

摘要：目前互联网上充斥着大量的关于 RESTful API（为了方便，以后 API 和 RESTful API 一个意思）如何设计的文章，然而却没有一个”万能“的设计标准：如何鉴权？API 格式如何？你的 API 是否应该加入版本信息？

背景

目前互联网上充斥着大量的关于 RESTful API（为了方便，以后 API 和 RESTful API 一个意思）如何设计的文章，然而却没有一个”万能“的设计标准：如何鉴权？API 格式如何？你的 API 是否应该加入版本信息？当你开始写一个 app 的时候，特别是后端模型部分已经写完的时候，你不得不殚精竭虑的设计和实现自己 app 的 public API 部分。因为一旦发布，对外发布的 API 将会很难改变。

在给 SupportedFu 设计 API 的时候，我试图以实用的角度来解决上面提到的问题。我希望可以设计出容易使用，容易部署，并且足够灵活的 API，本文因此而生。

API 设计的基本要求

网上的很多关于 API 设计的观点都十分”学院派“，它们也许更有理论基础，但是有时却和现实世界脱轨（因此我是自由派）。所以我这篇文章的目标是从实践的角度出发，给出当前网络应用的 API 设计最佳实践（当然，是我认为的最佳了~），如果觉得不合适，我不会遵从标准。当然作为设计的基础，几个必须的原则还是要遵守的：

1. 当标准合理的时候遵守标准。
2. API 应该对程序员友好，并且在[浏览器](#)地址栏容易输入。
3. API 应该简单，直观，容易使用的同时优雅。
4. API 应该具有足够的灵活性来支持上层 ui。
5. API 设计权衡上述几个原则。

需要强调的是：API 的就是程序员的 UI，和其他 UI 一样，你必须仔细考虑它的用户体验！

使用 RESTful URLs 和 action.

虽然前面我说没有一个万能的 API 设计标准。但确实有一个被普遍承认和遵守：RESTfu 设计原则。它被 Roy Felding 提出（在他的”基于网络的软件架构“论文中[第五章](#)）。而 REST 的核心原则是将你的 API 拆分为逻辑上的资源。这些资源通过 http 被操作（GET,POST,PUT,DELETE）。

那么我应该如何拆分出这些资源呢？

显然从 API 用户的角度来看，”资源“应该是个名词。即使你的内部数据模型和资源已经有了很好的对应，API 设计的时候你仍然不需要把它们一对一的都暴露出来。这里的关键是隐藏内部资源，暴露必需的外部资源。

在 SupportFu 里，资源是 ticket、user、group。

一旦定义好了要暴露的资源，你可以定义资源上允许的操作，以及这些操作和你的 API 的对应关系：

- GET /tickets # 获取 ticket 列表
- GET /tickets/12 # 查看某个具体的 ticket
- POST /tickets # 新建一个 ticket
- PUT /tickets/12 # 更新 ticket 12.
- DELETE /tickets/12 #删除 ticket 12

可以看出使用 REST 的好处在于可以充分利用 http 的强大实现对资源的 CURD 功能。而这里你只需要一个 endpoint: /tickets,再没有其他什么命名规则和 url 规则了，cool！

这个 endpoint 的单数复数

一个可以遵从的规则是：虽然看起来使用复数来描述某一个资源实例看起来别扭，但是统一所有的 endpoint，使用复数使得你的 URL 更加规整。这让 API 使用者更加容易理解，对开发者来说也更容易实现。

如何处理关联？关于如何处理资源之间的管理 REST 原则也有相关的描述：

- GET /tickets/12/messages- Retrieves list of messages for ticket #12
- GET /tickets/12/messages/5- Retrieves message #5 for ticket #12
- POST /tickets/12/messages- Creates a new message in ticket #12
- PUT /tickets/12/messages/5- Updates message #5 for ticket #12
- PATCH /tickets/12/messages/5- Partially updates message #5 for ticket #12
- DELETE /tickets/12/messages/5- Deletes message #5 for ticket #12

其中，如果这种关联和资源独立，那么我们可以在资源的输出表示中保存相应资源的 endpoint。然后 API 的使用者就可以通过点击链接找到相关的资源。如果关联和资源联系紧密。资源的输出表示就应该直接保存相应资源信息。（例如这里如果 message 资源是独立存在的，那么上面 GET /tickets/12/messages 就会返回相应 message 的链接；相反如果 message 不独立存在，他和 ticket 依附存在，则上面的 API 调用返回直接返回 message 信息）

