https://web.archive.org/web/20230524173852/https://jsonapi.org/format/#query-parameters May 30, 2025 at 12:16 PM CDT

https://jsonapi.org/format/ Go APR MAY JUN
1,425 captures 24
7 Jun 2013 - 27 May 2025 2022 2023 2024 About this capture

Latest Version (v1.1)

Status
Introduction
Semantics
Conventions
The JSON:API Media Type
  Media Type Parameters
  Extensions
  Profiles
  Rules for Media Type Parameters
    Rules for Extensions
    Rules for Profiles
Content Negotiation
  Universal Responsibilities
  Client Responsibilities
  Server Responsibilities
Document Structure
  Top Level
  Resource Objects
    Identification
    Fields
    Resource Links
  Resource Identifier Objects
  Compound Documents
  Meta Information
  Links
    Link objects
  JSON:API Object
  Member Names
    Allowed Characters
    Reserved Characters
Fetching Relationships
    Responses
  Inclusion of Related Resources
  Sparse Fieldsets
  Sorting
  Pagination
  Filtering
Creating, Updating and Deleting Resources
  Creating Resources
    Client-Generated IDs
    Responses
  Updating Resources

Latest Specification (v1.1)

1 Status

This page presents the latest published version of JSON:API, which is currently version 1.1. New versions of JSON:API **will always be backwards compatible** using a *never remove, only add* strategy. Additions can be proposed in our discussion forum.

If you catch an error in the specification's text, or if you write an implementation, please let us know by opening an issue or pull request at our GitHub repository.

2 Introduction

JSON:API is a specification for how a client should request that resources be fetched or modified, and how a server should respond to those requests. JSON:API can be easily extended with extensions and profiles.

JSON:API is designed to minimize both the number of requests and the amount of data transmitted between clients and servers. This efficiency is achieved without compromising readability, flexibility, or discoverability.

JSON:API requires use of the JSON:API media type (`application/vnd.api+json`) for exchanging data.

3 Semantics

All document members, query parameters, and processing rules defined by this specification are collectively called "specification semantics".

Certain document members, query parameters, and processing rules are reserved for implementors to define at their discretion. These are called "implementation semantics".

All other semantics are reserved for potential future use by this specification.

4 Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and

Updating a Resource's Attributes
Updating a Resource's Relationships
Responses
Updating Relationships
Updating To-One Relationships
Updating To-Many Relationships
Responses
Deleting Resources
Responses
Query Parameters
Query Parameter Families
Extension-Specific Query Parameters
Implementation-Specific Query Parameters
Errors
Processing Errors
Error Objects
Appendix
Query Parameters Details
Parsing/Serialization
Square Brackets in Parameter Names

"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.

5 The JSON:API Media Type

The JSON:API media type is `application/vnd.api+json`.

5.1 Media Type Parameters

The JSON:API specification supports two media type parameters: `ext` and `profile`, which are used to specify extensions and profiles, respectively.

> Note: A media type parameter is an extra piece of information that can accompany a media type. For example, in the header `Content-Type: text/html; charset="utf-8"`, the media type is `text/html` and `charset` is a parameter.

5.2 Extensions

Extensions provide a means to "extend" the base specification by defining additional specification semantics.

Extensions cannot alter or remove specification semantics, nor can they specify implementation semantics.

5.3 Profiles

Profiles provide a means to share a particular usage of the specification among implementations.

Profiles can specify implementation semantics, but cannot alter, add to, or remove specification semantics.

5.4 Rules for Media Type Parameters

The JSON:API media type **MUST NOT** be specified with any media type parameters other than `ext` and `profile`. The `ext` parameter is used to support extensions and the `profile` parameter is used to support profiles.

Extensions and profiles are each uniquely identified by a URI. Visiting an extension's or a profile's URI **SHOULD** return documentation that describes its usage. The values of the `ext` and `profile` parameters **MUST** equal a space-separated (U+0020 SPACE, " ") list of extension or profile URIs, respectively.

> Note: When serializing the `ext` or `profile` media type parameters, the HTTP specification requires that parameter values be surrounded by quotation marks (U+0022 QUOTATION MARK, """).

5.4.1 Rules for Extensions

An extension **MAY** impose additional processing rules or further restrictions and it **MAY** define new object members as described below.

An extension **MUST NOT** lessen or remove any processing rules, restrictions or object member requirements defined in this specification or other extensions.

An extension **MAY** define new members within the document structure defined by this specification. The rules for extension member names are covered below.

An extension **MAY** define new query parameters. The rules for extension-defined query parameters are covered below.

When an extension defines new query parameters or document members, the extension **MUST** define a namespace to guarantee that extensions will never conflict with current or future versions of this specification. A namespace **MUST** meet all of the following conditions:

- A namespace **MUST** contain at least one character.
- A namespace **MUST** contain only these characters:
  - U+0061 to U+007A, “a-z”
  - U+0041 to U+005A, “A-Z”
  - U+0030 to U+0039, “0-9”

An extension **MUST NOT** define more than one namespace. The namespace used for all query parameters and document members **MUST** be the same for any given extension.

In the following example, an extension with the namespace `version` has specified a resource object member `version:id` to support per-resource versioning. This member might appear as follows:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json;
ext="https://jsonapi.org/ext/version"

// ...
{
  "type": "articles",
  "id": "1",
  "version:id": "42",
  "attributes": {
    "title": "Rails is Omakase"
  }
}
// ...
```

5.4.2 Rules for Profiles

The rules for profile usage are dictated by RFC 6906.

A profile **MAY** define document members and processing rules that are reserved for implementors.

A profile **MUST NOT** define any query parameters.

A profile **MUST NOT** alter or remove processing rules that have been defined by this specification or by an extension. However, a profile **MAY** define processing rules for query parameters whose processing rules have been reserved for implementors to define at their discretion.

For example, a profile could define rules for interpreting the `filter` query parameter, but it could not specify that relationship names in the `include` query parameter are space-separated instead of dot-separated.

Unlike extensions, profiles do not need to define a namespace for document members because profiles cannot define specification semantics and thus cannot conflict with current or future versions of this specification. However, it is possible for profiles to conflict with other profiles. Therefore, it is the responsibility of implementors to ensure that they do not support conflicting profiles.

In the following example, a profile has defined a `timestamps` attribute. According to the profile, the attribute must be an object containing a `created` member and a `modified` member and these members' values must use the RFC 3339 format. With such a profile applied, a response might appear as follows:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json;
profile="https://example.com/resource-timestamps"

// ...
{
  "type": "articles",
  "id": "1",
  "attributes": {
    "title": "Rails is Omakase",
    "timestamps": {
      "created": "2020-07-21T12:09:00Z",
      "modified": "2020-07-30T10:19:01Z"
    }
  }
}
// ...
```

6 Content Negotiation

6.1 Universal Responsibilities

Clients and servers **MUST** send all JSON:API payloads using the JSON:API media type in the `Content-Type` header.

Clients and servers **MUST** specify the `ext` media type parameter in the `Content-Type` header when they have applied one or more extensions to a JSON:API document.

Clients and servers **MUST** specify the `profile` media type parameters in the `Content-Type` header when they have applied one or more profiles to a JSON:API

document.

6.2 Client Responsibilities

When processing a JSON:API response document, clients **MUST** ignore any parameters other than `ext` and `profile` parameters in the server's `Content-Type` header.

A client **MAY** use the `ext` media type parameter in an `Accept` header to require that a server apply all the specified extensions to the response document.

A client **MAY** use the `profile` media type parameter in an `Accept` header to request that the server apply one or more profiles to the response document.

> Note: A client is allowed to send more than one acceptable media type in the `Accept` header, including multiple instances of the JSON:API media type. This allows clients to request different combinations of the `ext` and `profile` media type parameters. A client can use quality values to indicate that some combinations are less preferable than others. Media types specified without a qvalue are equally preferable to each other, regardless of their order, and are always considered more preferable than a media type with a qvalue less than 1.

