RESTful API 设计最佳实践

数据模型已经稳定，接下来你可能需要为web（网站）应用创建一个公开的API（应用程序编程接口）。需要认识到这样一个问题：一旦API发布后，就很难对它做很大的改动并且保持像先前一样的正确性。现在，网络上有很多关于API设计的思路。但是在全部案例中没有一种被广泛采纳的标准，有很多的选择：你接受什么样的格式？如何认证？API应该被版本化吗？

在为SupportFu（一个轻量级的Zendesk替换实现）设计API时，对于这些问题我尽量得出一些务实的答案。我的目标是设计这样一个API，它容易使用和采纳，足够灵活去为我们用户接口去埋单。

API的关键要求

许多网上能找到的API设计观点都是些学术讨论，这些讨论是关于模糊标准的主观解释，而不是关于在现实世界中具有意义的事。本文中我的目标是，描述一下为当今的web应用而设计的实用的API的最佳实践。如果感觉不对，我不会去尝试满足某个标准。为了帮助进行决策，我已经写下了API必须力争满足的一些要求：

- 它应当在需要的地方使用 web 标准
- 它应当对开发者友好并且便于在浏览器地址栏中浏览和探索
- 它应当是简单、直观和一致的，使它用起来方便和舒适
- 它应当提供足够的灵活性来增强大多数的 SupportFu 用户界面
- 它应当是高效的，同时要维持和其他需求之间的平衡

一个 API 是一个开发者的 UI - 就像其他任何 UI 一样, 确保用户体验被认真的考虑过是很重要的!

使用 RESTful URLs and actions

如果有一样东西获得广泛认可的话，那就是 RESTful 原则。Roy Felding 在他论文 network based software architectures 的 第五章 中首次介绍了这些原则。

这些REST的关键原则与将你的 API 分割成逻辑资源紧密相关。使用HTTP请求控制这些资源，其中，这些方法（GET, POST, PUT, PATCH, DELETE）具有特殊含义。

可是我该整出什么样的资源呢? 好吧，它们应该是有意义于 API 使用者的名词（不是动词）。虽然内部Model可以简单地映射到资源上，但那不一定是个一对一的映射。这里的关键是不要泄漏与API不相关的实现细节。一些相关的名词可以是 *票，用户和小组*。

一旦定义好了资源, 需要确定什么样的 actions 应用它们，这些 actions 怎么映射到你的 API 上。RESTful 原则提供了 HTTP methods 映射作为策略来处理 CRUD actions，如下：

- GET /tickets - 获取 tickets 列表
- GET /tickets/12 - 获取一个单独的 ticket
- POST /tickets - 创建一个新的 ticket
- PUT /tickets/12 - 更新 ticket #12
- PATCH /tickets/12 - 部分更新 ticket #12
- DELETE /tickets/12 - 删除 ticket #12

REST 非常棒的是，利用现有的 HTTP 方法在单个的 /tickets 接入点上实现了显著的功能。没有什么方法命名约定需要去遵循，URL 结构是整洁干净的。 *REST 太棒了!*

接入点的名称应该选择单数还是复数呢? keep-it-simple原则可以在此应用。虽然你内在的语法知识会告诉你用复数形式描述单一资源实例是错误的，但实用主义的答案是保持URL格式一致并且始终使用复数形式。不用处理各种奇形怪状的复数形式（比如person/people，goose/geese）可以让API消费者的生活更加美好，也让API提供者更容易实现API（因为大多数现代框架天然地将/tickets和/tickets/12放在同一个控制器下处理）。

但是你该如何处理（资源的）关系呢? 如果关系依托于另外一个资源，Restful原则提供了很好的指导原则。让我们来看一个例子。SupportFu的一个ticket包含许多消息（message）。这些消息逻辑上与/tickets接入点的映射关系如下：

- GET /tickets/12/messages - 获取ticket #12下的消息列表
- GET /tickets/12/messages/5 - 获取ticket #12下的编号为5的消息
- POST /tickets/12/messages - 为ticket #12创建一个新消息
- PUT /tickets/12/messages/5 - 更新ticket #12下的编号为5的消息
- PATCH /tickets/12/messages/5 - 部分更新ticket #12下的编号为5的消息
- DELETE /tickets/12/messages/5 - 删除ticket #12下的编号为5的消息

或者如果某种关系不依赖于资源，那么在资源的输出表示中只包含一个标识符是有意义的。API消费者然后除了请求资源所在的接入点外，还得再请求一次关系所在的接入点。但是如果一般情况关系和资源一起被请求，API可以提供自动嵌套关系表示到资源表示中，这样可以防止两次请求API。

如果Action不符合CRUD操作那该怎么办?