不符合 CURD 的操作

对这个令人困惑的问题，下面是一些解决方法：

1. **重构**你的行为 **action**。当你的行为不需要参数的时候，你可以把 **active** 对应到 **activated** 这个资源，（更新使用 **patch**）。
2. 以子资源对待。例如：[GitHub](#) 上，对一个 **gists** 加星操作：**PUT /gists/:id/star** 并且取消星操作：**DELETE /gists/:id/star**。
3. 有时候 **action** 实在没有难以和某个资源对应上例如 **search**。那就这么办吧。我认为 API 的使用者对于 **/search** 这种 **url** 也不会有太大意见的（毕竟他很容易理解）。只要注意在文档中写清楚就可以了。

永远使用 SSL

毫无例外，永远都要使用 **SSL**。你的应用不知道要被谁，以及什么情况访问。有些是安全的，有些不是。使用 **SSL** 可以减少鉴权的成本：你只需要一个简单的令牌（**token**）就可以鉴权了，而不是每次让用户对每次请求签名。

值得注意的是：不要让非 **SSL** 的 **url** 访问重定向到 **SSL** 的 **url**。

文档

文档和 **API** 本身一样重要。文档应该容易找到，并且公开（把它们藏到 **pdf** 里面或者存到需要登录的地方都不太好）。文档应该有展示请求和输出的例子：或者以点击链接的方式或者通过 **curl** 的方式（请见 **openstack** 的文档）。如果有更新（特别是公开的 **API**），应该及时更新文档。文档中应该有何时弃用某个 **API** 的时间表以及详情。使用邮件列表或者博客记录是好方法。

版本化

在 **API** 上加入版本信息可以有效的防止用户访问已经更新了的 **API**，同时也能让不同主要版本之间平稳过渡。关于是否将版本信息放入 **url** 还是放入请求头有过争论：[API version should be included in the URL or in a header](#)。学术界说它应该放到 **header** 里面去，但是如果放到 **url** 里面我们就可以跨版本的访问资源了。。（参考 **openstack**）。

strip 使用的方法就很好：它的 **url** 里面有主版本信息，同时请求头俩面有子版本信息。这样在子版本变化过程中 **url** 的稳定的。变化有时是不可避免的，关键是如何管理变化。完整的文档和合理的时间表都会使得 **API** 使用者使用的更加轻松。

结果过滤，排序，搜索：

url 最好越简短越好，和结果过滤，排序，搜索相关的功能都应该通过参数实现(并且也很容易实现)。

过滤：为所有提供过滤功能的接口提供统一的参数。例如：你想限制 `get /tickets` 的返回结果:只返回那些 `open` 状态的 `ticket-get /ticketsstate=open` 这里的 `state` 就是过滤参数。

排序：和过滤一样，一个好的排序参数应该能够描述排序规则，而不业务相关。复杂的排序规则应该通过组合实现：

- `GET /ticketsort=-priority-` Retrieves a list of tickets in descending order of priority
- `GET /ticketsort=-priority,created_at-` Retrieves a list of tickets in descending order of priority. Within a specific priority, older tickets are ordered first

这里第二条查询中，排序规则有多个 `rule` 以逗号间隔组合而成。

搜索：有些时候简单的排序是不够的。我们可以使用搜索技术（`ElasticSearch` 和 `Lucene`）来实现（依旧可以作为 url 的参数）。

- `GET /ticketsq=return&state=open&sort=-priority,created_at-` Retrieve the highest priority open tickets mentioning the word 'return'

对于经常使用的搜索查询，我们可以为他们设立别名,这样会让 API 更加优雅。例如：

`get /ticketsq=recently_closed -> get /tickets/recently_closed.`

限制 API 返回值的域

有时候 API 使用者不需要所有的结果，在进行横向限制的时候（例如值返回 API 结果的前十项）还应该可以进行纵向限制。并且这个功能能有效的提高网络带宽使用率和速度。可以使用 `fields` 查询参数来限制返回的域例如：

`GET /ticketsfields=id,subject,customer_name,updated_at&state=open&sort=-updated_at`

更新和创建操作应该返回资源

`PUT`、`POST`、`PATCH` 操作在对资源进行操作的时候常常有一些副作用：例如 `created_at,updated_at` 时间戳。为了防止用户多次的 API 调用(为了进行此次的更新操作)，我们应该会返回更新的资源（`updated representation.`）例如：在 `POST` 操作以后，返回 `201 created` 状态码，并且包含一个指向新资源的 url 作为返回头