6.3 Server Responsibilities

If a request specifies the `Content-Type` header with the JSON:API media type, servers **MUST** respond with a `415 Unsupported Media Type` status code if that media type contains any media type parameters other than `ext` or `profile`.

If a request specifies the `Content-Type` header with an instance of the JSON:API media type modified by the `ext` media type parameter and that parameter contains an unsupported extension URI, the server **MUST** respond with a `415 Unsupported Media Type` status code.

> Note: JSON:API servers that do not support version 1.1 of this specification will respond with a `415 Unsupported Media Type` client error if the `ext` or `profile` media type parameter is present.

If a request's `Accept` header contains an instance of the JSON:API media type, servers **MUST** ignore instances of that media type which are modified by a media type parameter other than `ext` or `profile`. If all instances of that media type are modified with a media type parameter other than `ext` or `profile`, servers **MUST** respond with a `406 Not Acceptable` status code. If every instance of that media type is modified by the `ext` parameter and each contains at least one unsupported extension URI, the server **MUST** also respond with a `406 Not Acceptable`.

If the `profile` parameter is received, a server **SHOULD** attempt to apply any requested profile(s) to its response. A server **MUST** ignore any profiles that it does not recognize.

> Note: The above rules guarantee strict agreement on extensions between the client and server, while the application of profiles is left to the discretion of the server.

Servers that support the `ext` or `profile` media type parameters **SHOULD** specify the `Vary` header with `Accept` as one of its values. This applies to responses with and without any profiles or extensions applied.

> Note: Some HTTP intermediaries (e.g. CDNs) may ignore the `Vary` header unless specifically configured to respect it.

7 Document Structure

This section describes the structure of a JSON:API document, which is identified by the JSON:API media type. JSON:API documents are defined in JavaScript Object Notation (JSON) [RFC8259].

Although the same media type is used for both request and response documents, certain aspects are only applicable to one or the other. These differences are called out below.

Extensions **MAY** define new members within the document structure. These members **MUST** comply with the naming requirements specified below.

Unless otherwise noted, objects defined by this specification or any applied extensions **MUST NOT** contain any additional members. Client and server implementations **MUST** ignore non-compliant members.

> Note: These conditions allow this specification to evolve through additive changes.

7.1 Top Level

A JSON object **MUST** be at the root of every JSON:API request and response document containing data. This object defines a document's "top level".

A document **MUST** contain at least one of the following top-level members:

- `data`: the document's "primary data".
- `errors`: an array of error objects.
- `meta`: a meta object that contains non-standard meta-information.
- a member defined by an applied extension.

The members `data` and `errors` **MUST NOT** coexist in the same document.

A document **MAY** contain any of these top-level members:

- `jsonapi`: an object describing the server's implementation.
- `links`: a links object related to the primary data

- `links`: a links object related to the primary data.
- `included`: an array of resource objects that are related to the primary data and/or each other ("included resources").

If a document does not contain a top-level `data` key, the `included` member **MUST NOT** be present either.

The top-level links object **MAY** contain the following members:

- `self`: the link that generated the current response document. If a document has extensions or profiles applied to it, this link **SHOULD** be represented by a link object with the `type` target attribute specifying the JSON:API media type with all applicable parameters.
- `related`: a related resource link when the primary data represents a resource relationship.
- `describedby`: a link to a description document (e.g. OpenAPI or JSON Schema) for the current document.
- pagination links for the primary data.

> Note: The `self` link in the top-level `links` object allows a client to refresh the data represented by the current response document. The client should be able to use the provided link without applying any additional information. Therefore the link must contain the query parameters provided by the client to generate the response document. This includes but is not limited to query parameters used for [inclusion of related resources][fetching resources], [sparse fieldsets][fetching sparse fieldsets], [sorting][fetching sorting], [pagination][fetching pagination] and [filtering][fetching filtering].

The document's "primary data" is a representation of the resource or collection of resources targeted by a request.

Primary data **MUST** be either:

- a single resource object, a single resource identifier object, or `null`, for requests that target single resources
- an array of resource objects, an array of resource identifier objects, or an empty array (`[]`), for requests that target resource collections

For example, the following primary data is a single resource object:

```
{
  "data": {
    "type": "articles",
    "id": "1",
    "attributes": {
      // ... this article's attributes
    },
    "relationships": {
      // ... this article's relationships
    }
  }
}
```

The following primary data is a single resource identifier object that references the same resource:

```
{
  "data": {
    "type": "articles",
    "id": "1"
  }
}
```

A logical collection of resources **MUST** be represented as an array, even if it only contains one item or is empty.

7.2 Resource Objects

“Resource objects” appear in a JSON:API document to represent resources.

A resource object **MUST** contain at least the following top-level members:

- `id`
- `type`

Exception: The `id` member is not required when the resource object originates at the client and represents a new resource to be created on the server. In that case, a client **MAY** include a `lid` member to uniquely identify the resource by `type` *locally* within the document.

In addition, a resource object **MAY** contain any of these top-level members:

- `attributes`: an attributes object representing some of the resource’s data.
- `relationships`: a relationships object describing relationships between the resource and other JSON:API resources.
- `links`: a links object containing links related to the resource.
- `meta`: a meta object containing non-standard meta-information about a resource that can not be represented as an attribute or relationship.

Here’s how an article (i.e. a resource of type “articles”) might appear in a document:

```
// ...
{
  "type": "articles",
  "id": "1",
  "attributes": {
    "title": "Rails is Omakase"
  },
  "relationships": {
    "author": {
      "links": {
        "self": "/articles/1/relationships/author",
        "related": "/articles/1/author"
      },
      "data": { "type": "people", "id": "9" }
    }
  }
}
// ...
```

7.2.1 Identification

As noted above, every resource object **MUST** contain a `type` member. Every resource object **MUST** also contain an `id` member, except when the resource object originates at the client and represents a new resource to be created on the server. If `id` is omitted due to this exception, a `lid` member **MAY** be included to uniquely identify the resource by `type` *locally* within the document. The value of the `lid` member **MUST** be identical for every representation of the resource in the document, including resource identifier objects.

The values of the `id`, `type`, and `lid` members **MUST** be strings.

Within a given API, each resource object's `type` and `id` pair **MUST** identify a single, unique resource. (The set of URIs controlled by a server, or multiple servers acting as one, constitute an API.)

The `type` member is used to describe resource objects that share common attributes and relationships.

The values of `type` members **MUST** adhere to the same constraints as member names.

> Note: This spec is agnostic about inflection rules, so the value of `type` can be either plural or singular. However, the same value should be used consistently throughout an implementation.

7.2.2 Fields

A resource object's attributes and its relationships are collectively called its "fields".

Fields for a resource object **MUST** share a common namespace with each other and with `type` and `id`. In other words, a resource can not have an attribute and relationship with the same name, nor can it have an attribute or relationship named `type` or `id`.

7.2.2.1 Attributes

The value of the `attributes` key **MUST** be an object (an "attributes object"). Members of the attributes object ("attributes") represent information about the resource object in which it's defined.

Attributes may contain any valid JSON value, including complex data structures involving JSON objects and arrays.

Keys that reference related resources (e.g. `author_id`) **SHOULD NOT** appear as attributes. Instead, relationships **SHOULD** be used.

> Note: See fields and member names for more restrictions on this container.

7.2.2.2 Relationships

The value of the `relationships` key **MUST** be an object (a "relationships object"). Each member of a relationships object represents a "relationship" from the resource object in which it has been defined to other resource objects.

Relationships may be to-one or to-many.

A relationship's name is given by its key. The value at that key **MUST** be an object ("relationship object").