这是一个可能让人感到模糊不解的地方。有几种处理方法：

1. 重新构造这个Action，使得它像一个资源的field（我理解为部分域或者部分字段）。这种方法在Action不包含参数的情况下可以奏效。例如一个有效的action

可以映射成布尔类型field，并且可以通过PATCH更新资源。
2. 利用RESTful原则像处理子资源一样处理它。例如，Github的API让你通过PUT /gists/:id/star 来 star a gist ，而通过 DELETE /gists/:id/star来进行 unstar 。
3. 有时候你实在是没有办法将Action映射到任何有意义的RESTful结构。例如，多资源搜索没办法真正地映射到任何一个资源接入点。这种情况，/search 将非常有意义，虽然它不是一个名词。这样做没有问题 - 你只需要从API消费者的角度做正确的事，并确保所做的一切都用文档清晰记录下来了以避免（API消费者的）困惑。

总是使用 **SSH**

总是使用SSL，没有例外。今天，您的web api可以从任何地方访问互联网(如图书馆、咖啡店、机场等)。不是所有这些都是安全的，许多不加密通信,便于窃听或伪造，如果身份验证凭证被劫持。

另一个优点是,保证总是使用SSL加密通信简化了认证效果——你可以摆脱简单的访问令牌,而不是让每个API请求签署。

要注意的一点是非SSL访问API URLs。不要重定向这些到对应的SSL。相反，抛出一个系统错误!最后一件你想要的是配置不佳的客户发送请求到一个未加密的端点，只是默默地重定向到实际加密的端点。

文档

API的好坏关键看其文档的好坏. 好的API的说明文档应该很容易就被找到，并能公开访问。在尝试任何整合工作前大部分开发者会先查看其文档。当文档被藏于一个PDF之中或要求必须登记信息时，将很难被找到也很难搜索到。

好的文档须提供从请求到响应整个循环的示例。最好的是，请求应该是可粘贴的例子，要么是可以贴到浏览器的链接，要么是可以贴到终端里的curl示例 。 GitHub 和 Stripe 在这方面做的非常出色。

一旦你发布一个公开的API，你必须承诺"在没有通告的前提下，不会更改APIDe功能" .对于外部可见API的更新，文档必须包含任何将废弃的API的时间表和详情。应该通过博客(更新日志)或者邮件列表送达更新说明(最好两者都通知)。

版本控制

必须对API进行版本控制。版本控制可以快速迭代并避免无效的请求访问已更新的接入点。它也有助于帮助平滑过渡任何大范围的API版本变迁，这样就可以继续支持旧版本API。

关于API的版本是否应该包含在URL或者请求头中 莫衷一是。从学术派的角度来讲，它应该出现在请求头中。然而版本信息出现在URL中必须保证不同版本资源的浏览器可浏览性（browser explorability），还记得文章开始提到的API要求吗?

我非常赞成 approach that Stripe has taken to API versioning - URL包含一个主版本号（比如 http://shonzilla/api/v1/customers/1234）），但是API还包含基于日期的子版本（比如http://shonzilla/api/v1.2/customers/1234），可以通过配置HTTP请求头来进行选择。这种情况下，主版本确保API结构总体稳定性，而子版本会考虑细微的变化（field deprecation、接入点变化等）。

API不可能完全稳定。变更不可避免，重要的是变更是如何被控制的。维护良好的文档、公布未来数月的deprecation计划，这些对于很多API来说都是一些可行的举措。它归根结底是看对于业界和API的潜在消费者是否合理。

