TABLE OF CONTENTS

- Status
- Introduction
- Conventions
- Content Negotiation
  - Client Responsibilities
  - Server Responsibilities
- Document Structure
  - Top Level
  - Resource Objects
    - Identification
    - Fields
    - Attributes
    - Relationships
    - Related Resource Links
    - Resource Linkage
    - Resource Links
  - Resource Identifier Objects
  - Compound Documents
  - Meta Information
  - Links
  - JSON:API Object
  - Member Names
    - Allowed Characters
    - Reserved Characters

- Fetching Data
  - Fetching Resources
    - Responses
  - 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
    - 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
- Errors
  - Processing Errors
  - Error Objects

Latest Specification (v1.0)

Status

This page presents the latest published version of JSON:API, which is currently version 1.0. 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 <http://discuss.jsonapi.org/>.

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 <https://github.com/json-api/json-api>.

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 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` <http://www.iana.org/assignments/media-types/application/vnd.api+json>) for exchanging data.

Conventions

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119 [RFC2119 <http://tools.ietf.org/html/rfc2119>].

Content Negotiation

Client Responsibilities

Clients **MUST** send all JSON:API data in request documents with the header `Content-Type: application/vnd.api+json` without any media type parameters.

Clients that include the JSON:API media type in their `Accept` header **MUST** specify the media type there at least once without any media type parameters.

Clients **MUST** ignore any parameters for the `application/vnd.api+json` media type received in the `Content-Type` header of response documents.

Server Responsibilities

Servers **MUST** send all JSON:API data in response documents with the header `Content-Type: application/vnd.api+json` without any media type parameters.

Servers **MUST** respond with a `415 Unsupported Media Type` status code if a request specifies the header `Content-Type: application/vnd.api+json` with any media type parameters.

Servers **MUST** respond with a `406 Not Acceptable` status code if a request's `Accept` header contains the JSON:API media type and all instances of that media type are modified with media type parameters.

Note: The content negotiation requirements exist to allow future versions of this specification to use media type parameters for extension negotiation and versioning.

Document Structure

This section describes the structure of a JSON:API document, which is identified by the media type `application/vnd.api+json` <http://www.iana.org/assignments/media-types/application/vnd.api+json>. JSON:API documents are defined in JavaScript Object Notation (JSON) [RFC7159 <http://tools.ietf.org/html/rfc7159>].

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.

Unless otherwise noted, objects defined by this specification **MUST NOT** contain any additional members. Client and server implementations **MUST** ignore members not recognized by this specification.

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

Top Level

A JSON object **MUST** be at the root of every JSON:API request and response 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.

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.
- `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.
- `related` : a related resource link when the primary data represents a resource relationship.
- pagination links for the primary data.

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.

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 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" }
    }
  }
}
// ...
```

Identification

Every resource object **MUST** contain an `id` member and a `type` member. The values of the `id` and `type` 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.

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`.

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.

Complex data structures involving JSON objects and arrays are allowed as attribute values. However, any object that constitutes or is contained in an attribute **MUST NOT** contain a `relationships` or `links` member, as those members are reserved by this specification for future use.

Although has-one foreign keys (e.g. `author_id`) are often stored internally alongside other information to be represented in a resource object, these keys **SHOULD NOT** appear as attributes.

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

Relationships

The value of the `relationships` key **MUST** be an object (a "relationships object"). Members of the relationships object ("relationships") represent references from the resource object in which it's defined to other resource objects.

Relationships may be to-one or to-many.

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
- `data` : resource linkage
- `meta` : a meta object that contains non-standard meta-information about the relationship.

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.

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.

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.

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.

Resource Identifier Objects

A "resource identifier object" is an object that identifies an individual resource.

A "resource identifier object" **MUST** contain `type` and `id` members.

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

Compound Documents

To reduce the number of HTTP requests, 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.

Compound documents require "full linkage", meaning that every included resource **MUST** be identified by at least one resource identifier object in the same document. These resource identifier objects could either be primary data or represent resource linkage contained within primary or included resources.

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

> Note: Full linkage ensures that included resources are related to either the primary data (which could be resource objects or resource identifier objects) or to each other.

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": {
      "first-name": "Dan",
      "last-name": "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: This approach ensures that a single canonical resource object is returned with each response, even when the same resource is referenced multiple times.

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": {
    // ...
  }
}
```