A "relationship object" **MUST** contain at least one of the following:

- `links`: a links object containing at least one of the following:
  - `self`: a link for the relationship itself (a "relationship link"). This link allows the client to directly manipulate the relationship. For example, removing an `author` through an `article`'s relationship URL would disconnect the person from the `article` without deleting the `people` resource itself. When fetched successfully, this link returns the linkage for the related resources as its primary data. (See Fetching Relationships.)
  - `related`: a related resource link
  - a member defined by an applied extension.
- `data`: resource linkage
- `meta`: a meta object that contains non-standard meta-information about the relationship.
- a member defined by an applied extension.

A relationship object that represents a to-many relationship **MAY** also contain pagination links under the `links` member, as described below. Any pagination links in a relationship object **MUST** paginate the relationship data, not the related resources.

> Note: See fields and member names for more restrictions on this container.

7.2.2.3 Related Resource Links

A "related resource link" provides access to resource objects linked in a relationship. When fetched, the related resource object(s) are returned as the response's primary data.

For example, an `article`'s `comments` relationship could specify a link that returns a collection of comment resource objects when retrieved through a `GET` request.

If present, a related resource link **MUST** reference a valid URL, even if the relationship isn't currently associated with any target resources. Additionally, a related resource link **MUST NOT** change because its relationship's content changes.

7.2.2.4 Resource Linkage

Resource linkage in a compound document allows a client to link together all of the included resource objects without having to `GET` any URLs via links.

Resource linkage **MUST** be represented as one of the following:

- `null` for empty to-one relationships.
- an empty array (`[]`) for empty to-many relationships.
- a single resource identifier object for non-empty to-one relationships.
- an array of resource identifier objects for non-empty to-many relationships.

> Note: The spec does not impart meaning to order of resource identifier objects in linkage arrays of to-many relationships, although implementations may do that. Arrays of resource identifier objects may represent ordered or unordered relationships, and both types can be mixed in one response object.

For example, the following article is associated with an `author`:

```
// ...
{
  "type": "articles",
  "id": "1",
  "attributes": {
    "title": "Rails is Omakase"
  },
  "relationships": {
    "author": {
      "links": {
        "self": "http://example.com/articles/1/relationships/author",
        "related": "http://example.com/articles/1/author"
      },
      "data": { "type": "people", "id": "9" }
    }
  },
  "links": {
    "self": "http://example.com/articles/1"
  }
}
// ...
```

The `author` relationship includes a link for the relationship itself (which allows the client to change the related author directly), a related resource link to fetch the resource objects, and linkage information.

7.2.3 Resource Links

The optional `links` member within each resource object contains links related to the resource.

If present, this links object **MAY** contain a `self` link that identifies the resource represented by the resource object.

```
// ...
{
  "type": "articles",
  "id": "1",
  "attributes": {
    "title": "Rails is Omakase"
  },
  "links": {
    "self": "http://example.com/articles/1"
  }
}
```

```
// ...
```

A server **MUST** respond to a `GET` request to the specified URL with a response that includes the resource as the primary data.

7.3 Resource Identifier Objects

A “resource identifier object” is an object that identifies an individual resource.

A “resource identifier object” **MUST** contain a `type` member. It **MUST** also contain an `id` member, except when it represents a new resource to be created on the server. In this case, a `lid` member **MUST** be included that identifies the new resource.

The values of the `id`, `type`, and `lid` members **MUST** be strings.

A “resource identifier object” **MAY** also include a `meta` member, whose value is a meta object that contains non-standard meta-information.

7.4 Compound Documents

Servers **MAY** allow responses that include related resources along with the requested primary resources. Such responses are called “compound documents”.

In a compound document, all included resources **MUST** be represented as an array of resource objects in a top-level `included` member.

Every included resource object **MUST** be identified via a chain of relationships originating in a document’s primary data. This means that compound documents require “full linkage” and that no resource object can be included without a direct or indirect relationship to the document’s primary data.

The only exception to the full linkage requirement is when relationship fields that would otherwise contain linkage data are excluded due to sparse fieldsets requested by the client.

A complete example document with multiple included relationships:

```
{
  "data": [{
    "type": "articles",
    "id": "1",
    "attributes": {
      "title": "JSON:API paints my bikeshed!"
    },
    "links": {
      "self": "http://example.com/articles/1"
    },
    "relationships": {
      "author": {
        "links": {
          "self": "http://example.com/articles/1/relationships/author",
          "related": "http://example.com/articles/1/author"
        },
        "data": { "type": "people", "id": "9" }
      },
      "comments": {
```

```
        "links": {
          "self": "http://example.com/articles/1/relationships/comments",
          "related": "http://example.com/articles/1/comments"
        },
        "data": [
          { "type": "comments", "id": "5" },
          { "type": "comments", "id": "12" }
        ]
      }
    }
  }],
  "included": [{
    "type": "people",
    "id": "9",
    "attributes": {
      "firstName": "Dan",
      "lastName": "Gebhardt",
      "twitter": "dgeb"
    },
    "links": {
      "self": "http://example.com/people/9"
    }
  }, {
    "type": "comments",
    "id": "5",
    "attributes": {
      "body": "First!"
    },
    "relationships": {
      "author": {
        "data": { "type": "people", "id": "2" }
      }
    },
    "links": {
      "self": "http://example.com/comments/5"
    }
  }, {
    "type": "comments",
    "id": "12",
    "attributes": {
      "body": "I like XML better"
    },
    "relationships": {
      "author": {
        "data": { "type": "people", "id": "9" }
      }
    },
    "links": {
      "self": "http://example.com/comments/12"
    }
  }]
}
```

A compound document **MUST NOT** include more than one resource object for each `type` and `id` pair.

Note: In a single document, you can think of the `type` and `id` as a composite key that uniquely references resource objects in another part of the document.

Note: For resources that do not contain an `id` member but do contain a `lid`, the `lid` is sufficient to establish resource identity and thus linkage between resource objects and resource identifier objects throughout the document.

Note: This approach ensures that a single canonical resource object is returned with each response, even when the same resource is referenced multiple times.

7.5 Meta Information

Where specified, a `meta` member can be used to include non-standard meta-information. The value of each `meta` member **MUST** be an object (a “meta object”).

Any members **MAY** be specified within `meta` objects.

For example:

```
{
  "meta": {
    "copyright": "Copyright 2015 Example Corp.",
    "authors": [
      "Yehuda Katz",
      "Steve Klabnik",
      "Dan Gebhardt",
      "Tyler Kellen"
    ]
  },
  "data": {
    // ...
  }
}
```

7.6 Links

Where specified, a `links` member can be used to represent links. The value of this member **MUST** be an object (a “links object”).

Within this object, a link **MUST** be represented as either:

- a string whose value is a URI-reference [RFC3986 Section 4.1] pointing to the link’s target,
- a link object or
- `null` if the link does not exist.

A link’s relation type **SHOULD** be inferred from the name of the link unless the link is a link object and the link object has a `rel` member.

A link’s context is the top-level object, resource object, or relationship object in which it appears.

In the example below, the `self` link is a string whereas the `related` link is a link object. The `related` link object provides additional information about the targeted related resource collection as well as a schema that serves as a description document for that collection:

```
"links": {
  "self": "http://example.com/articles/1/relationships/comments",
  "related": {
    "href": "http://example.com/articles/1/comments",
    "title": "Comments",
```

```
      "describedby": "http://example.com/schemas/article-comments",
      "meta": {
        "count": 10
      }
    }
  }
```

7.6.1 Link objects

A “link object” is an object that represents a web link.

A link object **MUST** contain the following member:

- `href`: a string whose value is a URI-reference [RFC3986 Section 4.1] pointing to the link’s target.