结果过滤，排序和搜索

最好是尽量保持基本资源URL的简洁性。 复杂结果过滤器、排序需求和高级搜索 (当限定在单一类型的资源时) ，都能够作为在基本URL之上的查询参数来轻松实现。下面让我们更详细的看一下:

过滤: 对每一个字段使用一个唯一查询参数，就可以实现过滤。 例如，当通过"/tickets"终端来请求一个票据列表时，你可能想要限定只要那些在售的票。这可以通过一个像 GET /tickets?state=open 这样的请求来实现。这里"state"是一个实现了过滤功能的查询参数。

排序: 跟过滤类似, 一个泛型参数排序可以被用来描述排序的规则. 为适应复杂排序需求，让排序参数采取逗号分隔的字段列表的形式，每一个字段前都可能有一个负号来表示按降序排序。我们看几个例子:

- GET /tickets?sort=-priority - 获取票据列表，按优先级字段降序排序
- GET /tickets?sort=-priority,created_at - 获取票据列表，按"priority"字段降序排序。在一个特定的优先级内，较早的票排在前面。

搜索: 有时基本的过滤不能满足需求，这时你就需要全文检索的力量。或许你已经在使用 ElasticSearch 或者其它基于 Lucene 的搜索技术。当全文检索被用作获取某种特定资源的资源实例的机制时， 它可以被暴露在API中，作为资源终端的查询参数，我们叫它"q"。搜索类查询应当被直接交给搜索引擎，并且API的产出物应当具有同样的格式，以一个普通列表作为结果。

把这些组合在一起，我们可以创建以下一些查询:

- GET /tickets?sort=-updated_at - 获取最近更新的票
- GET /tickets?state=closed&sort=-updated_at - 获取最近更新并且状态为关闭的票。
- GET /tickets?q=return&state=open&sort=-priority,created_at - 获取优先级最高、最先创建的、状态为开放的票，并且票上有 'return' 字样。

一般查询的别名

为了使普通用户的API使用体验更加愉快， 考虑把条件集合包装进容易访问的RESTful 路径中。比如上面的，最近关闭的票的查询可以被包装成 GET /tickets/recently_closed

限制哪些字段由**API**返回

API的使用者并不总是需要一个资源的完整表示。选择返回字段的功能由来已久，它使得API使用者能够最小化网络阻塞，并加速他们对API

的调用。

使用一个字段查询参数，它包含一个用逗号隔开的字段列表。例如，下列请求获得的信息将刚刚足够展示一个在售票的有序列表：

GET /tickets?fields=id,subject,customer_name,updated_at&state=open&sort=-updated_at

更新和创建应该返回一个资源描述

一个 PUT, POST 或者 PATCH 调用可能会对指定资源的某些字段造成更改，而这些字段本不在提供的参数之列 (例如: created_at 或 updated_at 这两个时间戳)。 为了防止API使用者为了获取更新后的资源而再次调用该API，应当使API把更新(或创建)后的资源作为response的一部分来返回。

以一个产生创建活动的 POST 操作为例, 使用一个 HTTP 201 状态代码 然后包含一个 Location header 来指向新生资源的URL。

你是否应该**HATEOAS?**

（译注：Hypermedia as the Engine of Application State (*HATEOAS*)超媒体作为应用程序状态引擎）

对于API消费方是否应该创建链接，或者是否应该将链接提供给API，有许多混杂的观点。RESTful的设计原则指定了HATEOAS ，大致说明了与某个端点的交互应该定义在元数据(metadata)之中，这个元数据与输出结果一同到达，并不基于其他地方的信息。

虽然web逐渐依照HATEOAS类型的原则运作（我们打开一个网站首页并随着我们看到的页面中的链接浏览），我不认为我们已经准备好API的HATEOAS了。当浏览一个网站的时候，决定点击哪个链接是运行时做出的。然而，对于API，决定哪个请求被发送是在写API集成代码时做出的，并不是运行时。这个决定可以移交到运行时吗？当然可以，不过顺着这条路没有太多好处，因为代码仍然不能不中断的处理重大的API变化。也就是说，我认为HATEOAS做出了承诺，但是还没有准备好迎接它的黄金时间。为了完全实现它的潜能，需要付出更多的努力去定义围绕着这些原则的标准和工具。