Links

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

Each member of a links object is a "link". A link **MUST** be represented as either:

- a string containing the link's URL.
- an object ("link object") which can contain the following members:
- `href` : a string containing the link's URL.

- `meta` : a meta object containing non-standard meta-information about the link.

The following `self` link is simply a URL:

```
"links": {
  "self": "http://example.com/posts"
}
```

The following `related` link includes a URL as well as meta-information about a related resource collection:

```
"links": {
  "related": {
    "href": "http://example.com/articles/1/comments",
    "meta": {
      "count": 10
    }
  }
}
```

Note: Additional members may be specified for links objects and link objects in the future. It is also possible that the allowed values of additional members will be expanded (e.g. a `collection` link may support an array of values, whereas a `self` link does not).

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 a `version` member whose value is a string indicating the highest JSON API version supported. This object **MAY** also contain a `meta` member, whose value is a meta object that contains non-standard meta-information.

```
{
  "jsonapi": {
    "version": "1.0"
  }
}
```

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.

Member Names

All 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 <http://tools.ietf.org/html/rfc3986#page-13>.

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)*

Reserved Characters

The following characters **MUST NOT** be used in member names:

- U+002B PLUS SIGN, “+” *(used for ordering)*
- 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 sparse fieldsets)*
- U+005D RIGHT SQUARE BRACKET, “]” *(used in sparse fieldsets)*
- 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, “@”
- 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)

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.

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
```

Responses

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 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
}
```

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).

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` <http://tools.ietf.org/html/rfc7231>.

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
```

Responses

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": []
}
```

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.

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` <http://tools.ietf.org/html/rfc7231>.

Inclusion of Related Resources

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

An endpoint **MAY** also support an `include` request 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 **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.

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 alias such as `comment-authors`. This would allow a client to request `/articles/1?include=comment-authors` instead of `/articles/1?include=comments.author`. By abstracting the nested relationship with an alias, 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=author,comments.author 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.

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]` parameter.

The value of the `fields` parameter **MUST** be a comma-separated (U+002C COMMA, ",") list that refers to the name(s) of the fields to 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.

```
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 must be percent-encoded, per the requirements in RFC 3986 <http://tools.ietf.org/html/rfc3986#section-3.4>.

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.

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 association 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.

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 is reserved for pagination. Servers and clients **SHOULD** use this key for pagination operations.

> Note: JSON:API is agnostic about the pagination strategy used by a server. Effective pagination strategies include (but are not limited to): page-based, offset-based, and cursor-based. The `page` query parameter can be used as a basis for any of these strategies. For example, a page-based strategy might use query parameters such as `page[number]` and `page[size]`, an offset-based strategy might use `page[offset]` and `page[limit]`, while a cursor-based strategy might use `page[cursor]`.

> Note: The example query parameters above use unencoded `[` and `]` characters simply for readability. In practice, these characters must be percent-encoded, per the requirements in RFC 3986 <http://tools.ietf.org/html/rfc3986#section-3.4>.

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

Filtering

The `filter` query parameter is reserved for filtering data. Servers and clients **SHOULD** use this key for filtering operations.

> Note: JSON:API is agnostic about the strategies supported by a server. The `filter` query parameter can be used as the basis for any number of filtering strategies.

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.

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.

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 <http://tools.ietf.org/html/rfc4122.html>].

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.

Responses

201 Created

If a `POST` request did not include a Client-Generated ID and the requested resource has been created successfully, the server **MUST** return a `201 Created` status code.

The response **SHOULD** include a `Location` header identifying the location of the newly created resource.

The response **MUST** also include a document that contains the primary resource created.

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"
    }
  }
}
```

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.