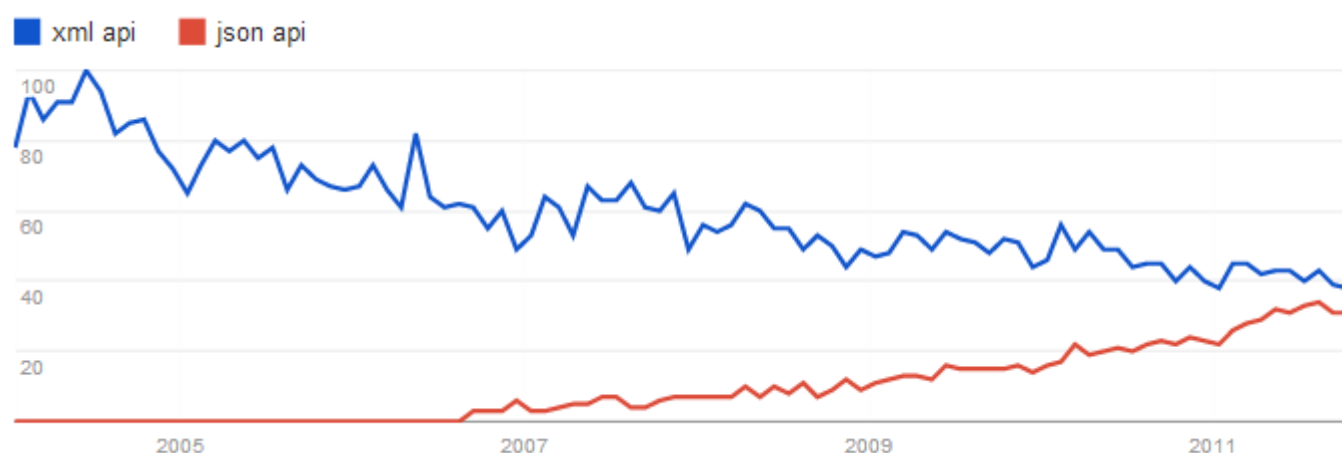
是否需要“HATEOAS”

网上关于是否允许用户创建新的 url 有很大的异议（注意不是创建资源产生的 url）。为此 REST 制定了 HATEOAS 来描述了和 endpoint 进行交互的时候，行为应该在资源的 metadata 返回值里面进行定义。

（译注：作者这里认为 HATEOAS 还不算成熟，我也不怎么理解这段就算了，读者感兴趣可以自己去原文查看）

只提供 json 作为返回格式

现在开始比较一下 XML 和 json 了。XML 即冗长，难以阅读，又不适合各种编程语言解析。当然 XML 有扩展性的优势，但是如果你只是将它来对内部资源串行化，那么他的扩展优势也发挥不出来。很多应用（youtube,twitter,box）都已经开始抛弃 XML 了，我也不想多费口舌。给了 google 上的趋势图吧：



当然如果的你使用用户里面企业用户居多，那么可能需要支持 XML。如果是这样的话你还有另外一个问题：你的 http 请求中的 media 类型是应该和 accept 头同步还是和 url？为了方便（browser explorability），应该是在 url 中（用户只要自己拼 url 就好了）。如果这样的话最好的方法是使用.xml 或者.json 的后缀。

命名方式？

是蛇形命令（下划线和小写）还是驼峰命名？如果使用 json 那么最好的应该是遵守 JAVASCRIPT 的命名方法-也就是说骆驼命名法。如果你正在使用多种语言写一个库，那么最好按照那些语言所推荐的，java，c#使用骆驼，python，ruby 使用 snake。

个人意见：我总觉得蛇形命令更好使一些，当然这没有什么理论的依据。有人说蛇形命名读起来更快，能达到 20%，也不知道真假

<http://ieeexplore.ieee.org/xpl/articleDetails.jsptp=&arnumber=5521745>

默认使用 **pretty print** 格式，使用 **gzip**

只是使用空格的返回结果从浏览器上看总是觉得很恶心(一大坨有没有？～)。当然你可以提供 url 上的参数来控制使用“**pretty print**”，但是默认开启这个选项还是更加友好。格外的传输上的损失不会太大。相反你如果忘了使用 **gzip** 那么传输效率将会大大减少，损失大大增加。想象一个用户正在 **debug** 那么默认的输出就是可读的-而不用将结果拷贝到其他什么软件中在格式化-是想起来就很爽的事，不是吗？

下面是一个例子：

```
$ curl https://API.github.com/users/veesahni > with-whitespace.txt

$ ruby -r json -e 'puts JSON.parse(STDIN.read)' <
with-whitespace.txt > without-whitespace.txt

$ gzip -c with-whitespace.txt > with-whitespace.txt.gz

$ gzip -c without-whitespace.txt > without-whitespace.txt.gz
```