A link object **MAY** also contain any of the following members:

- `rel`: a string indicating the link’s relation type. The string **MUST** be a valid link relation type.
- `describedby`: a link to a description document (e.g. OpenAPI or JSON Schema) for the link target.
- `title`: a string which serves as a label for the destination of a link such that it can be used as a human-readable identifier (e.g., a menu entry).
- `type`: a string indicating the media type of the link’s target.
- `hreflang`: a string or an array of strings indicating the language(s) of the link’s target. An array of strings indicates that the link’s target is available in multiple languages. Each string **MUST** be a valid language tag [RFC5646].
- `meta`: a meta object containing non-standard meta-information about the link.

> Note: the `type` and `hreflang` members are only hints; the target resource is not guaranteed to be available in the indicated media type or language when the link is actually followed.

7.7 JSON:API Object

A JSON:API document **MAY** include information about its implementation under a top level `jsonapi` member. If present, the value of the `jsonapi` member **MUST** be an object (a “jsonapi object”).

The jsonapi object **MAY** contain any of the following members:

- `version` - whose value is a string indicating the highest JSON:API version supported.
- `ext` - an array of URIs for all applied extensions.
- `profile` - an array of URIs for all applied profiles.
- `meta` - a meta object that contains non-standard meta-information.

Clients and servers **MUST NOT** use an `ext` or `profile` member for content negotiation. Content negotiation **MUST** only happen based on media type parameters in `Content-Type` header.

A simple example appears below:

```
{
  "jsonapi": {
    "version": "1.1",
    "ext": [
      "https://jsonapi.org/ext/atomic"
    ],
    "profile": [
      "http://example.com/profiles/flexible-pagination",
      "http://example.com/profiles/resource-versioning"
    ]
  }
}
```

If the `version` member is not present, clients should assume the server implements at least version 1.0 of the specification.

> Note: Because JSON:API is committed to making additive changes only, the version string primarily indicates which new features a server may support.

7.8 Member Names

Implementation and profile defined member names used in a JSON:API document **MUST** be treated as case sensitive by clients and servers, and they **MUST** meet all of the following conditions:

- Member names **MUST** contain at least one character.
- Member names **MUST** contain only the allowed characters listed below.
- Member names **MUST** start and end with a “globally allowed character”, as defined below.

To enable an easy mapping of member names to URLs, it is **RECOMMENDED** that member names use only non-reserved, URL safe characters specified in RFC 3986.

7.8.1 Allowed Characters

The following “globally allowed characters” **MAY** be used anywhere in a member name:

- U+0061 to U+007A, “a-z”
- U+0041 to U+005A, “A-Z”
- U+0030 to U+0039, “0-9”
- U+0080 and above (non-ASCII Unicode characters; *not recommended, not URL safe*)

Additionally, the following characters are allowed in member names, except as the first or last character:

- U+002D HYPHEN-MINUS, “-“
- U+005F LOW LINE, “_”
- U+0020 SPACE, “ “ *(not recommended, not URL safe)*

7.8.2 Reserved Characters

The following characters **MUST NOT** be used in implementation and profile defined member names:

- U+002B PLUS SIGN, "+" *(has overloaded meaning in URL query strings)*
- U+002C COMMA, "," *(used as a separator between relationship paths)*
- U+002E PERIOD, "." *(used as a separator within relationship paths)*
- U+005B LEFT SQUARE BRACKET, "[" *(used in query parameter families)*
- U+005D RIGHT SQUARE BRACKET, "]" *(used in query parameter families)*
- U+0021 EXCLAMATION MARK, "!"
- U+0022 QUOTATION MARK, '"'
- U+0023 NUMBER SIGN, "#"
- U+0024 DOLLAR SIGN, "$"
- U+0025 PERCENT SIGN, "%"
- U+0026 AMPERSAND, "&"
- U+0027 APOSTROPHE, "'"
- U+0028 LEFT PARENTHESIS, "("
- U+0029 RIGHT PARENTHESIS, ")"
- U+002A ASTERISK, "*"
- U+002F SOLIDUS, "/"
- U+003A COLON, ":"
- U+003B SEMICOLON, ";"
- U+003C LESS-THAN SIGN, "<"
- U+003D EQUALS SIGN, "="
- U+003E GREATER-THAN SIGN, ">"
- U+003F QUESTION MARK, "?"
- U+0040 COMMERCIAL AT, "@" (except as first character in @-Members)
- U+005C REVERSE SOLIDUS, "\"
- U+005E CIRCUMFLEX ACCENT, "^"
- U+0060 GRAVE ACCENT, "`"
- U+007B LEFT CURLY BRACKET, "{"
- U+007C VERTICAL LINE, "|"
- U+007D RIGHT CURLY BRACKET, "}"
- U+007E TILDE, "~"
- U+007F DELETE
- U+0000 to U+001F (C0 Controls)

7.8.3 @-Members

Member names **MAY** also begin with an at sign (U+0040 COMMERCIAL AT, "@"). Members named this way are called "@-Members". @-Members **MAY** appear anywhere in a document.

This specification provides no guidance on the meaning or usage of @-Members, which are considered to be implementation semantics. @-Members **MUST** be ignored when interpreting this specification's definitions and processing instructions given outside of this subsection. For example, an attribute is defined above as any member of the attributes object. However, because @-Members must be ignored when interpreting that definition, an @-Member that occurs in an attributes object is not an attribute.

> Note: Among other things, “@” members can be used to add JSON-LD data to a JSON:API document. Such documents should be served with an extra header to convey to JSON-LD clients that they contain JSON-LD data.

7.8.4 Extension Members

The name of every new member introduced by an extension **MUST** be prefixed with the extension’s namespace followed by a colon (`:`). The remainder of the name **MUST** adhere to the rules for implementation defined member names.

8 Fetching Data

Data, including resources and relationships, can be fetched by sending a `GET` request to an endpoint.

Responses can be further refined with the optional features described below.

8.1 Fetching Resources

A server **MUST** support fetching resource data for every URL provided as:

- a `self` link as part of the top-level links object
- a `self` link as part of a resource-level links object
- a `related` link as part of a relationship-level links object

For example, the following request fetches a collection of articles:

```
GET /articles HTTP/1.1
Accept: application/vnd.api+json
```

The following request fetches an article:

```
GET /articles/1 HTTP/1.1
Accept: application/vnd.api+json
```

And the following request fetches an article’s author:

```
GET /articles/1/author HTTP/1.1
Accept: application/vnd.api+json
```

8.1.1 Responses

8.1.1.1 200 OK

A server **MUST** respond to a successful request to fetch an individual resource or resource collection with a `200 OK` response.

A server **MUST** respond to a successful request to fetch a resource collection with an

A server **MUST** respond to a successful request to fetch a resource collection with an array of resource objects or an empty array (`[]`) as the response document's primary data.

For example, a `GET` request to a collection of articles could return:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "links": {
    "self": "http://example.com/articles"
  },
  "data": [{
    "type": "articles",
    "id": "1",
    "attributes": {
      "title": "JSON:API paints my bikeshed!"
    }
  }, {
    "type": "articles",
    "id": "2",
    "attributes": {
      "title": "Rails is Omakase"
    }
  }]
}
```

A similar response representing an empty collection would be:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "links": {
    "self": "http://example.com/articles"
  },
  "data": []
}
```

A server **MUST** respond to a successful request to fetch an individual resource with a resource object or `null` provided as the response document's primary data.

`null` is only an appropriate response when the requested URL is one that might correspond to a single resource, but doesn't currently.

> Note: Consider, for example, a request to fetch a to-one related resource link. This request would respond with `null` when the relationship is empty (such that the link is corresponding to no resources) but with the single related resource's resource object otherwise.

For example, a `GET` request to an individual article could return:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
```

```
  "links": {
    "self": "http://example.com/articles/1"
  },
  "data": {
    "type": "articles",
    "id": "1",
    "attributes": {
      "title": "JSON:API paints my bikeshed!"
    },
    "relationships": {
      "author": {
        "links": {
          "related": "http://example.com/articles/1/author"
        }
      }
    }
  }
}
```

If the above article's author is missing, then a `GET` request to that related resource would return:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "links": {
    "self": "http://example.com/articles/1/author"
  },
  "data": null
}
```

8.1.1.2 404 Not Found

A server **MUST** respond with `404 Not Found` when processing a request to fetch a single resource that does not exist, except when the request warrants a `200 OK` response with `null` as the primary data (as described above).

