TABLE OF CONTENTS

Examples

This page contains additional examples of how to apply various parts of the specification.

Sparse Fieldsets

Examples of how sparse fieldsets <http://jsonapi.org/format/#fetching-sparse-fieldsets> work.

Basic request:

```
GET /articles?include=author HTTP/1.1
```

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

{
  "data": [{
    "type": "articles",
    "id": "1",
    "attributes": {
      "title": "JSON API paints my bikeshed!",
      "body": "The shortest article. Ever.",
      "created": "2015-05-22T14:56:29.000Z",
      "updated": "2015-05-22T14:56:28.000Z"
    },
    "relationships": {
      "author": {
        "data": {"id": "42", "type": "people"}
      }
    }
  }],
  "included": [
    {
      "type": "people",
      "id": "42",
      "attributes": {
        "name": "John",
        "age": 80,
        "gender": "male"
      }
    }
  ]
}
```

Error Objects

Request with `fields` parameter:

A Basic Error Object

Multiple Errors

```
GET /articles?include=author&fields[articles]=title,body,author&fields[people]=name HTTP/1.1
```

> Note: The above example URI shows unencoded `[` and `]` characters simply for readability. In practice, these characters must be percent-encoded, as noted in the base specification.

Here we want `articles` objects to have fields `title`, `body` and `author` only and `people` objects to have `name` field only.

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

{
  "data": [{
    "type": "articles",
    "id": "1",
    "attributes": {
      "title": "JSON API paints my bikeshed!",
      "body": "The shortest article. Ever."
    },
    "relationships": {
      "author": {
        "data": {"id": "42", "type": "people"}
      }
    }
  }],
  "included": [
    {
      "type": "people",
      "id": "42",
      "attributes": {
        "name": "John"
      }
    }
  ]
}
```

Pay attention to the fact that you have to add a relationship name both in `include` and `fields` (since relationships are fields too), otherwise you'll get:

```
GET /articles?include=author&fields[articles]=title,body&fields[people]=name HTTP/1.1
```

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

{
  "data": [{
    "type": "articles",
    "id": "1",
    "attributes": {
      "title": "JSON API paints my bikeshed!",
      "body": "The shortest article. Ever."
    }
  }],
  "included": [
    {
      "type": "people",
      "id": "42",
      "attributes": {
        "name": "John"
      }
    }
  ]
}
```

Note: The above example URI shows unencoded `[` and `]` characters simply for readability. In practice, these characters must be percent-encoded, as noted in the base specification.

Pagination Links

Example of a page-based strategy on how to add pagination links <http://jsonapi.org/format/#fetching-pagination>.

TABLE OF CONTENTS

```
GET /articles?page[number]=3&page[size]=1 HTTP/1.1
```

Advanced source Usage

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

{
  "meta": {
    "total-pages": 13
  },
  "data": [
    {
      "type": "articles",
      "id": "3",
      "attributes": {
        "title": "JSON API paints my bikeshed!",
        "body": "The shortest article. Ever.",
        "created": "2015-05-22T14:56:29.000Z",
        "updated": "2015-05-22T14:56:28.000Z"
      }
    }
  ],
  "links": {
    "self": "http://example.com/articles?page[number]=3&page[size]=1",
    "first": "http://example.com/articles?page[number]=1&page[size]=1",
    "prev": "http://example.com/articles?page[number]=2&page[size]=1",
    "next": "http://example.com/articles?page[number]=4&page[size]=1",
    "last": "http://example.com/articles?page[number]=13&page[size]=1"
  }
}
```

Note: The above example URIs show unencoded `[` and `]` characters simply for readability. In practice, these characters must be percent-encoded, as noted in the base specification.

Note: Putting a property like `"total-pages"` in `"meta"` can be a convenient way to indicate to clients the total number of pages in a collection (as opposed to the `"last"` link, which simply gives the URI of the last page). However, all `"meta"` values are implementation-specific, so you can call this member whatever you like (`"total"`, `"count"`, etc.) or not use it at all.

TABLE OF CONTENTS

Error Objects

Examples of how error objects <http://jsonapi.org/format/#error-objects> work.

A Basic Error Object

In the response below, the server is indicating that it encountered an error while creating/updating the resource, and that this error was caused by an invalid `"first-name"` attribute:

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/vnd.api+json

{
  "errors": [
    {
      "status": "422",
      "source": { "pointer": "/data/attributes/first-name" },
      "title":  "Invalid Attribute",
      "detail": "First name must contain at least three characters."
    }
  ]
}
```

Every member in an error object is optional, but all help the client by providing extra details.

The `source` member is used to indicate <https://tools.ietf.org/html/rfc6901> which part of the request document caused the error.

The `title` and `detail` members are similar, but `detail` is specific to this occurrence of the problem, whereas `title` is more generic.

The `status` member represents the HTTP status code associated with the problem. It's very helpful when multiple errors are returned at once (see below), as the HTTP response itself can only have one status code. However, it can also be useful for single errors, to save clients the trouble of consulting the HTTP headers, or for using JSON API over non-HTTP protocols, which may be officially supported in the near future.

TABLE OF CONTENTS

Multiple Errors

When multiple errors occur in response to a single request, the server can simply add each error to the `errors` array:

