用JSON构建API的标准指南

极客学院出版

前言

JSON API设计用来最小化请求的数量，以及客户端与服务器间传输的数据量。在高效实现的同时，无需牺牲可读性、灵活性和可发现性。

如果你和你的团队曾经争论过使用什么方式构建合理 JSON 响应格式， 那么 JSON API 就是你的 anti-bikeshedding 武器。

通过遵循共同的约定，可以提高开发效率，利用更普遍的工具，可以是你更加专注于开发重点：你的程序。

基于 JSON API 的客户端还能够充分利用缓存，以提升性能，有时甚至可以完全不需要网络请求。

下面是一个使用 JSON API 发送响应（response）的示例：

```
{
  "links": {
    "posts.author": {
      "href": "http://example.com/people/{posts.author}",
      "type": "people"
    },
    "posts.comments": {
      "href": "http://example.com/comments/{posts.comments}",
      "type": "comments"
    }
  },
  "posts": [{
    "id": "1",
    "title": "Rails is Omakase",
    "links": {
      "author": "9",
      "comments": [ "5", "12", "17", "20" ]
    }
  }]
}
```

顶级的 `"links"` 部分是可选的。除去 `"links"` 部分，此响应看起来非常接近使用已经存在的 API 构建的响应。

JSON API 不仅可以用来构建响应，还包括创建和更新资源。

MIME 类型

JSON API 已经在 IANA 机构完成注册。 它的 MIME 类型是 application/vnd.api+json (http://www.iana.org/assignments/media-types/application/vnd.api+json) 。

目录

1

介绍

JSON API 是数据交互规范，用以定义客户端如何获取与修改资源，以及服务器如何响应对应请求。

JSON API设计用来最小化请求的数量，以及客户端与服务器间传输的数据量。在高效实现的同时，无需牺牲可读性、灵活性和可发现性。

JSON API需要使用JSON API媒体类型(application/vnd.api+json (http://www.iana.org/assignments/media-types/application/vnd.api+json)) 进行数据交互。

JSON API服务器支持通过GET方法获取资源。而且必须独立实现HTTP POST, PUT和DELETE方法的请求响应，以支持资源的创建、更新和删除。

JSON API服务器也可以选择性支持HTTP PATCH方法 [RFC5789 (http://tools.ietf.org/html/rfc5789)]和JSON Patch格式 [RFC6902 (http://tools.ietf.org/html/rfc6902)]，进行资源修改。JSON Patch支持是可行的，因为理论上来说，JSON API通过单一JSON 文档，反映域下的所有资源，并将JSON文档作为资源操作介质。在文档顶层，依据资源类型分组。每个资源都通过文档下的唯一路径辨识。

规则约定

文档中的关键字， "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" 依据RFC 2119 [RFC2119 (http://tools.ietf.org/html/rfc2119)]规范解释。

2

文档结构

这一章节描述JSON API文档结构，通过媒体类型 application/vnd.api+json (http://www.iana.org/assignments/media-types/application/vnd.api+json) 标示。JSON API文档使用javascript 对象（JSON）[RFC4627 (http://tools.ietf.org/html/rfc4627)]定义。

尽管同种媒体类型用以请求和响应文档，但某些特性只适用于其中一种。差异在下面呈现。

Top Level

JSON 对象必须位于每个JSON API文档的根级。这个对象定义文档的“top level”。

文档的top level必须包含请求资源或者请求资源集合的实例 (即主要资源)。

主要资源应该以资源类型或者通用键 "data" 索引.

- "meta" : 资源的元信息，比如分页.
- "links" : 扩展资源关联URLs的URL模板.
- "linked" : 资源对象集合，按照类型分组，链接到主要资源或彼此（即链接资源）

资源表示

这一章节描述JSON API文档如何表示资源。适用于主要资源和链接资源。

个体资源表示

个体资源使用单一“资源对象”（如下描述）或者包含资源ID（如下描述）的字符串表示。

The following post is represented as a resource object: 下面的post表示一个资源对象：

```
{
  "posts": {
    "id": "1",
    // ... attributes of this post
  }
}
```

这个post用ID简单地表示：

```
{
  "posts": "1"
}
```

资源集合表示

任意数量资源的集合应该使用资源对象数组，或者IDs数组，或者一个简单的”集合对象“表示。

下面这个post使用资源对象数组表示：

```
{
  "posts": [{
    "id": "1"
    // ... attributes of this post
  }, {
    "id": "2"
    // ... attributes of this post
  }]
}
```

这个posts使用IDs数组表示：

```
{
  "posts": ["1", "2"]
}
```

这些comments使用单一集合对象表示：

```
{
  "comments": {
    "href": "http://example.com/comments/5,12,17,20",
    "ids": [ "5", "12", "17", "20" ],
    "type": "comments"
  }
}
```

多资源对象

多个资源对象有相同的内部结构，不管他们表示主要资源还是链接资源。

下面是一个可能出现在文档中的post（即”posts"类型的一个资源）：

```
{
  "posts": {
    "id": "1",
    "title": "Rails is Omakase"
```

```
  }
}
```

在上面这个例子中，post的资源对象比较简单：

```
//...
  {
    "id": "1",
    "title": "Rails is Omakase"
  }
//...
```

这一章节专注于资源对象，在完整JSON API文档上下文环境之外。

资源属性

资源对象有四个保留字:

- `"id"`
- `"type"`
- `"href"`
- `"links"`

资源对象中的其它键表示一个“属性”。一个属性值可以是任何JSON值。

资源 IDs

Each resource object SHOULD contain a unique identifier, or ID, when available. IDs MAY be assigned by the server or by the client, as described below, and SHOULD be unique for a resource when scoped by its type. An ID SHOULD be represented by an `"id"` key and its value MUST be a string which SHOULD only contain alphanumeric characters, dashes and underscores.