目前而言，最好假定用户已经访问过输出结果中的文档&包含资源标识符，而这些API消费方会在制作链接的时候用到。关注标识符有几个优势——网络中的数据流减少了，API消费方存储的数据也减少了（因为它们存储的是小的标识符而不是包含标识符的URLs）。

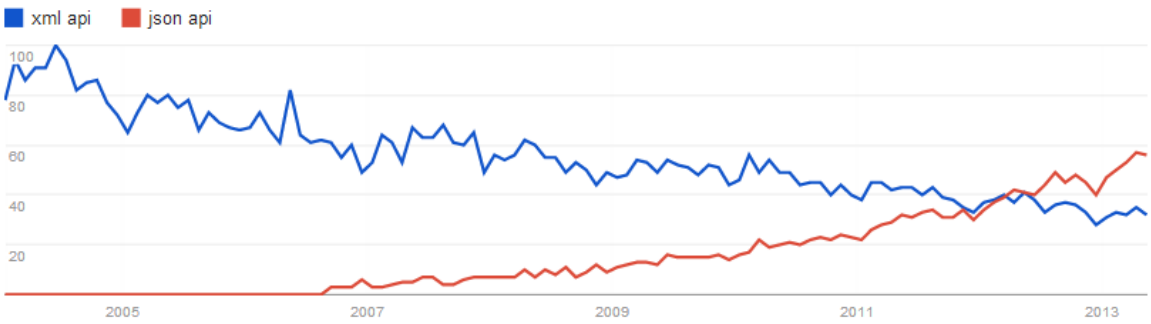
同样的，在URL中提供本文倡导的版本号，对于在一个很长时间内API消费方存储资源标识符（而不是URLs），它更有意义。总之，标识符相对版本是稳定的，但是表示这一点的URL却不是的！

只返回**JSON**

是时候在API中丢弃XML了。XML冗长，难以解析，很难读，他的数据模型和大部分编程语言的数据模型 不兼容，而他的可扩展性优势在你的主要需求是必须序列化一个内部数据进行输出展示时变得不相干。

我不打算对上述进行解释了，貌似诸如 (YouTube, Twitter 和 Box)之类的已经开始了去XML化.

给你一张google趋势图，比较XML API 和 JSON API的，供你参考：



但是，如果你的客户群包括大量的企业客户,你会发现自己不得不支持XML的方式。如果你必须这样,一个新问题出现了：

媒体类型是应该基于Accept头还是基于URL呢 ? 为确保浏览器的浏览性，应该基于URL。这里最明智的选择是在端点URL后面附加 .json 或 .xml 的扩展.

字段名称书写格式的 **snake_case vs camelCase**

如果你在使用JSON (*JavaScript* Object Notation) 作为你的主要表示格式，正确的方法就是遵守JavaScript命名约定——对字段名称使用camelCase！如果你要走用各种语言建设客户端库的路线，最好使用它们惯用的命名约定—— C# & Java 使用camelCase, python & ruby 使用snake_case。

深思：我一直认为snake_case比JavaScript的camelCase约定更容易阅读。我没有任何证据来支持我的直觉，直到现在，基于从2010年的camelCase 和 snake_case的眼动追踪研究 (PDF)，**snake_case比驼峰更容易阅读20%**！这种阅读上的影响会影响API的可勘探性和文档中的示例。

许多流行的JSON API使用snake_case。我怀疑这是由于序列化库遵从它们所使用的底层语言的命名约定。也许我们需要有JSON序列库来处理命名约定转换。

缺省情况下确保漂亮的打印和支持**gzip**

一个提供空白符压缩输出的API，从浏览器中查看结果并不美观。虽然一些有序的查询参数（如 ?pretty=true ）可以提供来使漂亮打印生效，一个默认情况下能进行漂亮打印的API更为平易近人。额外数据传输的成本是微不足道的，尤其是当你比较不执行gzip压缩的成本。

考虑一些用例：假设分析一个API消费者正在调试并且有自己的代码来打印出从API收到的数据——默认情况下这应是可读的。或者，如果消费者抓住他们的代码生成的URL，并直接从浏览器访问它——默认情况下这应是可读的。这些都是小事情。做好小事情会使一个API能被更愉快地使用!