输出如下：

- without-whitespace.txt- 1252 bytes
- with-whitespace.txt- 1369 bytes
- without-whitespace.txt.gz- 496 bytes
- with-whitespace.txt.gz- 509 bytes

在上面的例子中，多余的空格使得结果大小多出了 8.5%（没有使用 **gzip**），相反只多出了 2.6%。据说：twitter 使用 **gzip** 之后它的 streaming API 传输减少了 80%

（link:<https://dev.twitter.com/blog/announcing-gzip-compression-streaming-APIs>）。

只在需要的时候使用“**envelope**”

很多 API 象下面这样返回结果：

```
1  {
2      "data" : {
3          "id" : 123,
```

```
4         "name" : "John"
5     }
6 }
```

理由很简单：这样做可以很容易扩展返回结果，你可以加入一些分页信息，一些数据的元信息等——这对于那些不容易访问到返回头的 API 使用者来说确实有用，但是随着“标准”的发展（cors 和 <http://tools.ietf.org/html/rfc5988#page-6> 都开始被加入到标准中了），我个人推荐不要那么做。

何时使用 **envelope**?

有两种情况是应该使用 **envelope** 的。如果 API 使用者确实无法访问返回头，或者 API 需要支持交叉域请求（通过 **jsonp**）。

jsonp 请求在请求的 url 中包含了一个 **callback** 函数参数。如果给出了这个参数，那么 API 应该返回 **200**，并且把真正的状态码放到返回值里面（包装在信封里），例如：

```
1  callback_function({
2      status_code: 200,
3      next_page: "https://...",
4      response: {
5          ... actual JSON response body ...
6      }
7  })
```

同样为了支持无法方法返回头的 API 使用者，可以允许 **envelope=true** 这样的参数。

在 **post,put,patch** 上使用 **json** 作为输入

如果你认同我上面说的，那么你应该决定使用 **json** 作为所有的 API 输出格式，那么我们接下来考虑考虑 API 的输入数据格式。

很多的 API 使用 **url** 编码格式：就像是 **url** 查询参数的格式一样：单纯的键值对。这种方法简单有效，但是也有自己的问题：它没有数据类型的概念。这使得程序不得不根据字符串解析出布尔和整数，而且还没有层次结构——虽然有一些关于层次结构信息的约定存在可是和本身就支持层次结构的 **json** 比较一下还是不很好用。

当然如果 API 本身就很简单，那么使用 **url** 格式的输入没什么问题。但对于复杂的 API 你应该使用 **json**。或者干脆统一使用 **json**。

注意使用 **json** 传输的时候，要求请求头里面加入：**Content-Type: applicatin/json**。否则抛出 **415** 异常（**unsupported media type**）。

分页

分页数据可以放到“信封”里面，但随着标准的改进，现在我推荐将分页信息放到 link header 里面：<http://tools.ietf.org/html/rfc5988#page-6>。

使用 link header 的 API 应该返回一系列组合好了的 url 而不是让用户自己再去拼。这点在基于游标的分页中尤为重要。例如下面，来自 github 的文档

```
Link: <https://api.github.com/user/repos?page=3&per\_page=100>;  
rel="next",  
  
<https://api.github.com/user/repos?page=50&per\_page=100>; rel="last"
```

自动加载相关的资源

很多时候，自动加载相关资源非常有用，可以很大的提高效率。但是这却和 RESTful 的原则相背。为了如此，我们可以在 url 中添加参数：embed（或者 expend）。embed 可以是一个逗号分隔的串，例如：

```
GET /ticket/12embed=customer.name,assigned_user
```

对应的 API 返回值如下：

```
1  {  
2      "id" : 12,  
3      "subject" : "I have a question!",  
4      "summary" : "Hi, ....",  
5      "customer" : {  
6          "name" : "Bob"  
7      },  
8      assigned_user: {  
9          "id" : 42,  
10         "name" : "Jim",  
11     }  
12 }
```

值得提醒的是，这个功能有时候会很复杂，并且可能导致 N+1 SELECT 问题。

重写 HTTP 方法

有的客户端只能发出简单的 GET 和 POST 请求。为了照顾他们，我们可以重写 HTTP 请求。这里没有什么标准，但是一个普遍的方式是接受 X-HTTP-Method-Override 请求头。