204 No Content

If a `POST` request *did* include a Client-Generated ID and the requested resource has been created successfully, 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.

> Note: If a `204` response is received the client should consider the resource object sent in the request to be accepted by the server, as if the server had returned it back in a `201` response.

403 Forbidden

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

404 Not Found

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

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.

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` <http://tools.ietf.org/html/rfc7231>.

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"
    }
  }
}
```

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."
    }
  }
}
```

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.

Responses

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.

200 OK

If a server accepts an update but also changes the resource(s) in ways other than those specified by the request (for example, updating the `updated-at` attribute or a computed `sha`), it **MUST** return a `200 OK` response. The response document **MUST** include a representation of the updated resource(s) as if a `GET` request was made to the request URL.

A server **MUST** return a `200 OK` status code if an update is successful, the client's current fields remain up to date, and the server responds only with top-level meta data. In this case the server **MUST NOT** include a representation of the updated resource(s).

204 No Content

If an update is successful and the server doesn't update any fields besides those provided, 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.

403 Forbidden

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

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.

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` and `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.

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` <http://tools.ietf.org/html/rfc7231>.

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).

Updating To-One Relationships

A server **MUST** respond to `PATCH` requests to a URL from a to-one relationship link as described below.

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.

Updating To-Many Relationships

A server **MUST** respond to `PATCH` , `POST` , and `DELETE` requests to a URL from a to-many relationship link as described below.

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 can not 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.

> 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.

Responses

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.

204 No Content

A server **MUST** return a `204 No Content` status code if an update is successful and the representation of the resource in the request matches the result.

> 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.

200 OK

If a server accepts an update but also changes the targeted relationship(s) in other ways than those specified by the request, it **MUST** return a `200 OK` response. The response document **MUST** include a representation of the updated relationship(s).

A server **MUST** return a `200 OK` status code if an update is successful, the client's current data remain up to date, and the server responds only with top-level meta data. In this case the server **MUST NOT** include a representation of the updated relationship(s).

403 Forbidden

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

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` <http://tools.ietf.org/html/rfc7231>.

Deleting Resources

An individual resource can be *deleted* by making a `DELETE` request to the resource's URL:

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

Responses

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.

204 No Content

A server **MUST** return a `204 No Content` status code if a deletion request is successful and no content is returned.

200 OK

A server **MUST** return a `200 OK` status code if a deletion request is successful and the server responds with only top-level meta data.

404 NOT FOUND

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

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` <http://tools.ietf.org /html/rfc7231>.

Query Parameters

Implementation specific query parameters **MUST** adhere to the same constraints as member names with the additional requirement that they **MUST** contain at least one non a-z character (U+0061 to U+007A). It is **RECOMMENDED** that a U+002D HYPHEN-MINUS, "-", U+005F LOW LINE, "_", or capital letter is used (e.g. camelCasing).

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

Note: This is to preserve the ability of JSON:API to make additive additions to standard query parameters without conflicting with existing implementations.

Errors

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.

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:

- `id` : a unique identifier for this particular occurrence of the problem.
- `links` : a links object containing the following members:
- `about` : a link that leads to further details about this particular occurrence of the problem.
- `status` : the HTTP status code applicable to this problem, expressed as a string value.
- `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 source of the error, optionally including any of the following members:
- `pointer` : a JSON Pointer [RFC6901 <https://tools.ietf.org/html/rfc6901>] to the associated entity in the request document [e.g. `"/data"` for a primary data object, or `"/data/attributes/title"` for a specific attribute].
- `parameter` : a string indicating which URI query parameter caused the error.
- `meta` : a meta object containing non-standard meta-information about the error.