每一个资源对象应该有一个唯一标示符，或者ID。如下所示，IDs可由服务器或者客户端指定，and SHOULD be unique for a resource when scoped by its type. ID应该使用 `"id"` 键表示，值必须是字符串，且只包含字母，数字，连字符和下划线。

URL 模板可以使用IDs来获取关联资源，如下所示。

在特殊场景下，客户端与服务器之间的唯一标识符信息非必要，JSON API允许缺省IDs。

资源类型

每个资源对象的类型通常由它所在的上下文环境决定。如上面讨论，资源对象在文档中通过类型索引。

每一个资源对象可能包含 "type" 键来显示指定类型。

当资源的类型在文档中未声明时， "type" 键不可缺省。

资源 URLs

每一个资源的URL可能使用 "href" 键声明。资源URLs应该由服务器指定，因此通常包含在响应文档中。

```
//...
  [{
    "id": "1",
    "href": "http://example.com/comments/1",
    "body": "Mmmmmakase"
  }, {
    "id": "2",
    "href": "http://example.com/comments/2",
    "body": "I prefer unagi"
  }]
//...
```

服务器对特定URL GET 请求，响应内容必须包含资源。

通常在响应文档的根层级声明URL 模板会更高效，而不是在每一个资源对象内声明独立的URLs。

资源关联

"links" 键的值是一个表示链接资源的JSON对象，通过关联名索引。

举例来说，下面的post与一个 author 和一个 comments 集合相关联：

```
//...
  {
    "id": "1",
    "title": "Rails is Omakase",
    "links": {
      "author": "9",
      "comments": [ "5", "12", "17", "20" ]
    }
```

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

单对象关联

单对象关联必须使用上面所述单资源形式的一种来表示。

举例来说，下面的post与一个author相关联，通过ID标示：

```
//...
  {
    "id": "1",
    "title": "Rails is Omakase",
    "links": {
      "author": "17"
    }
  }
//...
```

下面是一个示例，链接的author用一个资源对象表示：

```
//...
  {
    "id": "1",
    "title": "Rails is Omakase",
    "links": {
      "author": {
        "href": "http://example.com/people/17",
        "id": "17",
        "type": "people"
      }
    }
  }
//...
```

空白的单对象关联应该用 `null` 值表示。举例来说，下面的post没有关联author:

```
//...
  {
    "id": "1",
    "title": "Rails is Omakase",
    "links": {
      "author": null
    }
  }
//...
```

多对象关联

多对象关联必须使用上述资源集合形式的一种来表示。

举例来说，下面的post与多个comments关联，通过IDs标示：

```
//...
  {
    "id": "1",
    "title": "Rails is Omakase",
    "links": {
      "comments": [ "5", "12", "17", "20" ]
    }
  }
//...
```

这是一个使用集合对象链接的comments数组：

```
//...
  {
    "id": "1",
    "title": "Rails is Omakase",
    "links": {
      "comments": {
        "href": "http://example.com/comments/5,12,17,20",
        "ids": [ "5", "12", "17", "20" ],
        "type": "comments"
      }
    }
  }
//...
```

空白的多对象关联应该使用空数组表示。举例来说，下面的post没有comments：

```
//...
  {
    "id": "1",
    "title": "Rails is Omakase",
    "links": {
      "comments": []
    }
  }
//...
```

集合对象

“集合对象”包含一个或多个元素：

- `"ids"` – 关联资源的IDs数组。
- `"type"` – 资源类型
- `"href"` – 关联资源的URL（适用于响应文档）。

提供包含`href`属性集合对象的服务器，必须响应特定URL `GET` 请求，响应内容包含资源对象集合的关联资源。

URL模板

顶层的 `"links"` 对象可用来声明URL模板，从而依据资源对象类型获取最终URLs。

举例说明:

```
{
  "links": {
    "posts.comments": "http://example.com/comments?posts={posts.id}"
  },
  "posts": [{
    "id": "1",
    "title": "Rails is Omakase"
  }, {
    "id": "2",
    "title": "The Parley Letter"
  }]
}
```

在这个示例中，请求`http://example.com/comments?posts=1` 将会得到`"Rails is Omakase"`的comments，请求`http://example.com/comments?posts=2` 将会得到 `"The Parley Letter"` 的comments.

