背景

目前互联网上充斥着大量的关于RESTful API（为方便，下文中“RESTful API ” 简写为“API” ）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ticekt 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；相反的如果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。
4.

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。文档中应该有关于何时弃用某个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 /tickektsstate=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 /ticketssort=-priority- Retrieves a list of tickets in descending order of priority
- GET /ticketssort=-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

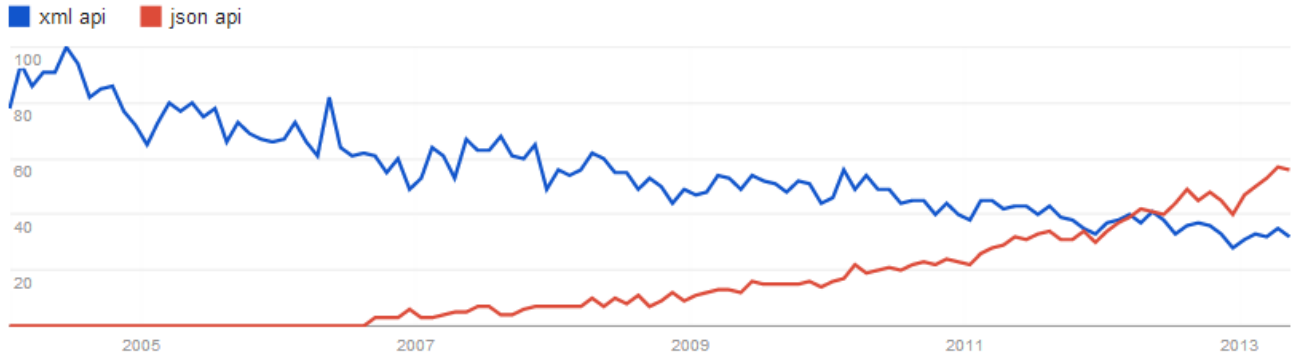
是否需要 “HATEOAS”

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，不是么？

下面是一个例子:

```
$ curl https://API.github.com/users/veesahni > with-whitespace.txt
$ ruby -r json -e 'puts JSON 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：

```
{
  "data" : {
    "id" : 123,
    "name" : "John"
  }
}
```

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：

```
callback_function({
  status_code: 200,
  next_page: "https://..",
  response: {
    ... actual JSON response body ...
  }
})
```

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：application/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>;
<https://api.github.com/user/repos?page=50&per_page=100>; rel=
```

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：

```
{
  "id" : 12,
  "subject" : "I have a question!",
  "summary" : "Hi, ....",
  "customer" : {
    "name" : "Bob"
  },
  assigned_user: {
   "id" : 42,
   "name" : "Jim",
```

```
    }
}
```

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，这个值和在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：

```
{
  "code" : 1234,
  "message" : "Something bad happened :-(",
  "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：

```
{
  "code" : 1024,
  "message" : "Validation Failed",
  "errors" : [
    {
      "code" : 5432,
      "field" : "first_name",
      "message" : "First name cannot have fancy characters"
    },
    {
       "code" : 5622,
       "field" : "password",
       "message" : "Password cannot be blank"
    }
  ]
}
```

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。
```