC6570 variable character set *MUST* be percent-encoded before it can be used in a URI Template.

C.4 Examples

Let's say we want to use the following data in a form query string, where `formulas` is exploded, and `words` is not:

```
formulas:
  a: x+y
  b: x/y
  c: x^y
words:
```

```
- math
- is
- fun
```

§ C.4.1 RFC6570-Equivalent Expansion

This array of Parameter Objects uses regular `style: "form"` expansion, fully supported by [RFC6570]:

```
parameters:
- name: formulas
  in: query
  schema:
    type: object
    additionalProperties:
      type: string
  explode: true
- name: words
  in: query
  schema:
    type: array
    items:
      type: string
```

This translates to the following URI Template:

```
{?formulas*,words}
```

when expanded with the data given earlier, we get:

```
?a=x%2By&b=x%2Fy&c=x%5Ey&words=math,is,fun
```

§ C.4.2 Expansion with Non-RFC6570-Supported Options

But now let's say that (for some reason), we really want that `/` in the `b` formula to show up as-is in the query string, and we want our words to be space-separated like in a written phrase. To do that, we'll add `allowReserved: true` to `formulas`, and change to `style: "spaceDelimited"` for `words`:

```
parameters:
- name: formulas
  in: query
```

```
  schema:
    type: object
    additionalProperties:
      type: string
  explode: true
  allowReserved: true
- name: words
  in: query
  style: spaceDelimited
  explode: false
  schema:
    type: array
    items:
      type: string
```

We can't combine the `?` and `+` RFC6570 prefixes, and there's no way with RFC6570 to replace the `,` separator with a space character. So we need to restructure the data to fit a manually constructed URI Template that passes all of the pieces through the right sort of expansion.

Here is one such template, using a made-up convention of `words.0` for the first entry in the words value, `words.1` for the second, and `words.2` for the third:

```
?a={+a}&b={+b}&c={+c}&words={words.0} {words.1} {words.2}
```

RFC6570 mentions the use of `.` "to indicate name hierarchy in substructures," but does not define any specific naming convention or behavior for it. Since the `.` usage is not automatic, we'll need to construct an appropriate input structure for this new template.

We'll also need to pre-process the values for `formulas` because while `/` and most other reserved characters are allowed in the query string by RFC3986, `[`, `]`, and `#` are not, and `&`, `=`, and `+` all have special behavior in the `application/x-www-form-urlencoded` format, which is what we are using in the query string.

Setting `allowReserved: true` does *not* make reserved characters that are not allowed in URIs allowed, it just allows them to be *passed through expansion unchanged*. Therefore, any tooling still needs to percent-encode those characters because reserved expansion will not do it, but it *will* leave the percent-encoded triples unchanged. See also Appendix E for further guidance on percent-encoding and form media types, including guidance on handling the delimiter characters for `spaceDelimited`, `pipeDelimited`, and `deepObject` in parameter names and values.

So here is our data structure that arranges the names and values to suit the template above, where values for `formulas` have `[]#&=+` pre-percent encoded (although only `+` appears in this example):

```
a: x%2By
b: x/y
c: x^y
```

```
words.0: math
words.1: is
words.2: fun
```

Expanding our manually assembled template with our restructured data yields the following query string:

```
?a=x%2By&b=x/y&c=x%5Ey&words=math%20is%20fun
```

The `/` and the pre-percent-encoded `%2B` have been left alone, but the disallowed `^` character (inside a value) and space characters (in the template but outside of the expanded variables) were percent-encoded.

§ C.4.3 Undefined Values and Manual URI Template Construction

Care must be taken when manually constructing templates to handle the values that RFC6570 considers to be *undefined* correctly:

```
formulas: {}
words:
- hello
- world
```

Using this data with our original RFC6570-friendly URI Template, `{?formulas*,words}`, produces the following:

```
?words=hello,world
```

This means that the manually constructed URI Template and restructured data need to leave out the `formulas` object entirely so that the `words` parameter is the first and only parameter in the query string.

Restructured data:

```
words.0: hello
words.1: world
```

Manually constructed URI Template:

```
?words={words.0} {words.1}
```

Result:

```
?words=hello%20world
```

C.4.4 Illegal Variable Names as Parameter Names

In this example, the heart emoji is not legal in URI Template names (or URIs):

```yaml
parameters:
- name: ❤️
  in: query
  schema:
    type: string
```

We can't just pass `❤️: "love!"` to an RFC6570 implementation. Instead, we have to pre-percent-encode the name (which is a six-octet UTF-8 sequence) in both the data and the URI Template:

```
"%E2%9D%A4%EF%B8%8F": love!
```

```
{?%E2%9D%A4%EF%B8%8F}
```

This will expand to the result:

```
?%E2%9D%A4%EF%B8%8F=love%21
```

D. Appendix D: Serializing Headers and Cookies

[RFC6570]'s percent-encoding behavior is not always appropriate for `in: "header"` and `in: "cookie"` parameters. In many cases, it is more appropriate to use `content` with a media type such as `text/plain` and require the application to assemble the correct string.