8.1.1.3 Other Responses

A server **MAY** respond with other HTTP status codes.

A server **MAY** include error details with error responses.

A server **MUST** prepare responses, and a client **MUST** interpret responses, in accordance with `HTTP semantics`.

8.2 Fetching Relationships

A server **MUST** support fetching relationship data for every relationship URL provided as a `self` link as part of a relationship's `links` object.

For example, the following request fetches data about an article's comments:

```
GET /articles/1/relationships/comments HTTP/1.1
Accept: application/vnd.api+json
```

And the following request fetches data about an article's author:

```
GET /articles/1/relationships/author HTTP/1.1
Accept: application/vnd.api+json
```

8.2.1 Responses

8.2.1.1 200 OK

A server **MUST** respond to a successful request to fetch a relationship with a `200 OK` response.

The primary data in the response document **MUST** match the appropriate value for resource linkage, as described above for relationship objects.

The top-level links object **MAY** contain `self` and `related` links, as described above for relationship objects.

For example, a `GET` request to a URL from a to-one relationship link could return:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "links": {
    "self": "/articles/1/relationships/author",
    "related": "/articles/1/author"
  },
  "data": {
    "type": "people",
    "id": "12"
  }
}
```

If the above relationship is empty, then a `GET` request to the same URL would return:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "links": {
    "self": "/articles/1/relationships/author",
    "related": "/articles/1/author"
  },
  "data": null
}
```

A `GET` request to a URL from a to-many relationship link could return:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "links": {
    "self": "/articles/1/relationships/tags",
    "related": "/articles/1/tags"
```

```
  },
  "data": [
    { "type": "tags", "id": "2" },
    { "type": "tags", "id": "3" }
  ]
}
```

If the above relationship is empty, then a `GET` request to the same URL would return:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.api+json

{
  "links": {
    "self": "/articles/1/relationships/tags",
    "related": "/articles/1/tags"
  },
  "data": []
}
```

8.2.1.2 404 Not Found

A server **MUST** return `404 Not Found` when processing a request to fetch a relationship link URL that does not exist.

> Note: This can happen when the parent resource of the relationship does not exist. For example, when `/articles/1` does not exist, request to `/articles/1/relationships/tags` returns `404 Not Found`.

If a relationship link URL exists but the relationship is empty, then `200 OK` **MUST** be returned, as described above.

8.2.1.3 Other Responses

A server **MAY** respond with other HTTP status codes.

A server **MAY** include error details with error responses.

A server **MUST** prepare responses, and a client **MUST** interpret responses, in accordance with `HTTP semantics`.

8.3 Inclusion of Related Resources

An endpoint **MAY** return resources related to the primary data by default.

An endpoint **MAY** also support an `include` query parameter to allow the client to customize which related resources should be returned.

If an endpoint does not support the `include` parameter, it **MUST** respond with `400 Bad Request` to any requests that include it.

If an endpoint supports the `include` parameter and a client supplies it:

- The server's response **MUST** be a compound document with an `included` key

— even if that `included` key holds an empty array (because the requested relationships are empty).
- The server **MUST NOT** include unrequested resource objects in the `included` section of the compound document.

The value of the `include` parameter **MUST** be a comma-separated (U+002C COMMA, ",") list of relationship paths. A relationship path is a dot-separated (U+002E FULL-STOP, ".") list of relationship names. An empty value indicates that no related resources should be returned.

If a server is unable to identify a relationship path or does not support inclusion of resources from a path, it **MUST** respond with 400 Bad Request.

> Note: For example, a relationship path could be `comments.author`, where `comments` is a relationship listed under a `articles` resource object, and `author` is a relationship listed under a `comments` resource object.

For instance, comments could be requested with an article:

```
GET /articles/1?include=comments HTTP/1.1
Accept: application/vnd.api+json
```

In order to request resources related to other resources, a dot-separated path for each relationship name can be specified:

```
GET /articles/1?include=comments.author HTTP/1.1
Accept: application/vnd.api+json
```

> Note: Because compound documents require full linkage (except when relationship linkage is excluded by sparse fieldsets), intermediate resources in a multi-part path must be returned along with the leaf nodes. For example, a response to a request for `comments.author` should include `comments` as well as the `author` of each of those `comments`.

> Note: A server may choose to expose a deeply nested relationship such as `comments.author` as a direct relationship with an alternative name such as `commentAuthors`. This would allow a client to request `/articles/1?include=commentAuthors` instead of `/articles/1?include=comments.author`. By exposing the nested relationship with an alternative name, the server can still provide full linkage in compound documents without including potentially unwanted intermediate resources.

Multiple related resources can be requested in a comma-separated list:

```
GET /articles/1?include=comments.author,ratings HTTP/1.1
Accept: application/vnd.api+json
```

Furthermore, related resources can be requested from a relationship endpoint:

```
GET /articles/1/relationships/comments?include=comments.author HTTP/1.1
Accept: application/vnd.api+json
```

In this case, the primary data would be a collection of resource identifier objects that represent linkage to comments for an article, while the full comments and comment authors would be returned as included data.

> Note: This section applies to any endpoint that responds with primary data, regardless of the request type. For instance, a server could support the inclusion of related resources along with a `POST` request to create a resource or relationship.

8.4 Sparse Fieldsets

A client **MAY** request that an endpoint return only specific fields in the response on a per-type basis by including a `fields[TYPE]` query parameter.

The value of any `fields[TYPE]` parameter **MUST** be a comma-separated (U+002C COMMA, ",") list that refers to the name(s) of the fields to be returned. An empty value indicates that no fields should be returned.

If a client requests a restricted set of fields for a given resource type, an endpoint **MUST NOT** include additional fields in resource objects of that type in its response.

If a client does not specify the set of fields for a given resource type, the server **MAY** send all fields, a subset of fields, or no fields for that resource type.

```
GET /articles?
include=author&fields[articles]=title,body&fields[people]=name HTTP/1.1
Accept: application/vnd.api+json
```

> Note: The above example URI shows unencoded `[` and `]` characters simply for readability. In practice, these characters should be percent-encoded. See "Square Brackets in Parameter Names".

> Note: This section applies to any endpoint that responds with resources as primary or included data, regardless of the request type. For instance, a server could support sparse fieldsets along with a `POST` request to create a resource.

8.5 Sorting

Sorting

A server **MAY** choose to support requests to sort resource collections according to one or more criteria ("sort fields").

> Note: Although recommended, sort fields do not necessarily need to correspond to resource attribute and relationship names.

> Note: It is recommended that dot-separated (U+002E FULL-STOP, ".") sort fields be used to request sorting based upon relationship attributes. For example, a sort field of `author.name` could be used to request that the primary data be sorted based upon the `name` attribute of the `author` relationship.

An endpoint **MAY** support requests to sort the primary data with a `sort` query parameter. The value for `sort` **MUST** represent sort fields.

```
GET /people?sort=age HTTP/1.1
Accept: application/vnd.api+json
```

An endpoint **MAY** support multiple sort fields by allowing comma-separated (U+002C COMMA, ",") sort fields. Sort fields **SHOULD** be applied in the order specified.

```
GET /people?sort=age,name HTTP/1.1
Accept: application/vnd.api+json
```

The sort order for each sort field **MUST** be ascending unless it is prefixed with a minus (U+002D HYPHEN-MINUS, "-"), in which case it **MUST** be descending.

```
GET /articles?sort=-created,title HTTP/1.1
Accept: application/vnd.api+json
```

The above example should return the newest articles first. Any articles created on the same date will then be sorted by their title in ascending alphabetical order.