下面是另外一个示例:

```
{
  "links": {
    "posts.comments": "http://example.com/comments/{posts.comments}"
  },
  "posts": [{
    "id": "1",
```

```
    "title": "Rails is Omakase",
    "links": {
      "comments": [ "1", "2", "3", "4" ]
    }
  }]
}
```

在这个示例中，处理每个post `"links"` 区块内的特定数组，以扩展 `posts.comments` 变量。URI模板规范 [RFC6570 (https://tools.ietf.org/html/rfc6570)]声明默认处理方式，使用%编码（即 `encodeURIComponent()` javascript原生方法）编码每一个元素，然后用逗号连接。在这个示例中，请求 `http://example.com/comments/1,2,3,4` ，将会获取一个 `comments` 列表。

顶层 `"links"` 对象具有以下行为:

- 每个键使用点分隔路径，指向重复的关联。路径以特定资源类型名开头，遍历相关的资源。举例来 说， `"posts.comments"` 指向每个 `"posts"` 对象的 `"comments"` 关联.
- 每个键的值作为URL模板处理。
- 每个path指向的资源，就像是使用实际指定的非URL值扩展URL模板形成的关联。

这是另外一个使用单对象关联的示例:

```
{
  "links": {
    "posts.author": "http://example.com/people/{posts.author}"
  },
  "posts": [{
    "id": "1",
    "title": "Rails is Omakase",
    "links": {
      "author": "12"
    }
  }, {
    "id": "2",
    "title": "The Parley Letter",
    "links": {
      "author": "12"
    }
  }, {
    "id": "3",
    "title": "Dependency Injection is Not a Virtue",
    "links": {
      "author": "12"
    }
```

```
 }]
}
```

这个实例中，三个posts指向author的URL都为 `http://example.com/people/12` .

顶层URL模板允许指定关联作为IDs，但是不要求客户端硬编码来获取URLs的信息.

注意：为防止冲突，单独资源对象的 `links` 对象优先级高于顶层的 `links` 对象。

复合文档

为减少HTTP请求，响应需要返回所请求的主要资源，同时可以选择性的包含链接资源。这样的响应称作“复合文档”。

在复合文档中，链接资源必须作为资源对象，包含在文档顶层 `"linked"` 对象中，依据类型，组合到不同数组中。

每个关联的类型，可以在资源层级，或顶层 `"links"` 对象层级，使用 `"type"` 键指定。能够辅助客户端查询链接资源对象。

```
{
 "links": {
  "posts.author": {
   "href": "http://example.com/people/{posts.author}",
   "type": "people"
  },
  "posts.comments": {
   "href": "http://example.com/comments/{posts.comments}",
   "type": "comments"
  }
 },
 "posts": [{
  "id": "1",
  "title": "Rails is Omakase",
  "links": {
   "author": "9",
   "comments": [ "1", "2", "3" ]
  }}, {
  "id": "2",
  "title": "The Parley Letter",
  "links": {
   "author": "9",
   "comments": [ "4", "5" ]
 }}, {
  "id": "1",
```

```
    "title": "Dependency Injection is Not a Virtue",
    "links": {
      "author": "9",
      "comments": [ "6" ]
    }
  }],
  "linked": {
    "people": [{
      "id": "9",
      "name": "@d2h"
    }],
    "comments": [{
      "id": "1",
      "body": "Mmmmmakase"
    }, {
      "id": "2",
      "body": "I prefer unagi"
    }, {
      "id": "3",
      "body": "What's Omakase?"
    }, {
      "id": "4",
      "body": "Parley is a discussion, especially one between enemies"
    }, {
      "id": "5",
      "body": "The parsley letter"
    }, {
      "id": "6",
      "body": "Dependency Injection is Not a Vice"
    }]
  }
}
```

这种处理方式，保证随每个响应返回每个文档的单例，即使当相同的文档被多次引用时（这个实例中三个posts的author）。沿着这种方式，如果主要文档链接到另外的主要或链接文档，在 `"linked"` 对象中也不应该重复。

3

URLs

关联文档

确定API的URL结构时，考虑把所有的资源放置于单一“关联文档"是有用的，在关联文档中，每个资源都被分配唯一路径。在文档顶层，资源依据类型分组。在分类资源集合中，单独资源通过ID索引。其属性和links，依据资源对象结构，唯一分配。

关联文档的概念，用于为资源及资源关系确定合适的URLs。重要的一点，出于不同目标和限制，用于传输资源的不同文档中，关联文档的结构有轻微差异。例如，在关联文档中的资源集当做集合处理，因为元素必须通过ID访问，在传输文档中的资源集当做数组处理，因为顺序比较重要。

资源集合URLs

资源集合的URL应该依据资源类型确定。

例如，”photos”类型的资源集合应该使用这种URL:

```
/photos
```

单独资源URLs

资源集应该作为集合，依据资源ID索引。单独资源的URL通过为集合URL添加资源ID生成。

例如，ID为 `"1"` 的photo使用这种URL:

```
/photos/1
```

多个单独资源的URL通过为集合URL添加逗号分隔的资源IDs列表生成。

例如，IDs为 `"1"` ，`"2"` ，`"3"` 的photos使用这种URL:

```
/photos/1,2,3
```

替代性URLs

资源的替代性URLs可以选择在响应中指定，或者通过 `"href"` 或URL模板指定。

关联 URLs

添加 /links/<relationship-name> 到资源URL后面，即得到访问特定资源的关联资源URL。相对路径与资源对象内部结构保持一致。

例如，photo的comments链接集使用这种URL:

```
/photos/1/links/comments
```

A photo's reference to an individual linked photographer will have the URL: photo的photographer链接使用这种URL:

```
/photos/1/links/photographer
```