那么该如何处理额外传输的数据呢?

让我们看一个实际例子。我从GitHub API上拉取了一些数据，默认这些数据使用了漂亮打印（pretty print）。我也将做一些GZIP压缩后的对比。

```
$ 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
```

输出文件的大小如下：

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

在这个例子中，当未启用GZIP压缩时空格增加了8.5%的额外输出大小，而当启用GZIP压缩时这个比例是2.6%。另一方面，GZIP压缩节省了60%的带宽。由于漂亮打印的代价相对比较小，最好默认使用漂亮打印，并确保GZIP压缩被支持。

关于这点想了解更多的话，Twitter发现当对他们的 Streaming API 开启GZIP支持后可以在某些情况获得 80%的带宽节省 。Stack Exchange甚至强制要求必须对API请求结果使用GZIP压缩（never return a response that's not compressed）。

不要默认使用大括号封装，但要在需要的时候支持

许多API会像下面这样包裹他们的响应信息:

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

有不少这样做的理由 - 更容易附加元数据或者分页信息，一些REST客户端不允许轻易的访问HTTP头信息，并且JSONP请求不能访问HTTP头信息。无论怎样，随着迅速被采用的标准，比如CORS和Link header from RFC 5988, 大括号封装开始变得不必要。

我们应当默认不使用大括号封装，而仅在特殊情况下使用它，从而使我们的API面向未来。

特殊情况下该如何使用大括号封装?

有两种情况确实需要大括号封装 - 当API需要通过JSONP来支持跨域的请求时，或者当客户端没有能力处理HTTP头信息时。

JSONP 请求附带有一个额外的查询参数(通常称为callback或jsonp) 表示了回调函数的名称。如果提供了这个参数，API应当切换至完整封装模式，这时它总是用200HTTP状态码作为响应，然后把真实的状态码放入JSON有效载荷中。任何被一并添加进响应中的额外的HTTP头信息都应当被映射到JSON字段中, 像这样:

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

类似的，为了支持HTTP受限的客户端，可以允许一个特殊的查询参数"?envelope=true"来触发完整封装(没有JSONP回调函数)。

使用JSON 编码的 POST, PUT & PATCH 请求体

如果你正在跟随本文中讲述的开发过程，那么你肯定已经接受JSON作为API的输出。下面让我们考虑使用JSON作为API的输入。

许多API在他们的API请求体中使用URL编码。URL编码正如它们听起来那样 - 将使用和编码URL查询参数时一样的约定，对请求体中的键值对进行编码。这很简单，被广泛支持而且实用。

然而，有几个问题使得URL编码不太好用。首先，它没有数据类型的概念。这迫使API从字符串中转换整数和布尔值。而且，它并没有真正的层次结构的概念。尽管有一些约定，可以用键值对构造出一些结构（比如给一个键增加"[]"来表示一个数组），但还是不能跟JSON原生的层次结构相比。

如果API很简单，URL编码可以满足需要。然而，复杂API应当严格对待他们的JSON格式的输入。不论哪种方式，选定一个并且整套API要保持一致。

一个能接受JSON编码的POST, PUT 和 PATCH请求的API，应当也需要把Content-Type头信息设置为application/json，或者抛出一个415不支持的媒体类型（Unsupported Media Type）的HTTP状态码。

分页

信封喜欢将分页信息包含在信封自身的API。我不能指责这点——直到最近，我们才找到更好的方法。正确的方法是使用 RFC 5988 中介绍的链接标头。

使用链接标头的API可以返回一系列线程的链接，API使用者无需自行生成链接。这在分页时指针导向 非常重要。下面是抓取自 Github的正确使用链接标头的文件：

```
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"
```