For both [RFC6265] cookies and HTTP headers using the [RFC8941] structured fields syntax, non-ASCII content is handled using base64 encoding (`contentEncoding: "base64"`). Note that the standard base64-encoding alphabet includes non-URL-safe characters that are percent-encoded by RFC6570 expansion; serializing values through both encodings is *NOT RECOMMENDED*. While `contentEncoding` also supports the `base64url` encoding, which is URL-safe, the header and cookie RFCs do not mention this encoding.

Most HTTP headers predate the structured field syntax, and a comprehensive assessment of their syntax and encoding rules is well beyond the scope of this specification. While [RFC8187] recommends percent-encoding HTTP (header or trailer) field parameters, these parameters appear after a `;` character. With `style: "simple"`, that delimiter would itself be percent-encoded, violating the general HTTP field syntax.

Using `style: "form"` with `in: "cookie"` is ambiguous for a single value, and incorrect for multiple values. This is true whether the multiple values are the result of using `explode: true` or not.

This style is specified to be equivalent to RFC6570 form expansion which includes the `?` character (see Appendix C for more details), which is not part of the cookie syntax. However, examples of this style in past versions of this specification have not included the `?` prefix, suggesting that the comparison is not exact. Because implementations that rely on an RFC6570 implementation and those that perform custom serialization based on the style example will produce different results, it is implementation-defined as to which of the two results is correct.

For multiple values, `style: "form"` is always incorrect as name=value pairs in cookies are delimited by `;` (a semicolon followed by a space character) rather than `&`.

E. Appendix E: Percent-Encoding and Form Media Types

NOTE: *In this section, the `application/x-www-form-urlencoded` and `multipart/form-data` media types are abbreviated as `form-urlencoded` and `form-data`, respectively, for readability.*

Percent-encoding is used in URIs and media types that derive their syntax from URIs. This process is concerned with three sets of characters, the names of which vary among specifications but are defined as follows for the purposes of this section:

- *unreserved* characters do not need to be percent-encoded; while it is safe to percent-encode them, doing so produces a URI that is not normalized
- *reserved* characters either have special behavior in the URI syntax (such as delimiting components) or are reserved for other specifications that need to define special behavior (e.g. `form-urlencoded` defines special behavior for `=`, `&`, and `+`)
- *unsafe* characters are known to cause problems when parsing URIs in certain environments

Unless otherwise specified, this section uses RFC3986's definition of reserved and unreserved, and defines the unsafe set as all characters not included in either of those sets.

E.1 Percent-Encoding and `form-urlencoded`

Each URI component (such as the query string) considers some of the reserved characters to be unsafe, either because they serve as delimiters between the components (e.g. `#`), or (in the case of `[` and `]`) were historically considered globally unsafe but were later given reserved status for limited purposes.

Reserved characters with no special meaning defined within a component can be left un-percent encoded. However, other specifications can define special meanings, requiring percent-encoding

for those characters outside of the additional special meanings.

The `form-urlencoded` media type defines special meanings for `=` and `&` as delimiters, and `+` as the replacement for the space character (instead of its percent-encoded form of `%20`). This means that while these three characters are reserved-but-allowed in query strings by RFC3986, they must be percent-encoded in `form-urlencoded` query strings except when used for their `form-urlencoded` purposes; see Appendix C for an example of handling `+` in form values.

§ E.2 Percent-Encoding and `form-data`

[RFC7578] Section 2 suggests RFC3986-based percent-encoding as a mechanism to keep text-based per-part header data such as file names within the ASCII character set. This suggestion was not part of older (pre-2015) specifications for `form-data`, so care must be taken to ensure interoperability.

The `form-data` media type allows arbitrary text or binary data in its parts, so percent-encoding is not needed and is likely to cause interoperability problems unless the `Content-Type` of the part is defined to require it.

§ E.3 Generating and Validating URIs and `form-urlencoded` Strings

URI percent encoding and the `form-urlencoded` media type have complex specification histories spanning multiple revisions and, in some cases, conflicting claims of ownership by different standards bodies. Unfortunately, these specifications each define slightly different percent-encoding rules, which need to be taken into account if the URIs or `form-urlencoded` message bodies will be subject to strict validation. (Note that many URI parsers do not perform validation by default.)