服务器使用单独资源响应单对象关联，使用资源集合响应多对象关联。

4

资源获取

资源，或者资源集合，通过向URL发出 GET 请求获取。

响应内容可以使用如下所示的特点，进一步细化。

过滤

服务器可以选择性支持，依据指定标准进行资源过滤。

通过向资源集合的基准URL添加过滤参数，来支持资源过滤。

例如，下面是请求与特定post关联的所有comments:

```
GET /comments?posts=1
```

使用这种方案，单一请求可以使用多过滤器:

```
GET /comments?posts=1&author=12
```

这种规范仅支持基于严格匹配的资源过滤。API允许使用的额外过滤器应该在它的侧写中指定。

内链资源

服务器可以选择性支持，返回包含主要资源和链接资源对象的复合文档。

默认情况下，后端返回链接主要资源的资源对象。

后端也可以基于请求中 include 的参数，支持自定义链接资源。参数应该指定一个或者多个，相对于主要资源的相对路径。如果指定参数值，只有请求的链接资源，应该随主要资源返回。

例如，comments可以通过post请求:

```
GET /posts/1?include=comments
```

为请求链接到其他资源的资源，需要指定每个关联的点分隔路径。

```
GET /posts/1?include=comments.author
```

注意：对 comments.author 的请求，在响应中不应该自动包含 comments 资源（尽管comments也需要显式查询，以获取authors响应请求）。

多链接资源可以使用点分隔列表请求:

```
GET /posts/1?include=author,comments,comments.author
```

稀疏字段

服务器可以选择性支持，仅返回资源对象的指定字段。

后端可以基于 `fields` 参数，以支持返回主要资源的指定字段。

```
GET /people?fields=id,name,age
```

后端可以基于 `fields[TYPE]` 参数，以支持返回任意类型资源的特定字段。

```
GET /posts?include=author&fields[posts]=id,title&fields[people]=id,name
```

若没有指定类型对象的字段，或者后端不支持 `field` 或 `fields[TYPE]` 参数，后端会默认返回资源对象的所有字段。

后端可以选择总是返回有限的，未指定的字段集，例如 `id` or `href`.

注意： `fields` 和 `fields[TYPE]` 不能混合使用。如果使用后者，那么必须与主要资源类型同时使用。

排序

服务器可以选择性支持，基于特定标准对资源集合排序。

后端基于 `sort` 参数，以支持主要资源类型的排序。

```
GET /people?sort=age
```

后端支持多字段排序，将 `sort` 值设置为点分隔值即可。排序标准用以获取特定顺序。

```
GET /people?sort=age,name
```

默认排序方式为升序排序。任意排序字段，使用 `-` 前缀指定降序排序。

```
GET /posts?sort=-created,title
```

上面的示例应该首先返回最新的posts。同一天创建的posts，依据title值进行字母升序排列。

后端基于 `sort[TYPE]` 参数，以支持对任意资源类型排序。

```
GET /posts?include=author&sort[posts]=-created,title&sort[people]=name
```

如果没有指定排序方式，或者后端不支持 `sort` 和 `sort[TYPE]`，后端将会返回使用重复算法排序的资源对象。换言之，资源应该总是以相同顺序返回，即使排序规则没有指定。

注意： `sort` 和 `sort[TYPE]` 不能混用。如果使用后者，必须与主要资源一同使用。

5

创建，更新，删除资源

服务器可能支持资源获取，创建，更新和删除。

服务器允许单次请求，更新多个资源，如下所述。多个资源更新必须完全成功或者失败，不允许部分更新成功。

任何包含内容的请求，必须包含 `Content-Type:application/vnd.api+json` 请求头。

创建资源

支持资源创建的服务器，必须支持创建单独的资源，可以选择性支持一次请求，创建多个资源。

向表示待创建资源所属资源集的URL，发出 `POST` 请求，创建一个或多个资源。

创建单独资源

创建单独资源的请求必须包含单一主要资源对象。

例如，新photo可以通过如下请求创建：

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

{
  "photos": {
    "title": "Ember Hamster",
    "src": "http://example.com/images/productivity.png"
  }
}
```

创建多个资源

创建多个资源的请求必须包含主要主要资源集合。

例如，多个photos通过如下请求创建：

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

{
  "photos": [{
    "title": "Ember Hamster",
```

```
    "src": "http://example.com/images/productivity.png"
  }, {
    "title": "Mustaches on a Stick",
    "src": "http://example.com/images/mustaches.png"
  }]
}
```

响应

201 状态码

服务器依据[HTTP semantics](http://tools.ietf.org/html/draft-ietf- httpbis-p2-semantics-22#section-6.3)规范，响应成功的资源创建请求。