不过这个并非完成版本，因为很多 API 喜欢返回额外信息，例如可用结果的总数。需要发送数量的 API 可用类似 X-Total-Count 的普通 HTTP 标头。

自动装载相关的资源描述

在很多种情况下，API的使用者需要加载和被请求资源相关的数据（或被请求资源引用的数据）。与要求使用者反复访问API来获取这些信息相比，允许在请求原始资源的同时一并返回和装载相关资源，将会带来明显的效率提升。

然而，由于这样确实 有悖于一些RESTful原则，所以我们可以只使用一个内置的（或扩展）的查询参数来实现这一功能，来最小化与原则的背离。

这种情况下，“embed”将是一个逗号隔开的需要被内置的字段列表。点号可以用来表示子字段。例如：

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

这将返回一个附带有详细内置信息的票据，如下：

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

当然，实现类似于这种功能的能力完全依赖于内在的复杂度。这种内置的做法很容易产生 N+1 select 问题。

重写/覆盖 HTTP 方法

一些HTTP客户端仅能处理简单的的GET和POST请求，为照顾这些功能有限的客户端，API需要一种方式来重写HTTP方法. 尽管没有一些硬性标准来做这事，但流行的惯例是接受一种叫 X-HTTP的请求头，重写是用一个字符串值包含PUT，PATCH或DELETE中的一个。

注意重写头应当仅接受POST请求，GET请求绝不应该 更改服务器上的数据!

速率限制

为了防止滥用，标准的做法是给API增加某种类型的速率限制。RFC 6585 中介绍了一个HTTP状态码429 请求过多来实现这一点。

不论怎样，在用户实际受到限制之前告知他们限制的存在是很有用的。这是一个现在还缺乏标准的领域，但是已经有了一些流行的使用HTTP响应头信息的惯用方法。

最少时包含下列头信息(使用Twitter的命名约定 来作为头信息，通常没有中间词的大写):

- X-Rate-Limit-Limit - 当期允许请求的次数
- X-Rate-Limit-Remaining - 当期剩余的请求次数
- X-Rate-Limit-Reset - 当期剩余的秒数

为什么对X-Rate-Limit-Reset不使用时间戳而使用秒数?

一个时间戳包含了各种各样的信息，比如日期和时区，但它们却不是必需的。一个API使用者其实只是想知道什么时候能再次发起请求，对他们来说一个秒数用最小的额外处理回答了这个问题。同时规避了时钟偏差的问题。

有些API给X-Rate-Limit-Reset使用UNIX时间戳(纪元以来的秒数)。不要这样做!

为什么对X-Rate-Limit-Reset使用UNIX时间戳是不好的做法?

HTTP 规范已经指定使用RFC 1123 的日期格式 (目前被使用在日期, If-Modified-Since & Last-Modified HTTP头信息中)。如果我们打算指定一种使用某种形式时间戳的、新的HTTP头信息，我们应当遵循RFC 1123规定，而不是使用UNIX时间戳。

认证

一个 RESTful API 应当是无状态的。这意味着认证请求应当不依赖于cookie或session。相反，每一个请求都应当携带某种类型的认证凭证。

由于总是使用SSL，认证凭证能够被简化为一个随机产生的访问令牌，里面传入一个使用HTTP Basic Auth的用户名字段。这样做的极大的好处是，它是完全的浏览器可探测的 - 如果浏览器从服务器收到一个401未授权状态码，它仅需要一个弹出框来索要凭证即可。

然而，这种基于基本认证的令牌的认证方法，仅在满足下列情形时才可用，即用户可以把令牌从一个管理接口复制到API使用者环境。当这种情形不能成立时，应当使用OAuth 2来产生安全令牌并传递给第三方。OAuth 2使用了承载令牌(Bearer tokens) 并且依赖于SSL的底层传输加密。

一个需要支持JSONP的API将需要第三种认证方法，因为JSONP请求不能发送HTTP基本认证凭据(HTTP Basic Auth)或承载令牌(Bearer

tokens) 。这种情况下，可以使用一个特殊的查询参数access_token。注意，使用查询参数token存在着一个固有的安全问题，即大多数的web服务器都会把查询参数记录到服务日志中。

