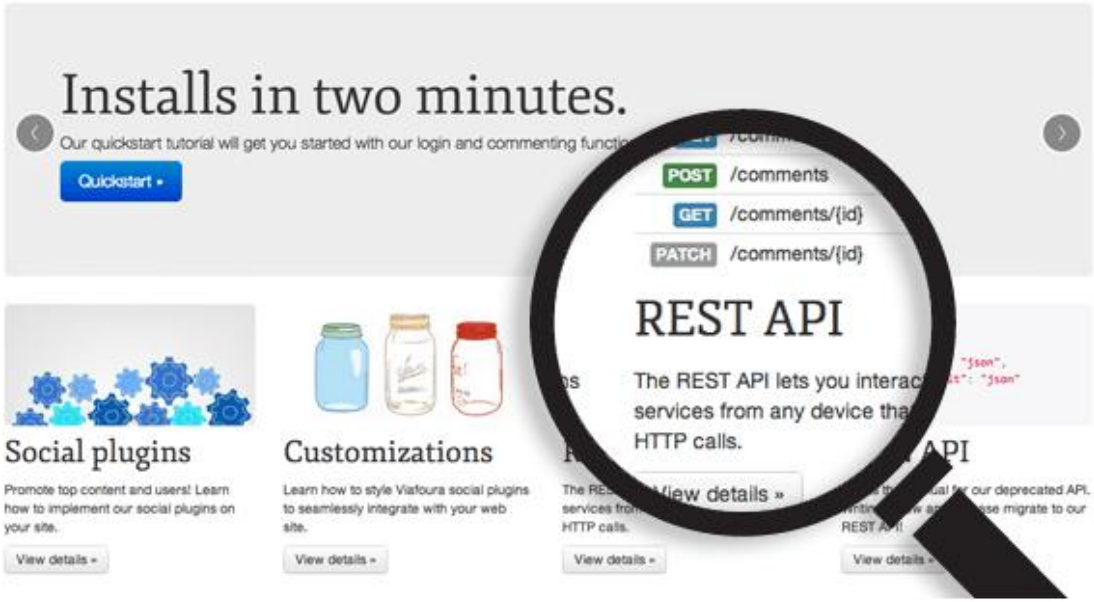
RESTful API 设计指南

网络应用程序，分为前端和后端两个部分。当前的发展趋势，就是前端设备层出不穷（手机、平板、桌面电脑、其他专用设备.....）。

因此，必须有一种统一的机制，方便不同的前端设备与后端进行通信。这导致API构架的流行，甚至出现"API First"的设计思想。RESTful API是目前比较成熟的一套互联网应用程序的API设计理论。我以前写过一篇《理解RESTful架构》，探讨如何理解这个概念。

今天，我将介绍RESTful API的设计细节，探讨如何设计一套合理、好用的API。我的主要参考了两篇文章（1，2）。



一、协议

API与用户的通信协议，总是使用HTTPs协议。

二、域名

应该尽量将API部署在专用域名之下。

> *https://api.example.com*

如果确定API很简单，不会有进一步扩展，可以考虑放在主域名下。

> *https://example.org/api/*

三、版本（Versioning）

应该将API的版本号放入URL。

> *https://api.example.com/v1/*

另一种做法是，将版本号放在HTTP头信息中，但不如放入URL方便和直观。
Github采用这种做法。

四、路径（Endpoint）

路径又称"终点"（endpoint），表示API的具体网址。

在RESTful架构中，每个网址代表一种资源（resource），所以网址中不能有动词，只能有名词，而且所用的名词往往与数据库的表格名对应。一般来说，数据库中的表都是同种记录的"集合"（collection），所以API中的名词也应该使用复数。

举例来说，有一个API提供动物园（zoo）的信息，还包括各种动物和雇员的信息，则它的路径应该设计成下面这样。

> *https://api.example.com/v1/zoos*
> *https://api.example.com/v1/animals*
> *https://api.example.com/v1/employees*

五、HTTP动词

对于资源的具体操作类型，由HTTP动词表示。

常用的HTTP动词有下面五个（括号里是对应的SQL命令）。

> *GET*（*SELECT*）：从服务器取出资源（一项或多项）。
> *POST*（*CREATE*）：在服务器新建一个资源。
> *PUT*（*UPDATE*）：在服务器更新资源（客户端提供改变后的完整资源）。
> *PATCH*（*UPDATE*）：在服务器更新资源（客户端提供改变的属性）。
> *DELETE*（*DELETE*）：从服务器删除资源。