当一个或多个资源创建成功，服务器返回 201 Created 状态码。

响应必须包含 Location 头，用以标示请求创建所有资源的位置。

如果创建了单个资源，且资源对象包含 href 键， Location URL必须匹配 href 值。

响应必须含有一个文档，用以存储所创建的主要资源。如果缺失，客户端则判定资源创建时，传输的文档未经修改。

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

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

其它响应

服务器可能使用其它HTTP错误状态码反映错误。客户端必须依据HTTP规范处理这些错误信息。如下所述，错误细节可能会一并返回。

客户端生成 IDs

请求创建一个或多个资源时，服务器可能接受客户端生成IDs。IDs必须使用 "id" 键来指定，其值必须正确生成，且为格式化的*UUID*。

例如：

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

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

更新资源

支持资源更新的服务器必须支持单个资源的更新，可以选择性的支持单次请求更新多个资源。

向表示单独资源或多个单独资源的URL发出 `PUT` 请求，即可进行资源更新。

更新单独资源

为更新单独资源，向表示资源的URL发出 `PUT` 请求。请求必须包含一个顶层资源对象。

例如：

```
PUT /articles/1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

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

更新多个资源

向表示多个单独资源（不是全部的资源集合）的URL发出 `PUT` 请求，即可更新多个资源。请求必须包含顶层资源对象集合，且每个资源具有 `“id"` 元素。

例如：

```
PUT /articles/1,2
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "articles": [{
    "id": "1",
    "title": "To TDD or Not"
  }, {
    "id": "2",
    "title": "LOL Engineering"
  }]
}
```

更新属性

要更新资源的一个或多个属性，主要资源对象应该只包括待更新的属性。资源对象缺省的属性将不会更新。

例如，下面的 `PUT` 请求，仅会更新article的 `title` 和 `text` 属性。

```
PUT /articles/1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

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

更新关联

更新单对象关联

单对象关联更新，可以在 `PUT` 请求资源对象中包含 `links` 键，从而与其它属性一起更新。

例如，下面的 `PUT` 请求将会更新article的 `title` 和 `author` 属性：

```
PUT /articles/1
Content-Type: application/vnd.api+json
```

```
Accept: application/vnd.api+json

{
  "articles": {
    "title": "Rails is a Melting Pot",
    "links": {
      "author": "1"
    }
  }
}
```

若要移除单对象关联，指定 `null` 作为值即可。

另外，单对象关联也可以通过它的关联URL访问。

向关联URL发出带有主要资源的 `POST` 请求，即可添加单对象关联。

例如：

```
POST /articles/1/links/author
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "people": "12"
}
```

向关联URL发出 `DELETE` 请求，即可删除单对象关联。例如：

```
DELETE /articles/1/links/author
```

更新多关联对象

更新多对象关联，可以在 `PUT` 请求中资源对象包含 `links` 键，从而与其它属性一起更新。

例如，下面 `PUT` 请求完全替换article的 `tags` 。

```
PUT /articles/1
Content-Type: application/vnd.api+json
Accept: application/vnd.api+json

{
  "articles": {
    "id": "1",
    "title": "Rails is a Melting Pot",
    "links": {
      "tags": ["2", "3"]
    }
```

```
  }
}
```

若要移除多对象关联，指定空数组 [] 为值即可。

在分布式系统中，完全替换一个数据集合并不总是合适。替换方案是允许单独的添加或移除关联。

为促进细化访问，多对象关联也可以通过关联URL访问。

向关联URL发出带有主要资源的 POST 请求，即可添加多对象关联。

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

{
  "comments": ["1", "2"]
}
```

向关联URL发出 DELETE 请求，即可删除多对象对象关联。例如：

```
DELETE /articles/1/links/tags/1
```

向关联URL发出 DELETE 请求，即可删除多个多对象对象关联。例如：

```
DELETE /articles/1/links/tags/1,2
```

响应

204 No Content

如果更新成功，且客户端属性保持最新，服务器必须返回 204 No Content 状态码。适用于 PUT 请求，以及仅调整links，不涉及其它属性的 POST ， DELETE 请求。

200 OK

如果服务器接受更新，但是在请求指定内容之外做了资源修改，必须响应 200 OK 以及更新的资源实例，像是向此URL发出 GET 请求。

其它响应

服务器使用其它HTTP错误状态码反映错误。客户端必须依据HTTP规范处理这些错误信息。如下所述，错误细节可能会一并返回。

资源删除

向资源URL发出 `DELETE` 请求即可删除单个资源。