This specification normatively cites the following relevant standards:

Specification	Date	OAS Usage	Percent-Encoding	Notes
[RFC3986]	01/2005	URI/URL syntax	[RFC3986]	obsoletes [RFC1738], [RFC2396]
[RFC6570]	03/2012	style-based serialization	[RFC3986]	does not use `+` for `form-urlencoded`

Specification	Date	OAS Usage	Percent-Encoding	Notes
[RFC1866] Section 8.2.1	11/1995	content-based serialization	[RFC1738]	obsoleted by [HTML401] Section 17.13.4.1, [URL] Section 5

Style-based serialization is used in the Parameter Object when `schema` is present, and in the Encoding Object when at least one of `style`, `explode`, or `allowReserved` is present. See Appendix C for more details of RFC6570's two different approaches to percent-encoding, including an example involving `+`.

Content-based serialization is defined by the Media Type Object, and used with the Parameter Object when the `content` field is present, and with the Encoding Object based on the `contentType` field when the fields `style`, `explode`, and `allowReserved` are absent. Each part is encoded based on the media type (e.g. `text/plain` or `application/json`), and must then be percent-encoded for use in a `form-urlencoded` string.

Note that content-based serialization for `form-data` does not expect or require percent-encoding in the data, only in per-part header values.

§ E.3.1 Interoperability with Historical Specifications

In most cases, generating query strings in strict compliance with [RFC3986] is sufficient to pass validation (including JSON Schema's `format: "uri"` and `format: "uri-reference"`), but some `form-urlencoded` implementations still expect the slightly more restrictive [RFC1738] rules to be used.

Since all RFC1738-compliant URIs are compliant with RFC3986, applications needing to ensure historical interoperability *SHOULD* use RFC1738's rules.

§ E.3.2 Interoperability with Web Browser Environments

WHATWG is a web browser-oriented standards group that has defined a "URL Living Standard" for parsing and serializing URLs in a browser context, including parsing and serializing `form-urlencoded` data. WHATWG's percent-encoding rules for query strings are different depending on whether the query string is being treated as `form-urlencoded` (where it requires more percent-encoding than [RFC1738]) or as part of the generic syntax, where it allows characters that [RFC3986] forbids.

Implementations needing maximum compatibility with web browsers *SHOULD* use WHATWG's `form-urlencoded` percent-encoding rules. However, they *SHOULD NOT* rely on WHATWG's less stringent generic query string rules, as the resulting URLs would fail RFC3986 validation, including JSON Schema's `format: uri` and `format: uri-reference`.

§ E.4 Decoding URIs and `form-urlencoded` Strings

The percent-decoding algorithm does not care which characters were or were not percent-decoded, which means that URIs percent-encoded according to any specification will be decoded correctly.

Similarly, all `form-urlencoded` decoding algorithms simply add `+`-for-space handling to the percent-decoding algorithm, and will work regardless of the encoding specification used.

However, care must be taken to use `form-urlencoded` decoding if `+` represents a space, and to use regular percent-decoding if `+` represents itself as a literal value.

§ E.5 Percent-Encoding and Illegal or Reserved Delimiters

The `[`, `]`, `|`, and space characters, which are used as delimiters for the `deepObject`, `pipeDelimited`, and `spaceDelimited` styles, respectively, all *MUST* be percent-encoded to comply with [RFC3986]. This requires users to pre-encode the character(s) in some other way in parameter names and values to distinguish them from the delimiter usage when using one of these styles.

The space character is always illegal and encoded in some way by all implementations of all versions of the relevant standards. While one could use the `form-urlencoded` convention of `+` to distinguish spaces in parameter names and values from `spaceDelimited` delimiters encoded as `%20`, the specifications define the decoding as a single pass, making it impossible to distinguish the different usages in the decoded result.

Some environments use `[`, `]`, and possibly `|` unencoded in query strings without apparent difficulties, and WHATWG's generic query string rules do not require percent-encoding them. Code that relies on leaving these delimiters unencoded, while using regular percent-encoding for them within names and values, is not guaranteed to be interoperable across all implementations.

For maximum interoperability, it is *RECOMMENDED* to either define and document an additional escape convention while percent-encoding the delimiters for these styles, or to avoid these styles entirely. The exact method of additional encoding/escaping is left to the API designer, and is expected to be performed before serialization and encoding described in this specification, and

reversed after this specification's encoding and serialization steps are reversed. This keeps it outside of the processes governed by this specification.