If the server does not support sorting as specified in the query parameter `sort`, it **MUST** return `400 Bad Request`.

If sorting is supported by the server and requested by the client via query parameter `sort`, the server **MUST** return elements of the top-level `data` array of the response ordered according to the criteria specified. The server **MAY** apply default sorting rules to top-level `data` if request parameter `sort` is not specified.

> Note: This section applies to any endpoint that responds with a resource collection as primary data, regardless of the request type.

8.6 Pagination

A server **MAY** choose to limit the number of resources returned in a response to a subset ("page") of the whole set available.

A server **MAY** provide links to traverse a paginated data set ("pagination links").

Pagination links **MUST** appear in the links object that corresponds to a collection. To paginate the primary data, supply pagination links in the top-level `links` object. To paginate an included collection returned in a compound document, supply pagination links in the corresponding links object.

The following keys **MUST** be used for pagination links:

- `first`: the first page of data
- `last`: the last page of data
- `prev`: the previous page of data
- `next`: the next page of data

Keys **MUST** either be omitted or have a `null` value to indicate that a particular link is unavailable.

Concepts of order, as expressed in the naming of pagination links, **MUST** remain consistent with JSON:API's sorting rules.

The `page` query parameter family is reserved for pagination. Servers and clients **SHOULD** use these parameters for pagination operations.

> Note: JSON API is agnostic about the pagination strategy used by a server, but the `page` query parameter family can be used regardless of the strategy employed. For example, a page-based strategy might use query parameters such as `page[number]` and `page[size]`, while a cursor-based strategy might use `page[cursor]`.

> Note: This section applies to any endpoint that responds with a resource collection as primary data, regardless of the request type.

8.7 Filtering

The `filter` query parameter family is reserved for filtering data. Servers and clients **SHOULD** use these parameters for filtering operations.

> Note: JSON API is agnostic about the strategies supported by a server.

9 Creating, Updating and Deleting Resources

A server **MAY** allow resources of a given type to be created. It **MAY** also allow existing

resources to be modified or deleted.

A request **MUST** completely succeed or fail (in a single “transaction”). No partial updates are allowed.

> Note: The `type` member is required in every resource object throughout requests and responses in JSON:API. There are some cases, such as when `POST`ing to an endpoint representing heterogeneous data, when the `type` could not be inferred from the endpoint. However, picking and choosing when it is required would be confusing; it would be hard to remember when it was required and when it was not. Therefore, to improve consistency and minimize confusion, `type` is always required.

9.1 Creating Resources

A resource can be created by sending a `POST` request to a URL that represents a collection of resources. The request **MUST** include a single resource object as primary data. The resource object **MUST** contain at least a `type` member.

For instance, a new photo might be created with the following request:

```
POST /photos HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
    "type": "photos",
    "attributes": {
      "title": "Ember Hamster",
      "src": "http://example.com/images/productivity.png"
    },
    "relationships": {
      "photographer": {
        "data": { "type": "people", "id": "9" }
      }
    }
  }
}
```

If a relationship is provided in the `relationships` member of the resource object, its value **MUST** be a relationship object with a `data` member. The value of this key represents the linkage the new resource is to have.

9.1.1 Client-Generated IDs

A server **MAY** accept a client-generated ID along with a request to create a resource. An ID **MUST** be specified with an `id` key, the value of which **MUST** be a universally unique identifier. The client **SHOULD** use a properly generated and formatted *UUID* as described in RFC 4122 [RFC4122].

> NOTE: In some use-cases, such as importing data from another source, it may be

possible to use something other than a UUID that is still guaranteed to be globally unique. Do not use anything other than a UUID unless you are 100% confident that the strategy you are using indeed generates globally unique identifiers.

For example:

```
POST /photos HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
    "type": "photos",
    "id": "550e8400-e29b-41d4-a716-446655440000",
    "attributes": {
      "title": "Ember Hamster",
      "src": "http://example.com/images/productivity.png"
    }
  }
}
```

A server **MUST** return `403 Forbidden` in response to an unsupported request to create a resource with a client-generated ID.

9.1.2 Responses

9.1.2.1 201 Created

If the requested resource has been created successfully and the server changes the resource in any way (for example, by assigning an `id`), the server **MUST** return a `201 Created` response and a document that contains the resource as primary data.

The response **SHOULD** include a `Location` header identifying the location of the newly created resource, in order to comply with RFC 7231.

If the resource object returned by the response contains a `self` key in its `links` member and a `Location` header is provided, the value of the `self` member **MUST** match the value of the `Location` header.

```
HTTP/1.1 201 Created
Location: http://example.com/photos/550e8400-e29b-41d4-a716-446655440000
Content-Type: application/vnd.api+json

{
  "data": {
    "type": "photos",
    "id": "550e8400-e29b-41d4-a716-446655440000",
    "attributes": {
      "title": "Ember Hamster",
      "src": "http://example.com/images/productivity.png"
    },
    "links": {
      "self": "http://example.com/photos/550e8400-e29b-41d4-a716-
446655440000"
    }
  }
}
```

A server **MAY** return a `201 Created` response with a document that contains no primary data if the requested resource has been created successfully and the server does not change the resource in any way (for example, by assigning an `id` or `createdAt` attribute). Other top-level members, such as meta, could be included in the response document.

> Note: Only servers that accept Client-Generated IDs can avoid assigning an `id` to a new resource.

9.1.2.2 202 Accepted

If a request to create a resource has been accepted for processing, but the processing has not been completed by the time the server responds, the server **MUST** return a `202 Accepted` status code.

9.1.2.3 204 No Content

If the requested resource has been created successfully and the server does not change the resource in any way (for example, by assigning an `id` or `createdAt` attribute), the server **MUST** return either a `201 Created` status code and response document (as described above) or a `204 No Content` status code with no response document.

9.1.2.4 403 Forbidden

A server **MAY** return `403 Forbidden` in response to an unsupported request to create a resource.

9.1.2.5 404 Not Found

A server **MUST** return `404 Not Found` when processing a request that references a related resource that does not exist.

9.1.2.6 409 Conflict

A server **MUST** return `409 Conflict` when processing a `POST` request to create a resource with a client-generated ID that already exists.

A server **MUST** return `409 Conflict` when processing a `POST` request in which the resource object's `type` is not among the type(s) that constitute the collection represented by the endpoint.

A server **SHOULD** include error details and provide enough information to recognize the source of the conflict.

9.1.2.7 Other Responses

A server **MAY** respond with other HTTP status codes.

A server **MAY** include error details with error responses.

A server **MUST** prepare responses, and a client **MUST** interpret responses, in accordance with `HTTP semantics`.

9.2 Updating Resources

A resource can be updated by sending a `PATCH` request to the URL that represents the resource.

The URL for a resource can be obtained in the `self` link of the resource object. Alternatively, when a `GET` request returns a single resource object as primary data, the same request URL can be used for updates.

The `PATCH` request **MUST** include a single resource object as primary data. The resource object **MUST** contain `type` and `id` members.

For example:

```
PATCH /articles/1 HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
    "type": "articles",
    "id": "1",
    "attributes": {
      "title": "To TDD or Not"
    }
  }
}
```

9.2.1 Updating a Resource's Attributes

Any or all of a resource's attributes **MAY** be included in the resource object included in a `PATCH` request.

If a request does not include all of the attributes for a resource, the server **MUST** interpret the missing attributes as if they were included with their current values. The server **MUST NOT** interpret missing attributes as `null` values.

For example, the following `PATCH` request is interpreted as a request to update only the `title` and `text` attributes of an article:

```
PATCH /articles/1 HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
    "type": "articles",
    "id": "1",
    "attributes": {
      "title": "To TDD or Not",
      "text": "TLDR; It's complicated... but check your test coverage
regardless."
    }
  }
}
```

9.2.2 Updating a Resource's Relationships

Any or all of a resource's relationships **MAY** be included in the resource object included in a `PATCH` request.