```
DELETE /photos/1
```

服务器可以选择性的支持，在一个请求里删除多个资源。

```
DELETE /photos/1,2,3
```

响应

204 No Content

如果删除请求成功，服务器必须返回 `204 No Content` 状态码。

其它响应

服务器使用其它HTTP错误状态码反映错误。客户端必须依据HTTP规范处理这些错误信息。如下所述，错误细节可能会一并返回。

6

Errors

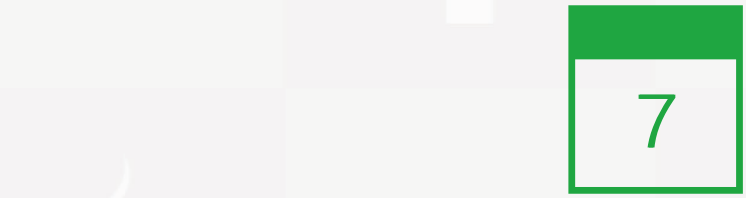
错误对象是特殊化的资源对象，可能在响应中一并返回，用以提供执行操作遭遇问题的额外信息。在在JSON API文档顶层，`"errors"`对应值即为错误对象集合，此时文档不应该包含其它顶层资源。

错误对象可能有以下元素：

- `"id"` – 特定问题的唯一标示符。
- `"href"` – 提供特定问题更多细节的URI。
- `"status"` – 适用于这个问题的HTTP状态码，使用字符串表示。
- `"code"` – 应用特定的错误码，以字符串表示。
- `"title"` – 简短的，可读性高的问题总结。除了国际化本地化处理之外，不同场景下，相同的问题，值是不应该变动的。
- `"detail"` – 针对该问题的高可读性解释。
- `"links"` – 可以在请求文档中取消应用的关联资源。
- `"path"` – 关联资源中相关属性的相对路径。在单资源或单类型资源中出现的问题，这个值才是合适的。

额外的元素可以在错误对象中指定。

实现接口可以选择使用其它的errors媒体类型。



7

PATCH Support

JSON API服务器可以选择性支持，遵循JSON Patch规范[RFC6902 (http://tools.ietf.org/html/rfc6902)]的HTTP `PATCH` 请求。上面提到使用 `POST` ，`PUT` 和 `DELETE` 进行的操作，JSON Patch都拥有等效操作。从这里开始， `PATCH` 请求发出的JSON Patch操作，都被简单的称作“ `PATCH` "操作。

`PATCH` 请求必须声明 `Content-Type:application/json-patch+json` 头。

`PATCH` 操作必须作为数组发送，以遵循JSON Patch规范。服务器可能会限制顶层数组类型，顺序和操作数量。

请求 URLs

每个 `PATCH` 请求的URL应该映射待更新的资源或关联。

`PATCH` 操作内的每个 `"path"` 应该相对于请求URL。请求URL和 `PATCH` 操作的 `"path"` 是附加的，合并后获取特定资源，集合，属性，或者关联目标。

`PATCH` 操作可以在API的根URL使用。此时， `PATCH` 操作的 `"path"` 必须包含完整的资源URL。API表示的任意资源都可以进行常规的"fire hose"更新。如上所述，服务器可能会限制类型，排序和批量操作数量。

创建资源with PATCH

要创建资源，执行 `"add"` 操作， `"path"` 指向对应资源集合的末尾 (`"/-"`)。 `"value"` 必须包含一个资源对象。

比如，新photo通过如下请求创建：

```
PATCH /photos
Content-Type: application/json-patch+json
Accept: application/json

[
  {
    "op": "add",
    "path": "/-",
    "value": {
      "title": "Ember Hamster",
      "src": "http://example.com/images/productivity.png"
    }
  }
]
```

更新属性with PATCH

要更新属性，执行 "replace" 操作， "path" 值指定属性名。

例如，下面请求只更新 /photos/1 的 src 值。

```
PATCH /photos/1
Content-Type: application/json-patch+json

[
 {
  "op": "replace",
  "path": "/src",
  "value": "http://example.com/hamster.png"
 }
]
```

更新关联with PATCH

要更新关联，向对应的关联URL发出合适的 PATCH 操作即可。

服务器可能支持更高层级的更新，像资源的URL（甚至是API的根URL）。如上所述，请求URL和每个操作的 "path" 是附加的，合并后获得特定关联URL目标。

关联更新with PATCH

要更新单对象关联，对指向关联的URL和 "path" 执行 "replace" 操作。

例如：下面请求更新article的 author ：

```
PATCH /article/1/links/author
Content-Type: application/json-patch+json

[
 {
  "op": "replace",
  "path": "/",
  "value": "1"
 }
]
```

要移除单对象关联，对关联执行 `remove` 操作。例如：

```
PATCH /article/1/links/author
Content-Type: application/json-patch+json

[
  {
    "op": "remove",
    "path": "/"
  }
]
```

更新多对象关联 with PATCH

尽管在 `GET` 响应中，多对象关联以JSON数组形式表示，但是更新时更像是集合。

要添加元素到多对象关联，执行指向关联URL的 `"add"` 请求。由于操作指向集合末尾， `"path"` 必须以 `"/-"` 结尾。

例如，考虑下面的 `GET` 请求：

```
GET /photos/1
Content-Type: application/vnd.api+json

{
  "links": {
    "comments": "http://example.com/comments/{comments}"
  },
  "photos": {
    "id": "1",
    "href": "http://example.com/photos/1",
    "title": "Hamster",
    "src": "images/hamster.png",
    "links": {
      "comments": [ "1", "5", "12", "17" ]
    }
  }
}
```

向 `PATCH` 请求执行 `add` 操作，即可将comment 30 转移到这个photo。

```
PATCH /photos/1/links/comments
Content-Type: application/json-patch+json
```

```
[
  {
    "op": "add",
    "path": "/-",
    "value": "30"
  }
]
```

要移除多对象关联，对指向关联对象的URL执行 "remove" 操作。因为操作目标是元素集合， "path" 必须以 "/<id>" 结尾。

比如，要移除photo的comment 5，执行 "remove" 操作：

```
PATCH /photos/1/links/comments
Content-Type: application/json-patch+json

[
  {
    "op": "remove",
    "path": "/5"
  }
]
```

删除资源with PATCH

要删除资源，对指向资源的 URL 和 "path" 执行 "remove" 操作。

例如，photo 1 能使用下面请求删除：

```
PATCH /photos/1
Content-Type: application/json-patch+json
Accept: application/vnd.api+json

