Restful API

1. Restful API

Restful API通常满足以下原则：

a. 一类资源可以用两个URL

一个URL表示该类型资源集合，另一个URL用来表示特定的资源元素。

```
# 资源集合：
/epics
# 资源元素：
/epics/5
```

b. 使用名词而不是动词

使用Http方法来表达动作（增、删、改、查）。

- 增（POST，非幂等性）：使用POST方法创建新的资源。
- 删（DELETE，幂等性）：使用DELETE方法删除存在的资源。
- 改（PUT，幂等性）：使用PUT或PATCH方法来更新已存在的资源。
- 查（GET，幂等性）：使用GET方法读取资源。

反例：

```
/getAllEpics
/getAllFinishedEpics
/createEpic
/updateEpic
```

正解：

```
GET /epics
GET /epics?state=finished
POST /epics
PUT /epics/5
```

c. 使用一致的复数名词

避免混用复数和单数形式，只应该使用统一的复数名词来表达资源。

错误：

```
GET /story
GET /story/3
```

正解：

```
GET /stories
GET /stories/3
```

d. 将实际数据包装在data字段中

GET /epics在数据字段中返回epic资源列表：

```
{
  "data": [
    { "id": 1, "name": "epic1" }
    , { "id": 2, "name": "epic2" }
  ]
}
```

GET /epic/1在数据字段中返回id为1的epic对象：

```
{
  "data": {
    "id": 1,
    "name": "epic1"
  }
}
```

PUT，POST和PATCH请求的有效负荷还应包含实际对象的数据字段。

优点：

- 还有空间扩展元数据
- 一致性
- 兼容JSON API标准

e. 对可选及复杂参数使用查询字符串（？）

反例：

```
GET /employees
GET /externalEmployees
GET /internalEmployees
GET /internalAndSeniorEmployees
```

正解：

- 保持URL简单短小。 坚持使用基本URL，将复杂或可选参数移动到查询字符串。

```
GET /employees?state=internal&title=senior
GET /employees?id=1,2
```

- 另外还可以使用JSON API方式过滤：

```
GET /employees?filter[state]=internal&filter[title]=senior
GET /employees?filter[id]=1,2
```

f. 使用HTTP状态码

状态码	描述
1xx	代表请求已被接受，需要继续处理
2xx	请求已成功，请求所希望的响应头或数据体将随此响应返回

3xx	重定向
4xx	客户端原因引起的错误
5xx	服务端引起的错误

请注意，使用所有过多的HTTP状态码可能会让API用户感到困惑。所以应该保持使用精简的HTTP状态码集。常用状态码如下：

- **2xx**
  - 200：请求成功。一般用于GET与POST请求
  - 201：已创建。成功请求并创建了新的资源
- **3xx**
  - 301：永久移动。请求的资源已被永久的移动到新URI，返回信息会包括新的URI，浏览器会自动定向到新URI。今后任何新的请求都应使用新的URI代替
  - 304：未修改。所请求的资源未修改，服务器返回此状态码时，不会返回任何资源。客户端通常会缓存访问过的资源，通过提供一个头信息指出客户端希望只返回在指定日期之后修改的资源
- **4xx**
  - 400： 客户端请求的语法错误，服务器无法理解
  - 401：请求要求用户的身份认证
  - 403： 服务器理解请求客户端的请求，但是拒绝执行此请求
  - 404：服务器无法根据客户端的请求找到资源（网页）。通过此代码，网站设计人员可设置”您所请求的资源无法找到”的个性页面
  - 410：客户端请求的资源已经不存在。410不同于404，如果资源以前有现在被永久删除了可使用410代码，网站设计人员可通过301代码指定资源的新位置
- **5xx**
  - 500：服务器内部错误，无法完成请求

g. 提供有用的错误信息

除了提供恰当的HTTP状态代码外，还应该在HTTP响应正文中提供有用且详细的错误描述。

请求：

```
GET /epics?state=unknow
```

响应：

```
// 400 Bad Request
{
  "errors": [
    {
      "status": 400,
      "detail": "Invalid state. Valid values are 'biz' or 'tech'",
      "code": 352,
      "links": {
        "about": "http://www.jira.com/rest/errorcode/352"
      }
    }
  ]
}
```

h. 使用HATEOAS

HATEOAS 是 Hypermedia As The Engine Of Application State 的缩写，从字面上理解是“超媒体即是应用状态引擎”。其原则就是客户端与服务器的交互完全由超媒体动态提供，客户端无需事先了解如何与数据或者服务器交互。相反的，在一些RPC服务或者Redis,Mysql等软件，需要事先了解接口定义或者特定的交互语法。举例如下：