这是值得的，所有上面三种方法都只是跨API边界两端的传递令牌的方式。实际的底层令牌本身可能都是相同的。

缓存

HTTP 提供了一套内置的缓存框架! 所有你必须做的是，包含一些额外的出站响应头信息，并且在收到一些入站请求头信息时做一点儿校验工作。

有两种方式：ETag和Last-Modified

ETag: 当产生一个请求时，包含一个HTTP 头，ETag会在里面置入一个和表达内容对应的哈希值或校验值。这个值应当跟随表达内容的变化而变化。现在，如果一个入站HTTP请求包含了一个If-None-Match头和一个匹配的ETag值，API应当返回一个304未修改状态码，而不是返回请求的资源。

Last-Modified: 基本上像ETag那样工作，不同的是它使用时间戳。在响应头中，Last-Modified包含了一个RFC 1123格式的时间戳，它使用If-Modified-Since来进行验证。注意，HTTP规范已经有了 3 种不同的可接受的日期格式 ，服务器应当准备好接收其中的任何一种。

错误

就像一个HTML错误页面给访问者展示了有用的错误信息一样，一个API应当以一种已知的可使用的格式来提供有用的错误信息。 错误的表示形式应当和其它任何资源没有区别，只是有一套自己的字段。

API应当总是返回有意义的HTTP状态代码。API错误通常被分成两种类型：代表客户端问题的400系列状态码和代表服务器问题的500系列状态码。最简情况下，API应当把便于使用的JSON格式作为400系列错误的标准化表示。如果可能(意思是，如果负载均衡和反向代理能创建自定义的错误实体)，这也适用于500系列错误代码。

一个JSON格式的错误信息体应当为开发者提供几样东西 - 一个有用的错误信息，一个唯一的错误代码 (能够用来在文档中查询详细的错误信息) 和可能的详细描述。这样一个JSON格式的输出可能会像下面这样：

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

对PUT, PATCH和POST请求进行错误验证将需要一个字段分解。下面可能是最好的模式：使用一个固定的顶层错误代码来验证错误，并在额外的字段中提供详细错误信息，就像这样：

```
{
  "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 状态代码

HTTP定义了一套可以从API返回的有意义的状态代码。 这些代码能够用来帮助API使用者对不同的响应做出相应处理。我已经把你必然会用到的那些列成了一个简短的清单：

- 200 OK (成功) - 对一次成功的GET, PUT, PATCH 或 DELETE的响应。也能够用于一次未产生创建活动的POST
- 201 Created (已创建) - 对一次导致创建活动的POST的响应。 同时结合使用一个位置头信息指向新资源的位置- Response to a POST that results in a creation. Should be combined with a Location header pointing to the location of the new resource
- 204 No Content (没有内容) - 对一次没有返回主体信息(像一次DELETE请求)的请求的响应
- 304 Not Modified (未修改) - 当使用HTTP缓存头信息时使用304
- 400 Bad Request (错误的请求) - 请求是畸形的，比如无法解析请求体
- 401 Unauthorized (未授权) - 当没有提供或提供了无效认证细节时。如果从浏览器使用API，也可以用来触发弹出一次认证请求
- 403 Forbidden (禁止访问) - 当认证成功但是认证用户无权访问该资源时
- 404 Not Found (未找到) - 当一个不存在的资源被请求时
- 405 Method Not Allowed (方法被禁止) - 当一个对认证用户禁止的HTTP方法被请求时
- 410 Gone (已删除) - 表示资源在终端不再可用。当访问老版本API时，作为一个通用响应很有用
- 415 Unsupported Media Type (不支持的媒体类型) - 如果请求中包含了不正确的内容类型
- 422 Unprocessable Entity (无法处理的实体) - 出现验证错误时使用
- 429 Too Many Requests (请求过多) - 当请求由于访问速率限制而被拒绝时

总结

一个API是一个给开发者使用的用户接口。要努力确保它不仅功能上可用，更要用起来愉快。

本文地址：http://www.oschina.net/translate/best-practices-for-a-pragmatic-restful-api
原文地址：http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api

---