[
  {
    "op": "remove",
    "path": "/"
  }
]
```

响应

204 No Content

若 PATCH 请求成功，客户端当前的属性保持最新，服务器必须响应 204 No Content 状态码。

200 OK

如果服务器接受更新，但是在请求指定内容之外做了资源修改，必须响应 200 OK 以及更新的资源实例。

服务器必须指定 Content-Type:application/json 头。响应内容必须包含JSON对象数组，每个对象必须遵循JSON API媒体类型(application/vnd.api+json)。数组中的响应对象必须有序，并且对应请求文档操作。

例如，一个请求以分离操作，创建两个photos。

```
PATCH /photos
Content-Type: application/json-patch+json
Accept: application/json

[
  {
    "op": "add",
    "path": "/-",
    "value": {
      "title": "Ember Hamster",
      "src": "http://example.com/images/productivity.png"
    }
  },
  {
    "op": "add",
    "path": "/-",
    "value": {
      "title": "Mustaches on a Stick",
      "src": "http://example.com/images/mustaches.png"
    }
  }
]
```

响应内容在数组中包含对应的JSON API文档。

```
HTTP/1.1 200 OK
Content-Type: application/json
```

```
[
 {
  "photos": [{
   "id": "123",
   "title": "Ember Hamster",
   "src": "http://example.com/images/productivity.png"
  }]
 }, {
  "photos": [{
   "id": "124",
   "title": "Mustaches on a Stick",
   "src": "http://example.com/images/mustaches.png"
  }]
 }
]
```

其它响应

当服务器执行 `PATCH` 请求时出现一个或多个问题，应该在响应中指定最合适的HTTP状态码。客户端依据HTTP规范解析错误。

服务器可以选择在第一个问题出现时，立刻终止 `PATCH` 操作，或者继续执行，遇到多个问题。例如，服务器可能多属性更新，然后返回在一个响应里返回多个校验问题。

当服务器单个请求遇到多个问题，响应中应该指定最通用可行的HTTP错误码。例如， `400 Bad Request` 适用于多个4xx errors， `500 Internal Server Error` 适用于多个5xx errors。

服务器可能会返回与每个操作对应的错误对象。服务器需要指定 `Content-Type:application/json` 头，响应体必须包含JSON对象数组，每个对象必须遵循JSON API媒体类型（ `application/vnd.api+json` ）。数组中的响应对象，必须是有序，且与请求文档中的操作相对应。每个响应对象应该仅包含error对象，当错误发生时，没有操作会完全成功。每个特定操作的错误码，应该在每个error对象 `"status"` 元素反映。

HTTP 缓存

服务器可能会使用遵循HTTP 1.1规范的HTTP 缓存头（ `ETag` ， `Last-Modified` ）。

layout: page

title: 扩展

{% include status.md %}

扩展 (页 43)

如果你想扩展 JSON API，你应该遵循 RFC 6906 (http://tools.ietf.org/html/rfc6906) 定义的 profile。 另请参阅 Mark Nottingham 写的这篇文章 (http://www.mnot.net/blog/2012/04/17/profiles) 。

`meta` 部分定义了 profile 的链接。

注意，在 RFC 规范中，profile：

> 不改变资源表示的语义定义本身，但让客户了解更多的语义(约束、规范、扩展)相关联的资源表示形式， 还有这些定义的媒体类型和可能的其他机制。

示例 (页 43)

例如，假设你想让你的 API 支持不同的分页设计，如基于游标。 你会制作某种信息页面在你的网站上，如 `http://api.example.com/profile` ， 然后会响应中包含 `meta` 键:

```
GET http://api.example.com/
```

```
{
  "meta": {
    "profile": "http://api.example.com/profile"
  },

  "posts": [{
    // 一份单独的文档
  }]
}
```

That document will de-reference to explain your link relations:

这份文档将解释链接之间的关系:

```
GET http://api.example.com/profile HTTP/1.1
```

```
HTTP/1.1 200 OK
Content-Type: text/plain
```

第 8 章 The Example.com API Profile

Example.com API 使用基于游标的分页。 它是这样工作的: 在想要的 `meta` 部分，它将返回一个 `cursors` 的关系（relation）, 其中包括 `after`， `before` 和 `limit`，用来描述该游标。 您可以使用 `href` 给出的 URI 模板来生成分页的 URIs。