> 客户想要访问epic的用户故事清单。 因此，他必须知道他可以通过将查询参数stories附加到员工URL（例如/epics/21/stories）来访问用户故事清单。 这种字符串拼接易错，脆弱且难以维护。 如果更改了在REST API中访问salary语句的方式（例如，现在使用“storyStatements”或“userStories”），则所有客户端都将中断。

更好的做法是在响应中提供客户可以跟进的链接。 例如，对GET /epic的响应可能如下所示：

```
{
  "data": [
    {
      "id":1,
      "name":"epic1",
      "links": [
        {
          "story": "http://www.domain.com/epics/21/stories"
        }
      ]
    }
  ]
}
```

优点：

- 如果API被更改，客户端依旧会获取有效的URL（只要保证在URL更改时更新链接）。
- API变得更具自描述性，客户端不必经常查找文档。

i. 恰当地设计关系

假设每个story都有一个epic和几个sub task。 在API中设计关系基本上有三种常用选项：链接，侧载和嵌入。

它们都是有效的，正确的选择取决于用例。 基本上，应根据客户端的访问模式以及可容忍的请求数量和有效负载大小来设计关系。

链接：

```
{
  "data": [
    {
      "id": 1,
      "name": "用户故事1",
      "relationships": {
        "epic": "http://www.domain.com/story/1/epic",
        "subTasks": [
          "http://www.domain.com/subTasks/12",
          "http://www.domain.com/subTasks/13"
        ]
        //or "subTasks": "http://www.domain.com/story/1/subTasks"
```

```
      }
    }
  ]
}
```

- 有效负载小。
- 许多请求。
- 客户端必须将数据拼接在一起才能获得所有数据。

侧载：

```
{
  "data": [
    {
      "id": 1,
      "name": "用户故事1",
      "relationships": {
        "epic":  5 ,
        "subTask": [ 12, 13 ]
      }
    }
  ],
  "included": {
    "epic": {
      "id": 5,
      "name": "epic5"
    },
    "subTasks": [
      { "id": 12, "name": "子任务12" }
      , { "id": 13, "name": "子任务13" }
    ]
  }
}
```

客户端还可以通过诸如GET /stories? include=epic，subTasks之类的查询参数来控制侧载实体。

- 一次请求。
- 定制的有效载荷大小。 没有重复（例如，即使被许多用户故事引用，也只用提供一次epic）
- 客户端仍然必须将数据拼接在一起以便解决关系，这可能非常麻烦。

嵌入：

```
{
  "data": [
    {
      "id": 1,
      "name": "用户故事1",
      "epic": {
        "id": 5,
        "name": "epic5"
      },
      "subTask": [
```

```
        { "id": 12, "name": "子任务12" }
        , { "id": 13, "name": "子任务13" }
      ]
    }
  ]
}
```

- 对客户来说最方便。 是可以直接通过关系来获取实际数据。
- 如果客户端不需要关系，白白加载关系。
- 有效负载大小和重复增加。 可能多次嵌入引用的实体。

j. 使用小驼峰命名法来命名属性

```
{ "epic.dateOfCreated": 2019-05-16 }
// 反例
epic.created_date // 违反JavaScript规范
epic.DateOfCreated // 建议用于构造方法
// 正例
epic.dateOfCreated
```

k. 使用动词进行操作

有时对API调用的响应不涉及资源（如计算，转义或变换）。 例：

```
// 读取
GET /translate?from=de_DE&to=en_US&text=Hallo
GET /calculate?para2=23&para2=432
// 触发更改服务器端状态的操作
POST /restartServer
// 无消息体
POST /banUserFromChannel
{ "user": "123", "channel": "serious-chat-channel" }
```

通过动词来表达RPC风格API，它比尝试RESTful风格来进行操作更简单，更直观（例如PATCH / server with {“restart”：true}）。REST风格非常适合与领域模型交互，RPC适合于操作。 更多信息请查看“Understanding RPC Vs REST For HTTP APIs”。

l. 分页

i. 基于偏移的分页

一般方法是使用参数offset和limit来进行分页：

```
# 返回30至45的epics
/epics?offset=30&limit=15
```

如果未填参数，则可使用默认值（offset=0， limit=100 ）：

```
# 返回0至100的epics
/epics
```

还可以在响应数据中，提供前一页和后一页的链接。

请求：

```
# 返回30至45的epics
/epics?offset=30&limit=15
```

响应：

```
{
  "pagination": {
    "offset": 20,
    "limit": 10,
    "total": 3465,
  },
  "data": [
    //...
  ],
  "links": {
    "next": "http://www.domain.com/epics?offset=30&limit=10",
    "prev": "http://www.domain.com/epics?offset=10&limit=10"
  }
}
```