速度限制

为了避免请求泛滥，给 API 设置速度限制很重要。为此 [RFC 6585](#) 引入了 HTTP 状态码 [429 \(too many requests\)](#)。加入速度设置之后，应该提示用户，至于如何提示标准上没有说明，不过流行的方法是使用 HTTP 的返回头。

下面是几个必须的返回头（依照 [twitter](#) 的命名规则）：

- **X-Rate-Limit-Limit**: 当前时间段允许的并发请求数
- **X-Rate-Limit-Remaining**: 当前时间段保留的请求数。
- **X-Rate-Limit-Reset**: 当前时间段剩余秒数

为什么使用当前时间段剩余秒数而不是时间戳？

时间戳保存的信息很多，但是也包含了很多不必要的信息，用户只需要知道还剩几秒就可以再发请求了这样也避免了 [clock skew 问题](#)。

有些 API 使用 UNIX 格式时间戳，我建议不要那么干。为什么？HTTP 已经规定了使用 [RFC 1123](#) 时间格式

鉴权 Authentication

restful API 是无状态的也就是说用户请求的鉴权和 cookie 以及 session 无关，每一次请求都应该包含鉴权证明。

通过使用 [ssl](#) 我们可以不用每次都提供用户名和密码：我们可以给用户返回一个随机产生的 token。这样可以极大的方便使用浏览器访问 API 的用户。这种方法适用于用户可以首先通过一次用户名-密码的验证并得到 token，并且可以拷贝返回的 token 到以后的请求中。如果不方便，可以使用 [OAuth 2](#) 来进行 token 的安全传输。

支持 jsonp 的 API 需要额外的鉴权方法，因为 jsonp 请求无法发送普通的 credential。这种情况下可以在查询 url 中添加参数：access_token。注意使用 url 参数的问题是：目前大部分的网络服务器都会讲 query 参数保存到服务器日志中，这可能会成为大的安全风险。

注意上面说到的只是三种传输 token 的方法，实际传输的 token 可能是一样的。

缓存

HTTP 提供了自带的缓存框架。你需要做的是在返回的时候加入一些返回头信息，在接受输入的时候加入输入验证。基本两种方法：

ETag: 当生成请求的时候，在 HTTP 头里面加入 ETag，其中包含请求的校验和和哈希值，这个值和输入变化的时候也应该变化。如果输入的 HTTP 请求包含 IF-NONE-MATCH 头以及一个 ETag 值，那么 API 应该返回 304 not modified 状态码，而不是常规的输出结果。

Last-Modified: 和 etag 一样，只是多了一个时间戳。返回头里的 Last-Modified: 包含了 RFC 1123 时间戳，它和 IF-MODIFIED-SINCE 一致。HTTP 规范里面有三种 date 格式，服务器应该都能处理。

出错处理

就像 html 错误页面能够显示错误信息一样，API 也应该能返回可读的错误信息—它应该和一般的资源格式一致。API 应该始终返回相应的状态码，以反映服务器或者请求的状态。API 的错误码可以分为两部分，400 系列和 500 系列，400 系列表明客户端错误：如错误的请求格式等。500 系列表示服务器错误。API 应该至少将所有的 400 系列的错误以 json 形式返回。如果可能 500 系列的错误也应该如此。json 格式的错误应该包含以下信息：一个有用的错误信息，一个唯一的错误码，以及任何可能的详细错误描述。如下：

```
{
1   "code" : 1234,
2   "message" : "Something bad happened  ",
5   "description" : "More details about the error here"
}
```

对 PUT,POST,PATCH 的输入的校验也应该返回相应的错误信息，例如：

```
1  {
2      "code" : 1024,
3      "message" : "Validation Failed",
4      "errors" : [
5          {
6              "code" : 5432,
7              "field" : "first_name",
8              "message" : "First name cannot have fancy characters"
9          },
10         {
11             "code" : 5622,
12             "field" : "password",
13             "message" : "Password cannot be blank"
14         }
15     ]
16 }
```

```
15     ]  
16 }
```

HTTP 状态码

200 ok - 成功返回状态，对应，GET, PUT, PATCH, DELETE.

201 created - 成功创建。

304 not modified - HTTP 缓存有效。

400 bad request - 请求格式错误。

401 unauthorized - 未授权。

403 forbidden - 鉴权成功，但是该用户没有权限。

404 not found - 请求的资源不存在

405 method not allowed - 该 http 方法不被允许。

410 gone - 这个 url 对应的资源现在不可用。

415 unsupported media type - 请求类型错误。

422 unprocessable entity - 校验错误时用。

429 too many request - 请求过多。