```
"meta": {
  "cursors": {
    "after": "abcd1234",
    "before": "wxyz0987",
    "limit": 25,
    "href": "https://api.example.com/whatever{?after,before,limit}"
  }
}
```

9

Examples

Examples are excellent learning aids. The following projects implementing JSON API are divided into server- and client-side. The server-side is further divided by implementation language. If you'd like y our project listed, send a Pull Request (https://github.com/json-api/json-api) .

Client

JavaScript

- ember-data (https://github.com/emberjs/data) is one of the original exemplar implementations. T here is a custom adapter (https://github.com/daliwali/ember-json-api) to support json-api.
- backbone-jsonapi (https://github.com/guillaumervls/backbone-jsonapi) is a Backbone adapter fo r JSON API. Supports fetching Models & Collections from a JSON API source.

iOS

- jsonapi-ios (https://github.com/joshdholtz/jsonapi-ios) is a library for loading data from a JSON A PI datasource. Parses JSON API data into models with support for auto-linking of resources and custom model classes.

Server

PHP

- FriendsOfSymfony / FOSRestBundle (https://github.com/FriendsOfSymfony/FOSRestBundle/iss ues/452)

Node.js

- Fortune.js (http://fortunejs.com) is a framework built to implement json-api.

Ruby

- ActiveModel::Serializers (https://github.com/rails-api/active_model_serializers) is one of the origi nal exemplar implementations, but is slightly out of date at the moment.

- JsonApiClient (https://github.com/chingor13/json_api_client) attempts to give you a query building framework that is easy to understand (similar to ActiveRecord scopes)
- The rabl wiki (https://github.com/nesquena/rabl/wiki/Conforming-to-jsonapi.org-format) has a page describing how to emit conformant JSON.
- RestPack::Serializer (https://github.com/RestPack/restpack_serializer) implements the read elements of json-api. It also supports paging and side-loading.
- Oat (https://github.com/ismasan/oat#adapters) ships with a JSON API adapter.

Python

- Hyp (https://github.com/kalasjocke/hyp) is a library for creating json-api responses.
- SQLAlchemy-JSONAPI (https://github.com/coltonprovias/sqlalchemy-jsonapi) provides JSON API serialization for SQLAlchemy models.

Messages

- RestPack::Serializer provides examples (http://restpack-serializer-sample.herokuapp.com/) which demonstrate sample responses.

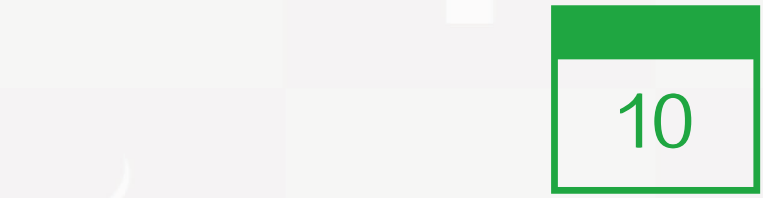
Related Tools

Ruby

- json-patch (https://github.com/guillec/json-patch) implementation of JSON Patch (rfc6902)
- hana (https://github.com/tenderlove/hana) implementation of the JSON Patch and JSON pointer spec

Node.js

- json-patch (https://www.npmjs.org/package/json-patch) implementation of JSON Patch (rfc6902)



10

FAQ

为什么 JSON API 还没有发布版?

一旦 JSON API 发布稳定版，它将保持向后兼容，它将遵守**永不删除，只是添加**的开发策略。 #46 (https://github.com/json-api/json-api/issues/46)

为什么不使用 HAL 规范?

有几个原因:

- HAL 递归嵌套子文档，而 JSON API 在顶层采用扁平化对象结构。意味着不同的对象引用相同的 “people”（例如，posts和comments的author）时，这种规范能够保证每个person document仅存在一个有效实例。
- 相似的，JSON API 使用 IDs 做链接，使从复合响应中缓存文档成为可能，仅当本地不存在对应文档，才会发出后续请求。如果幸运，甚至可以完全无需HTTP请求。
- HAL 是序列化格式，但完全未定义文档更新操作。JSON API 则仔细考虑如何更新已存在文档（依赖 PATCH 和 JSON PATCH），以及更新操作与GET请求返回复合文档交互方式。同时定义如何创建，删除文档，以及更新操作的 200,204 响应。

简单来说，JSON API 尝试格式化相似的，特殊的client-server通讯接口，使用 JSON 作为数据交换格式。专注于使用成熟的客户端来调用相关API，客户端能够缓存已经获取到的文档，避免再次请求已缓存信息。

JSON API 从大量实际项目所使用的库中抽象而出。同时定义请求/响应（HAL 未定义），以及对应数据交互格式。

如何获取资源可能的行为?

你应该使用 OPTIONS HTTP 方法来获取当前特定资源的行为。OPTIONS 请求返回方法的语义遵循 JSON API 标准。

举例来说，如果 `"GET,POST"` 是 URL OPTIONS 请求的响应，那么就可以获取该资源信息，以及创建新资源。

如果你想知道特定资源属性作用，你不得不使用应用级别的描述来定义属性的含义与功能，并使用错误响应通知用户。这个特性依旧在讨论中，尚未加入最终标准。discussion (https://github.com/json-api/json-api/issues/7) .

有没有 JSON 规范来定义 JSON API?

当然，你可以在http://jsonapi.org/format找到 JSON 规范定义。注意这个规范并不完美。 因为JSON文档可能会通过规范检查，但并不意味着是合适的 JSON API 文档。规范只是为了常规性排错检查。

可以在http://json-schema.org找到更多关于JSON 规范格式的信息。

为什么资源集合作为数组返回，而不是ID索引集合?

JSON 数组是自然排序，而集合需要元数据进行成员排序。因此，默认情况下，数组能够实现更自然的排序或者特殊方式排序。

除此之外，JSON API 允许返回不包含 IDs 的只读资源，与IDs索引集合方式不兼容。

为什么关联资源嵌套在复合文档的 `linked` 对象中?

主要资源应该相互独立，因为他们的顺序和数量通常比较重要。通过多种方式，分离主要资源和关联资源是必要的，因为主要资源可能会有相同类型的关联资源（e.g. the "parents" of a "person"）。关联资源嵌套在 `linked` 中能够防止可能的冲突。

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/json-api/