基于偏移量的分页实现很简单，但是有两个缺点：

- 查询慢，数据量大时SQL偏移子句执行会很慢。
- 分页期间的变更。

ii. 基于键集的分页，又称继续令牌，也称为光标（推荐）

简单来说就是使用索引列来进行分页。 假设epic有一个索引列data_created，我们就可以使用data_created来分页。

```
GET /epics?pageSize=100
# 客户端接受最靠前的100条epic信息，使用`data_created`字段排序
# 该分页最老epic的`dataCreated` 字段值为 1504224000000 (= Sep 1, 2017
12:00:00 AM)
GET /epics?pageSize=100&createdSince=1504224000000
# 客户端请求1504224000000之后的100个epics数据。
# 该分页最前面的epic创建于1506816000000.
```

该分页方式解决了基于偏移的分页的许多缺点，但对调用方来说不太方便。
更好的方式是通过向日期添加附加信息（如id）来创建所谓的 continuation token，以提高可靠性和效率。 此外，应该向该令牌的有效负载中提供专用字段，以便客户端不用必须通过查看元素才能搞清楚。 甚至还可以进一步提供下一页链接。
因此 GET /epics?pageSize=100请求将返回如下：

```
{
  "pagination": {
    "continuationToken": "1504224000000_10",
  },
  "data": [
    // ...
    // last element:
    { "id": 10, "dateCreated": 1504224000000 }
  ],
  "links": {
```

```
    "next": "http://www.domain.com/epics?pageSize=100&continue=1504224000000_10"
  }
}
```

下一页链接使API真正成为RESTful风格，因为客户端只需通过这些链接（HATEOAS）即可查看集合。 无需手动构建URL。 此外，服务端可以简单地更改URL结构而不会破坏客户端，保证接口的演进性。

m. 确保API的可演进性

i. 避免破坏性变更

- 保持向后兼容。只要客户端能接受就通过添加字段的方式。
- 复制和弃用。要更改现有字段（重命名或更改结构），可在该字段旁边添加新字段，并在接口手册中弃用该字段。一段时间后，删除旧字段。
- 利用超媒体和HATEOAS。只要客户端使用响应中的链接来访问（且不会手动创建URL），即可以安全地更改URL而不会破坏客户端。
- 使用新名称创建新资源。如果新业务需求导致全新的领域模型和工作流，则可以创建新资源。

ii. 保持业务逻辑在服务侧

- 不要让服务成为转储数据访问层，它通过直接公开数据库模型（低级API）来提供CRUD功能。这造成了高耦合。
- 因此，我们应该构建高层次/基于工作流的API而不是低级API。

n. 版本化

API实在无法演进，则必须提供不同版本的API。版本控制允许在不破坏客户端的情况下，在新版本中发布不兼容和重大更改的API。

有两种最流行的版本控制方法：

- 通过URLs版本化
- 通过Accept HTTP Header进行版本控制（内容协商）

i. 通过URLs版本化

只需将API的版本号放在每个资源的URL中即可。

```
/v1/epics
```

优点：

- 对API开发人员非常简单。
- 对客户端访问也非常简单。
- 可以复制和粘贴URL。

缺点：

- 非RESTful。（该方式会令URL发生变化）
- 破坏URLs。 客户端必须维护和更新URL。

由于其简单性，该方式被各大厂商广泛使用，例如：Facebook, Twitter, Google/YouTube, Bing, Dropbox, Tumblr以及Disqus等。

ii. 通过Accept HTTP Header进行版本控制（内容协商）

更RESTFul的方式是利用通过Accept HTTP请求头的内容协商。

```
GET /epics
Accept: application/vnd.myapi.v2+json
```

优点：

- URLs保持不变
- RESTFul方式
- HATEOAS友好

缺点：

- 稍微难以使用。 客户必须注意标题。
- 无法再复制和粘贴网址。

2. 相关资料

a. REST

https://restfulapi.net/

b. RESTful API 设计规范

https://github.com/godruoyi/restful-api-specification

a. django-rest-framework-tutorial

https://github.com/twtrubiks/django-rest-framework-tutorial

d. 使用 Spring HATEOAS 开发 REST 服务

https://www.ibm.com/developerworks/cn/java/j-lo-SpringHATEOAS/

e. SOAP 和 REST的区别

https://smartbear.com/blog/test-and-monitor/soap-vs-rest-whats-the-difference/

f. SOAP Web服务教程

https://www.guru99.com/soap-simple-object-access-protocol.html