```
HTTP/1.1 400 Bad Request
Content-Type: application/vnd.api+json

{
  "errors": [
    {
      "status": "403",
      "source": { "pointer": "/data/attributes/secret-powers" },
      "detail": "Editing secret powers is not authorized on Sundays."
    },
    {
      "status": "422",
      "source": { "pointer": "/data/attributes/volume" },
      "detail": "Volume does not, in fact, go to 11."
    },
    {
      "status": "500",
      "source": { "pointer": "/data/attributes/reputation" },
      "title": "The backend responded with an error",
      "detail": "Reputation service not responding after three requests."
    }
  ]
}
```

The only uniqueness constraint on error objects is the `id` field. Thus, multiple errors on the same attribute can each be given their own error object. The example below shows multiple errors on the `"first-name"` attribute:

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/vnd.api+json

{
  "errors": [
    {
      "source": { "pointer": "/data/attributes/first-name" },
      "title": "Invalid Attribute",
      "detail": "First name must contain at least three characters."
    },
    {
      "source": { "pointer": "/data/attributes/first-name" },
      "title": "Invalid Attribute",
      "detail": "First name must contain an emoji."
    }
  ]
}
```

Note: in the responses above with a 422 status code, `400 Bad Request` would also be acceptable. (More details. <http://stackoverflow.com/a/20215807/1261879>) JSON API doesn't take a position on 400 vs. 422.

Error Codes

The `code` member of an error object contains an application-specific code representing the type of problem encountered. `code` is similar to `title` in that both identify a general type of problem (unlike `detail`, which is specific to the particular instance of the problem), but dealing with `code` is easier programatically, because the "same" `title` may appear in different forms due to localization.

For the example below, imagine the API docs specifed the following mapping:

Code	Problem
123	Value too short
225	Password lacks a letter, number, or punctuation character
226	Passwords do not match
227	Password cannot be one of last five passwords

Multiple errors on `"password"` attribute, with error `code`:

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/vnd.api+json

{
  "jsonapi": { "version": "1.0" },
  "errors": [
    {
      "code":   "123",
      "source": { "pointer": "/data/attributes/first-name" },
      "title":  "Value is too short",
      "detail": "First name must contain at least three characters."
    },
    {
      "code":   "225",
      "source": { "pointer": "/data/attributes/password" },
      "title": "Passwords must contain a letter, number, and punctuation character.",
      "detail": "The password provided is missing a punctuation character."
    },
    {
      "code":   "226",
      "source": { "pointer": "/data/attributes/password" },
      "title": "Password and password confirmation do not match."
    }
  ]
}
```

Notice that this response includes not only the `errors` top-level member, but the `jsonapi` top-level member. Error responses may not contain the top-level `data` member, but can include all the other top-level members JSON API defines.

Also, notice that the third error object lacks a `detail` member (perhaps for security). Again, all error object members are optional.

Advanced source Usage

In the example below, the user is sending an invalid JSON API request, because it's missing the `data` member:

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

{ "datum": [ ] }
```

Therefore, the server responds:

```
HTTP/1.1 422 Unprocesssable Entity
Content-Type: application/vnd.api+json

{
  "errors": [
    {
      "source": { "pointer": "" },
      "detail":  "Missing `data` Member at document's top level."
    }
  ]
}
```

It uses `source` to point to the top-level of the document (`""`). (Pointing to “/” would be an appropriate reference to the string `"some value"` in the request document `{"": "some value"}` . Pointing to `"/data"` would be invalid because the request document did not have a value at `"/data"` , and `source` is always given with reference to the request document.)

If the server cannot parse the request as valid JSON, including `source` doesn’t make sense (because there’s no JSON document for `source` to refer to). Here’s how the server might respond to an invalid JSON document:

```
{
  "errors": [{
    "status": "400",
    "detail": "JSON parse error - Expecting property name at line 1 column 2 (char 1)."
  }]
}
```

Invalid Query Parameters

The `source` member can also be used to indicate that the error originated from a problem with a URI query parameter, like so:

```
GET /api/posts/1?include=author HTTP/1.1
```

```
HTTP/1.1 400 Bad Request
Content-Type: application/vnd.api+json

{
  "errors": [
    {
      "source": { "parameter": "include" },
      "title":  "Invalid Query Parameter",
      "detail": "The resource does not have an `author` relationship path."
    }
  ]
}
```

In most cases, JSON API requires the server to return an error when it encounters an invalid value for a JSON API–defined query parameter. However, for API-specific query parameters (i.e. those not defined by JSON API), a server may choose to ignore an invalid parameter and have the request succeed, rather than respond with an error. API-specific query parameters must contain one non a-z character. <http://jsonapi.org/format/#query-parameters>

Other examples of invalid parameters include: `?felds[people]=` (invalid parameter name; should be `fields[people]`) and `?redirect_to=http%3A%2F%2Fwww.owasp.org` (invalid parameter, in this case, a phishing attack), etc.