F. Appendix F: Resolving Security Requirements in a Referenced Document

This appendix shows how to retrieve an HTTP-accessible multi-document OpenAPI Description (OAD) and resolve a Security Requirement Object in the referenced (non-entry) document. See Resolving Implicit Connections for more information.

First, the entry document is where parsing begins. It defines the `MySecurity` security scheme to be JWT-based, and it defines a Path Item as a reference to a component in another document:

```
GET /api/description/openapi HTTP/1.1
Host: www.example.com
Accept: application/openapi+json
```

```
"components": {
  "securitySchemes": {
    "MySecurity": {
      "type": "http",
      "scheme": "bearer",
      "bearerFormat": "JWT"
    }
  }
},
"paths": {
  "/foo": {
    "$ref": "other#/components/pathItems/Foo"
  }
}
```

```
GET /api/description/openapi HTTP/1.1
Host: www.example.com
Accept: application/openapi+yaml
```

```
components:
  securitySchemes:
    MySecurity:
      type: http
      scheme: bearer
      bearerFormat: JWT
paths:
  /foo:
    $ref: 'other#/components/pathItems/Foo'
```

This entry document references another document, `other`, without using a file extension. This gives the client the flexibility to choose an acceptable format on a resource-by-resource basis, assuming both representations are available:

```
GET /api/description/other HTTP/1.1
Host: www.example.com
Accept: application/openapi+json
```

```
"components": {
  "securitySchemes": {
    "MySecurity": {
      "type": "http",
      "scheme": "basic"
    }
  },
  "pathItems": {
    "Foo": {
      "get": {
        "security": [
          "MySecurity": []
        ]
      }
    }
  }
}
```

```
GET /api/description/other HTTP/1.1
Host: www.example.com
Accept: application/openapi+yaml
```

```
components:
  securitySchemes:
    MySecurity:
      type: http
      scheme: basic
  pathItems:
    Foo:
      get:
        security:
          - MySecurity: []
```

In the `other` document, the referenced path item has a Security Requirement for a Security Scheme, `MySecurity`. The same Security Scheme exists in the original entry document. As outlined in Resolving Implicit Connections, `MySecurity` is resolved with an implementation-defined behavior. However, documented in that section, it is *RECOMMENDED* that tools resolve component names from the entry document. As with all implementation-defined behavior, it is important to check tool documentation to determine which behavior is supported.

G. References

G.1 Normative references

[ABNF]
Augmented BNF for Syntax Specifications: ABNF. D. Crocker, Ed.; P. Overell. IETF. January 2008. Internet Standard. URL: https://www.rfc-editor.org/rfc/rfc5234

[CommonMark]
CommonMark Spec. URL: https://spec.commonmark.org/

[CommonMark-0.27]
CommonMark Spec, Version 0.27. John MacFarlane. 18 November 2016. URL: https://spec.commonmark.org/0.27/

[HTML401]
HTML 4.01 Specification. Dave Raggett; Arnaud Le Hors; Ian Jacobs. W3C. 27 March 2018. W3C Recommendation. URL: https://www.w3.org/TR/html401/

[IANA-HTTP-AUTHSCHEMES]
Hypertext Transfer Protocol (HTTP) Authentication Scheme Registry. IANA. URL: https://www.iana.org/assignments/http-authschemes/

[IANA-HTTP-STATUS-CODES]
Hypertext Transfer Protocol (HTTP) Status Code Registry. IANA. URL: https://www.iana.org/assignments/http-status-codes/

[JSON-Schema-2020-12]
JSON Schema: A Media Type for Describing JSON Documents. Draft 2020-12. Austin Wright; Henry Andrews; Ben Hutton; Greg Dennis. Internet Engineering Task Force (IETF). 10 June 2022. Internet-Draft. URL: https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-01

[JSON-Schema-Validation-2020-12]
JSON Schema Validation: A Vocabulary for Structural Validation of JSON. Draft 2020-12. Austin Wright; Henry Andrews; Ben Hutton. Internet Engineering Task Force (IETF). 10 June 2022. Internet-Draft. URL: https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-validation-01

[OpenAPI-Registry]
OpenAPI Initiative Registry. OpenAPI Initiative. URL: https://spec.openapis.org/registry/index.html

[OpenID-Connect-Core]
OpenID Connect Core 1.0 incorporating errata set 2. N. Sakimura; J. Bradley; M. Jones; B. de Medeiros; C. Mortimore. OpenID Foundation. 15 December 2023. Final. URL:

https://openid.net/specs/openid-connect-core-1_0.html

[OpenID-Connect-Discovery]