If a request does not include all of the relationships for a resource, the server **MUST** interpret the missing relationships as if they were included with their current values. It **MUST NOT** interpret them as `null` or empty values.

If a relationship is provided in the `relationships` member of a resource object in a `PATCH` request, its value **MUST** be a relationship object with a `data` member. The relationship's value will be replaced with the value specified in this member.

For instance, the following `PATCH` request will update the `author` relationship of an article:

```
PATCH /articles/1 HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
    "type": "articles",
    "id": "1",
    "relationships": {
      "author": {
        "data": { "type": "people", "id": "1" }
      }
    }
  }
}
```

Likewise, the following `PATCH` request performs a complete replacement of the `tags` for an article:

```
PATCH /articles/1 HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": {
    "type": "articles",
    "id": "1",
    "relationships": {
      "tags": {
        "data": [
          { "type": "tags", "id": "2" },
          { "type": "tags", "id": "3" }
        ]
      }
    }
  }
}
```

A server **MAY** reject an attempt to do a full replacement of a to-many relationship. In such a case, the server **MUST** reject the entire update, and return a `403 Forbidden` response.

Note: Since full replacement may be a very dangerous operation, a server may choose to disallow it. For example, a server may reject full replacement if it has not provided the client with the full list of associated objects, and does not want to allow deletion of records the client has not seen.

9.2.3 Responses

9.2.3.1 200 OK

If a server accepts an update but also changes the targeted resource in ways other than those specified by the request (for example, updating the `updatedAt` attribute or a computed `sha`), it **MUST** return a `200 OK` response and a document that contains the updated resource as primary data.

A server **MAY** return a `200 OK` response with a document that contains no primary data if an update is successful and the server does not change the targeted resource in ways other than those specified by the request. Other top-level members, such as meta, could be included in the response document.

9.2.3.2 202 Accepted

If an update request has been accepted for processing, but the processing has not been completed by the time the server responds, the server **MUST** return a `202 Accepted` status code.

9.2.3.3 204 No Content

If an update is successful and the server doesn't change the targeted resource in ways other than those specified by the request, the server **MUST** return either a `200 OK` status code and response document (as described above) or a `204 No Content` status code with no response document.

9.2.3.4 403 Forbidden

A server **MUST** return `403 Forbidden` in response to an unsupported request to update a resource or relationship.

9.2.3.5 404 Not Found

A server **MUST** return `404 Not Found` when processing a request to modify a resource that does not exist.

A server **MUST** return `404 Not Found` when processing a request that references a related resource that does not exist.

9.2.3.6 409 Conflict

A server **MAY** return `409 Conflict` when processing a `PATCH` request to update a resource if that update would violate other server-enforced constraints (such as a uniqueness constraint on a property other than `id`).

A server **MUST** return `409 Conflict` when processing a `PATCH` request in which

the resource object's `type` or `id` do not match the server's endpoint.

A server **SHOULD** include error details and provide enough information to recognize the source of the conflict.

9.2.3.7 Other Responses

A server **MAY** respond with other HTTP status codes.

A server **MAY** include error details with error responses.

A server **MUST** prepare responses, and a client **MUST** interpret responses, in accordance with `HTTP semantics`.

9.3 Updating Relationships

Although relationships can be modified along with resources (as described above), JSON:API also supports updating of relationships independently at URLs from relationship links.

> Note: Relationships are updated without exposing the underlying server semantics, such as foreign keys. Furthermore, relationships can be updated without necessarily affecting the related resources. For example, if an article has many authors, it is possible to remove one of the authors from the article without deleting the person itself. Similarly, if an article has many tags, it is possible to add or remove tags. Under the hood on the server, the first of these examples might be implemented with a foreign key, while the second could be implemented with a join table, but the JSON:API protocol would be the same in both cases.

> Note: A server may choose to delete the underlying resource if a relationship is deleted (as a garbage collection measure).

9.3.1 Updating To-One Relationships

A to-one relationship can be updated by sending a `PATCH` request to a URL from a to-one relationship link.

The `PATCH` request **MUST** include a top-level member named `data` containing one of:

- a resource identifier object corresponding to the new related resource.
- `null`, to remove the relationship.

For example, the following request updates the author of an article:

```
PATCH /articles/1/relationships/author HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": { "type": "people", "id": "12" }
```

```
}
```

And the following request clears the author of the same article:

```
PATCH /articles/1/relationships/author HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": null
}
```

If the relationship is updated successfully then the server **MUST** return a successful response.

9.3.2 Updating To-Many Relationships

A to-many relationship can be updated by sending a `PATCH`, `POST`, or `DELETE` request to a URL from a to-many relationship link.

For all request types, the body **MUST** contain a `data` member whose value is an empty array or an array of resource identifier objects.

If a client makes a `PATCH` request to a URL from a to-many relationship link, the server **MUST** either completely replace every member of the relationship, return an appropriate error response if some resources cannot be found or accessed, or return a `403 Forbidden` response if complete replacement is not allowed by the server.

For example, the following request replaces every tag for an article:

```
PATCH /articles/1/relationships/tags HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": [
    { "type": "tags", "id": "2" },
    { "type": "tags", "id": "3" }
  ]
}
```

And the following request clears every tag for an article:

```
PATCH /articles/1/relationships/tags HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": []
}
```

If a client makes a `POST` request to a URL from a relationship link, the server **MUST** add the specified members to the relationship unless they are already present. If a given `type` and `id` is already in the relationship, the server **MUST NOT** add it again

`type` and `id` is already in the relationship, the server **MUST NOT** add it again.

> Note: This matches the semantics of databases that use foreign keys for has-many relationships. Document-based storage should check the has-many relationship before appending to avoid duplicates.

If all of the specified resources can be added to, or are already present in, the relationship then the server **MUST** return a successful response.

> Note: This approach ensures that a request is successful if the server's state matches the requested state, and helps avoid pointless race conditions caused by multiple clients making the same changes to a relationship.

In the following example, the comment with ID `123` is added to the list of comments for the article with ID `1`:

```
POST /articles/1/relationships/comments HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": [
    { "type": "comments", "id": "123" }
  ]
}
```

If the client makes a `DELETE` request to a URL from a relationship link the server **MUST** delete the specified members from the relationship or return a `403 Forbidden` response. If all of the specified resources are able to be removed from, or are already missing from, the relationship then the server **MUST** return a successful response.

> Note: As described above for `POST` requests, this approach helps avoid pointless race conditions between multiple clients making the same changes.

Relationship members are specified in the same way as in the `POST` request.

In the following example, comments with IDs of `12` and `13` are removed from the list of comments for the article with ID `1`:

```
DELETE /articles/1/relationships/comments HTTP/1.1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "data": [
    { "type": "comments", "id": "12" },
    { "type": "comments", "id": "13" }
  ]
}
```

Note: RFC 7231 specifies that a DELETE request may include a body, but that a server may reject the request. This spec defines the semantics of a server, and we are defining its semantics for JSON:API.

9.3.3 Responses

9.3.3.1 200 OK

If a server accepts an update but also changes the targeted relationship in other ways than those specified by the request, it **MUST** return a `200 OK` response and a document that includes the updated relationship data as its primary data.

A server **MAY** return a `200 OK` response with a document that contains no primary data if an update is successful and the server does not change the targeted relationship in ways other than those specified by the request. Other top-level members, such as meta, could be included in the response document.

9.3.3.2 202 Accepted

If a relationship update request has been accepted for processing, but the processing has not been completed by the time the server responds, the server **MUST** return a `202 Accepted` status code.

9.3.3.3 204 No Content

If an update is successful and the server doesn't change the targeted relationship in ways other than those specified by the request, the server **MUST** return either a `200 OK` status code and response document (as described above) or a `204 No Content` status code with no response document.