还有两个不常用的HTTP动词。

> *HEAD*：获取资源的元数据。
> *OPTIONS*：获取信息，关于资源的哪些属性是客户端可以改变的。

下面是一些例子。

> *GET /zoos*：列出所有动物园
> *POST /zoos*：新建一个动物园
> *GET /zoos/ID*：获取某个指定动物园的信息
> *PUT /zoos/ID*：更新某个指定动物园的信息（提供该动物园的全部信息）
> *PATCH /zoos/ID*：更新某个指定动物园的信息（提供该动物园的部分信息）
> *DELETE /zoos/ID*：删除某个动物园
> *GET /zoos/ID/animals*：列出某个指定动物园的所有动物
> *DELETE /zoos/ID/animals/ID*：删除某个指定动物园的指定动物

六、过滤信息（Filtering）

如果记录数量很多，服务器不可能都将它们返回给用户。API应该提供参数，过滤返回结果。

下面是一些常见的参数。

> *?limit=10*：指定返回记录的数量
> *?offset=10*：指定返回记录的开始位置。
> *?page=2&per_page=100*：指定第几页，以及每页的记录数。
> *?sortby=name&order=asc*：指定返回结果按照哪个属性排序，以及排序顺序。
> *?animal_type_id=1*：指定筛选条件

参数的设计允许存在冗余，即允许API路径和URL参数偶尔有重复。比如，GET /zoo/ID/animals 与 GET /animals?zoo_id=ID 的含义是相同的。

七、状态码（Status Codes）

服务器向用户返回的状态码和提示信息，常见的有以下一些（方括号中是该状态码对应的HTTP动词）。

> *200 OK - [GET]*：服务器成功返回用户请求的数据，该操作是幂等的（*Idempotent*）。
> *201 CREATED - [POST/PUT/PATCH]*：用户新建或修改数据成功。
> *202 Accepted - [*]*：表示一个请求已经进入后台排队（异步任务）
> *204 NO CONTENT - [DELETE]*：用户删除数据成功。
> *400 INVALID REQUEST - [POST/PUT/PATCH]*：用户发出的请求有错误，服务器没有进行新建或修改数据的操作，该操作是幂等的。
> *401 Unauthorized - [*]*：表示用户没有权限（令牌、用户名、密码错误）。
> *403 Forbidden - [*]* 表示用户得到授权（与*401*错误相对），但是访问是被禁止的。
> *404 NOT FOUND - [*]*：用户发出的请求针对的是不存在的记录，服务器没有进行操作，该操作是幂等的。
> *406 Not Acceptable - [GET]*：用户请求的格式不可得（比如用户请求*JSON*格式，但是只有*XML*格式）。
> *410 Gone -[GET]*：用户请求的资源被永久删除，且不会再得到的。
> *422 Unprocesable entity - [POST/PUT/PATCH]* 当创建一个对象时，发生一个验证错误。
> *500 INTERNAL SERVER ERROR - [*]*：服务器发生错误，用户将无法判断发出的请求是否成功。

状态码的完全列表参见这里。

八、错误处理（Error handling）

如果状态码是4xx，就应该向用户返回出错信息。一般来说，返回的信息中将error作为键名，出错信息作为键值即可。

```
{
error: "Invalid API key"
}
```

九、返回结果

针对不同操作，服务器向用户返回的结果应该符合以下规范。

> *GET /collection*：返回资源对象的列表（数组）
> *GET /collection/resource*：返回单个资源对象
> *POST /collection*：返回新生成的资源对象
> *PUT /collection/resource*：返回完整的资源对象
> *PATCH /collection/resource*：返回完整的资源对象
> *DELETE /collection/resource*：返回一个空文档

十、Hypermedia API

RESTful API最好做到Hypermedia，即返回结果中提供链接，连向其他API方法，使得用户不查文档，也知道下一步应该做什么。

比如，当用户向api.example.com的根目录发出请求，会得到这样一个文档。

```
{"link": {
```

```
"rel": "collection https://www.example.com/zoos",
"href": "https://api.example.com/zoos",
"title": "List of zoos",
"type": "application/vnd.yourformat+json"
}}
```

上面代码表示，文档中有一个link属性，用户读取这个属性就知道下一步该调用什么API了。rel表示这个API与当前网址的关系（collection关系，并给出该collection的网址），href表示API的路径，title表示API的标题，type表示返回类型。

Hypermedia API的设计被称为HATEOAS。Github的API就是这种设计，访问api.github.com会得到一个所有可用API的网址列表。

```
{
"current_user_url": "https://api.github.com/user",
"authorizations_url": "https://api.github.com/authorizations",
// ...
}
```

从上面可以看到，如果想获取当前用户的信息，应该去访问api.github.com/user，然后就得到了下面结果。

```
{
"message": "Requires authentication",
"documentation_url": "https://developer.github.com/v3"
}
```

上面代码表示，服务器给出了提示信息，以及文档的网址。

十一、其他

（1）API的身份认证应该使用OAuth 2.0框架。

（2）服务器返回的数据格式，应该尽量使用JSON，避免使用XML。

（完）