OpenID Connect Discovery 1.0 incorporating errata set 2. N. Sakimura; J. Bradley; M. Jones; E. Jay. OpenID Foundation. 15 December 2023. Final. URL: https://openid.net/specs/openid-connect-discovery-1_0.html

[RFC1738]

Uniform Resource Locators (URL). T. Berners-Lee; L. Masinter; M. McCahill. IETF. December 1994. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc1738

[RFC1866]

Hypertext Markup Language - 2.0. T. Berners-Lee; D. Connolly. IETF. November 1995. Historic. URL: https://www.rfc-editor.org/rfc/rfc1866

[RFC2046]

Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. N. Freed; N. Borenstein. IETF. November 1996. Draft Standard. URL: https://www.rfc-editor.org/rfc/rfc2046

[RFC2119]

Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc2119

[RFC2396]

Uniform Resource Identifiers (URI): Generic Syntax. T. Berners-Lee; R. Fielding; L. Masinter. IETF. August 1998. Draft Standard. URL: https://www.rfc-editor.org/rfc/rfc2396

[RFC3986]

Uniform Resource Identifier (URI): Generic Syntax. T. Berners-Lee; R. Fielding; L. Masinter. IETF. January 2005. Internet Standard. URL: https://www.rfc-editor.org/rfc/rfc3986

[RFC3987]

Internationalized Resource Identifiers (IRIs). M. Duerst; M. Suignard. IETF. January 2005. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc3987

[RFC4648]

The Base16, Base32, and Base64 Data Encodings. S. Josefsson. IETF. October 2006. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc4648

[RFC6265]

HTTP State Management Mechanism. A. Barth. IETF. April 2011. Proposed Standard. URL: https://httpwg.org/specs/rfc6265.html

[RFC6570]

URI Template. J. Gregorio; R. Fielding; M. Hadley; M. Nottingham; D. Orchard. IETF. March 2012. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc6570

[RFC6749]

The OAuth 2.0 Authorization Framework. D. Hardt, Ed. IETF. October 2012. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc6749

[RFC6838]

Media Type Specifications and Registration Procedures. N. Freed; J. Klensin; T. Hansen. IETF. January 2013. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc6838

[RFC6901]

JavaScript Object Notation (JSON) Pointer. P. Bryan, Ed.; K. Zyp; M. Nottingham, Ed. IETF. April 2013. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc6901

[RFC7159]

The JavaScript Object Notation (JSON) Data Interchange Format. T. Bray, Ed. IETF. March 2014. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc7159

[RFC7230]

Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. R. Fielding, Ed.; J. Reschke, Ed. IETF. June 2014. Proposed Standard. URL: https://httpwg.org/specs/rfc7230.html

[RFC7231]

Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. R. Fielding, Ed.; J. Reschke, Ed. IETF. June 2014. Proposed Standard. URL: https://httpwg.org/specs/rfc7231.html

[RFC7235]

Hypertext Transfer Protocol (HTTP/1.1): Authentication. R. Fielding, Ed.; J. Reschke, Ed. IETF. June 2014. Proposed Standard. URL: https://httpwg.org/specs/rfc7235.html

[RFC7578]

Returning Values from Forms: multipart/form-data. L. Masinter. IETF. July 2015. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc7578

[RFC8174]

Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. B. Leiba. IETF. May 2017. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc8174

[RFC8187]

Indicating Character Encoding and Language for HTTP Header Field Parameters. J. Reschke. IETF. September 2017. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc8187

[RFC8941]

Structured Field Values for HTTP. M. Nottingham; P-H. Kamp. IETF. February 2021. Proposed Standard. URL: https://httpwg.org/specs/rfc8941.html

[SPDX-Licenses]

SPDX License List. Linux Foundation. URL: https://spdx.org/licenses/

[URL]

URL Standard. Anne van Kesteren. WHATWG. Living Standard. URL: https://url.spec.whatwg.org/

[xml-names11]

Namespaces in XML 1.1 (Second Edition). Tim Bray; Dave Hollander; Andrew Layman; Richard Tobin et al. W3C. 16 August 2006. W3C Recommendation. URL: https://www.w3.org/TR/xml-names11/

[YAML]

YAML Ain't Markup Language (YAML™) Version 1.2. Oren Ben-Kiki; Clark Evans; Ingy döt Net. 1 October 2009. URL: http://yaml.org/spec/1.2/spec.html

§ G.2 Informative references

[OpenAPI-Learn]

OpenAPI - Getting started, and the specification explained. OpenAPI Initiative. URL: https://learn.openapis.org/

↥