Note: This is the appropriate response to a `POST` request sent to a URL from a to-many relationship link when that relationship already exists. It is also the appropriate response to a `DELETE` request sent to a URL from a to-many relationship link when that relationship does not exist.

9.3.3.4 403 Forbidden

A server **MUST** return `403 Forbidden` in response to an unsupported request to update a relationship.

9.3.3.5 Other Responses

A server **MAY** respond with other HTTP status codes.

A server **MAY** include error details with error responses.

A server **MUST** prepare responses, and a client **MUST** interpret responses, in accordance with `HTTP semantics`.

9.4 Deleting Resources

A resource can be deleted by sending a `DELETE` request to the URL that represents the resource:

```
DELETE /photos/1 HTTP/1.1
Accept: application/vnd.api+json
```

9.4.1 Responses

9.4.1.1 200 OK

A server **MAY** return a `200 OK` response with a document that contains no primary data if a deletion request is successful. Other top-level members, such as meta, could be included in the response document.

9.4.1.2 202 Accepted

If a deletion request has been accepted for processing, but the processing has not been completed by the time the server responds, the server **MUST** return a `202 Accepted` status code.

9.4.1.3 204 No Content

If a deletion request is successful, the server **MUST** return either a `200 OK` status code and response document (as described above) or a `204 No Content` status code with no response document.

9.4.1.4 404 NOT FOUND

A server **SHOULD** return a `404 Not Found` status code if a deletion request fails due to the resource not existing.

9.4.1.5 Other Responses

A server **MAY** respond with other HTTP status codes.

A server **MAY** include error details with error responses.

A server **MUST** prepare responses, and a client **MUST** interpret responses, in accordance with `HTTP semantics`.

10 Query Parameters

10.1 Query Parameter Families

Although “query parameter” is a common term in everyday web development, it is not a well-standardized concept. Therefore, JSON:API provides its own definition of a query parameter.

For the most part, JSON:API’s definition coincides with colloquial usage, and its details can be safely ignored. However, one important consequence of this definition is that a

URL like the following is considered to have two distinct query parameters:

```
/?page[offset]=0&page[limit]=10
```

The two parameters are named `page[offset]` and `page[limit]`; there is no single `page` parameter.

In practice, however, parameters like `page[offset]` and `page[limit]` are usually defined and processed together, and it's convenient to refer to them collectively. Therefore, JSON:API introduces the concept of a query parameter family.

A "query parameter family" is the set of all query parameters whose name starts with a "base name", followed by zero or more instances of empty square brackets (i.e. `[]`) or square-bracketed legal member names. The family is referred to by its base name.

For example, the `filter` query parameter family includes parameters named: `filter`, `filter[x]`, `filter[]`, `filter[x][]`, `filter[][]`, `filter[x][y]`, etc. However, `filter[_]` is not a valid parameter name in the family, because `_` is not a valid member name.

10.2 Extension-Specific Query Parameters

The base name of every query parameter introduced by an extension **MUST** be prefixed with the extension's namespace followed by a colon (`:`). The remainder of the base name **MUST** contain only the characters [a-z] (U+0061 to U+007A, "a-z").

10.3 Implementation-Specific Query Parameters

Implementations **MAY** support custom query parameters. However, the names of these query parameters **MUST** come from a family whose base name is a legal member name and also contains at least one non a-z character (i.e., outside U+0061 to U+007A).

It is **RECOMMENDED** that a capital letter (e.g. camelCasing) be used to satisfy the above requirement.

If a server encounters a query parameter that does not follow the naming conventions above, or the server does not know how to process it as a query parameter from this specification, it **MUST** return `400 Bad Request`.

> Note: By forbidding the use of query parameters that contain only the characters [a-z], JSON:API is reserving the ability to standardize additional query parameters later without conflicting with existing implementations.

11 Errors

11.1 Processing Errors

A server **MAY** choose to stop processing as soon as a problem is encountered, or it **MAY** continue processing and encounter multiple problems. For instance, a server might process multiple attributes and then return multiple validation problems in a single response.

When a server encounters multiple problems for a single request, the most generally applicable HTTP error code **SHOULD** be used in the response. For instance, `400 Bad Request` might be appropriate for multiple 4xx errors or `500 Internal Server Error` might be appropriate for multiple 5xx errors.

11.2 Error Objects

Error objects provide additional information about problems encountered while performing an operation. Error objects **MUST** be returned as an array keyed by `errors` in the top level of a JSON:API document.

An error object **MAY** have the following members, and **MUST** contain at least one of:

- `id`: a unique identifier for this particular occurrence of the problem.
- `links`: a links object that **MAY** contain the following members:
  - `about`: a link that leads to further details about this particular occurrence of the problem. When derefenced, this URI **SHOULD** return a human-readable description of the error.
  - `type`: a link that identifies the type of error that this particular error is an instance of. This URI **SHOULD** be dereferencable to a human-readable explanation of the general error.
- `status`: the HTTP status code applicable to this problem, expressed as a string value. This **SHOULD** be provided.
- `code`: an application-specific error code, expressed as a string value.
- `title`: a short, human-readable summary of the problem that **SHOULD NOT** change from occurrence to occurrence of the problem, except for purposes of localization.
- `detail`: a human-readable explanation specific to this occurrence of the problem. Like `title`, this field's value can be localized.
- `source`: an object containing references to the primary source of the error. It **SHOULD** include one of the following members or be omitted:
  - `pointer`: a JSON Pointer [RFC6901] to the value in the request document that caused the error [e.g. `"/data"` for a primary data object, or `"/data/attributes/title"` for a specific attribute]. This **MUST** point to a value in the request document that exists; if it doesn't, the client **SHOULD** simply ignore the pointer.
  - `parameter`: a string indicating which URI query parameter caused the error.
  - `header`: a string indicating the name of a single request header which caused the error.
- `meta`: a meta object containing non-standard meta-information about the error.

12 Appendix

IPR2025-01086
Orca Security EX1018 Page 39

12 Appendix

12.1 Query Parameters Details

12.1.1 Parsing/Serialization

A query parameter is a name–value pair extracted from, or serialized into, a URI's query string.

To extract the query parameters from a URI, an implementation **MUST** run the URI's query string, excluding the leading question mark, through the `application/x-www-form-urlencoded` parsing algorithm, with one exception: JSON:API allows the specification that defines a query parameter's usage to provide its own rules for parsing the parameter's value from the `value` bytes identified in steps 3.2 and and 3.3 of the `application/x-www-form-urlencoded` parsing algorithm. The resulting value might not be a string.

> Note: In general, the query string parsing built in to servers and browsers will match the process specified above, so most implementations do not need to worry about this.
>
> The `application/x-www-form-urlencoded` format is referenced because it is the basis for the `a=b&c=d` style that almost all query strings use today.
>
> However, `application/x-www-form-urlencoded` parsing contains the bizarre historical artifact that `+` characters must be treated as spaces, and it requires that all values be percent-decoded during parsing, which makes it impossible to use RFC 3986 delimiter characters as delimiters. These issues motivate the exception that JSON:API defines above.

Similarly, to serialize a query parameter into a URI, an implementation **MUST** use the the `application/x-www-form-urlencoded` serializer, with the corresponding exception that a parameter's value — but not its name — may be serialized differently than that algorithm requires, provided the serialization does not interfere with the ability to parse back the resulting URI.

12.1.2 Square Brackets in Parameter Names

With query parameter families, JSON:API allows for query parameters whose names contain square brackets (i.e., U+005B "[" and U+005D "]").

According to the query parameter serialization rules above, a compliant implementation will percent-encode these square brackets. However, some URI producers — namely browsers — do not always encode them. Servers **SHOULD** accept requests in which these square brackets are left unencoded in a query parameter's name. If a server does accept these requests, it **MUST** treat the request as equivalent to one in which the square brackets were percent-encoded.

Built with Jekyll and Highlight.js

