使用 Python 和 Flask 设计 RESTful API

使用 Python 和 Flask 设计 RESTful API - Designing a RESTful API with Python and Flask 1.0 documentation
http://www.pythondoc.com/flask-restful/first.html

近些年来 REST (REpresentational State Transfer) 已经变成了 web services 和 web APIs 的标配

在本文中我将向你展示如何简单地使用 Python 和 Flask 框架来创建一个 RESTful 的 web service。

什么是REST?

六条设计规范定义了一个REST系统的特点

- **客户端-服务器：** 客户端和服务器之间隔离，服务器提供服务，客户端进行消费。
- **无状态：** 从和护短到服务器的每个请求必须包含理解请求所必须的信息，换句话说，服务器不会储存客户端的上一次请求的信息来给下一次使用。
- **可缓存** 服务器必须明示客户端能否缓存
- **分层系统** 客户端和服务器之间的通信应该以一种标准的方式，就是中间层代替服务器做出响应的时候，客户端不需要做任何变动。
- **统一的接口** 服务器和客户端通信的方法必须是统一的。
- **按需编码** 服务器可以提供可执行代码或脚本，为客户端在它们的环境中执行，这个约束是唯一一个可选的。

什么是一个 RESTful 的 web service?

REST 架构的最初目的是适应万维网的 HTTP 协议。

RESTful web services 概念的核心就是“资源”。 资源可以用 URI 来表示。客户端使用 HTTP 协议定义的方法来发送请求到这些 URIs，当然可能会导致这些被访问的”资源“状态的改变。

HTTP 标准的方法有如下:

```
==========  ====================  ==================================
HTTP 方法   行为                  示例
==========  ====================  ==================================
GET         获取资源的信息         http://example.com/api/orders
GET         获取某个特定资源的信息 http://example.com/api/orders/123
POST        创建新资源             http://example.com/api/orders
PUT         更新资源               http://example.com/api/orders/123
DELETE      删除资源               http://example.com/api/orders/123
==========  ==================== ==================================
```

REST 设计不需要特定的数据格式。在请求中数据可以以 JSON 形式, 或者有时候作为 url 中查询参数项。

设计一个简单的 Web Service

坚持 REST 的准则设计一个 web service 或者 API 的任务就变成一个标识资源被展示出来以及它们是怎样受不同的请求方法影响的练习。

比如说，我们要编写一个待办事项应用程序而且我们想要为它设计一个 web service。要做的第一件事情就是决定用什么样的根 URL 来访问该服务。例如，我们可以通过这个来访问

http://[hostname]/todo/api/v1.0/

在这里我已经决定在 URL 中包含应用的名称以及 API 的版本号。在 URL 中包含应用名称有助于提供一个命名空间以便区分同一系统上的其它服务。在 URL 中包含版本号能够帮助以后的更新，如果新版本中存在新的和潜在不兼容的功能，可以不影响依赖于较旧的功能的应用程序。

在这里我已经决定在 URL 中包含应用的名称以及 API 的版本号。在 URL 中包含应用名称有助于提供一个命名空间以便区分同一系统上的其它服务。在 URL 中包含版本号能够帮助以后的更新，如果新版本中存在新的和潜在不兼容的功能，可以不影响依赖于较旧的功能的应用程序。

我们的任务资源将要使用 HTTP 方法如下:

```
==========  ================================================  ============================
HTTP 方法   URL                                               动作
==========  ================================================  ==============================
GET         http://[hostname]/todo/api/v1.0/tasks             检索任务列表
GET         http://[hostname]/todo/api/v1.0/tasks/[task_id]   检索某个任务
POST        http://[hostname]/todo/api/v1.0/tasks             创建新任务
PUT         http://[hostname]/todo/api/v1.0/tasks/[task_id]   更新任务
DELETE      http://[hostname]/todo/api/v1.0/tasks/[task_id]   删除任务
==========  ================================================ ============================
```

我们定义的任务有如下一些属性:

- **id**: 任务的唯一标识符。数字类型。
- **title**: 简短的任务描述。字符串类型。
- **description**: 具体的任务描述。文本类型。
- **done**: 任务完成的状态。布尔值。

目前为止关于我们的 web service 的设计基本完成。剩下的事情就是实现它!

Flask 框架的简介

如果你读过 Flask Mega-Tutorial 系列，就会知道 Flask 是一个简单却十分强大的 Python web 框架。

在我们深入研究 web services 的细节之前，让我们回顾一下一个普通的 Flask Web 应用程序的结构。

我会首先假设你知道 Python 在你的平台上工作的基本知识。 我将讲解的例子是工作在一个类 Unix 操作系统。简而言之，这意味着它们能工作在 Linux，Mac OS X 和 Windows(如果你使用Cygwin)。 如果你使用 Windows 上原生的 Python 版本的话，命令会有所不同。

让我们开始在一个虚拟环境上安装 Flask。如果你的系统上没有 virtualenv，你可以
从 https://pypi.python.org/pypi/virtualenv 上下载:

```
$ mkdir todo-api
$ cd todo-api
$ virtualenv flask
New python executable in flask/bin/python
Installing setuptools.............................done.
Installing pip...................done.
$ flask/bin/pip install flask
```

既然已经安装了 Flask，现在开始创建一个简单地网页应用，我们把它放在一个叫 app.py 的文件中:

```
#!flask/bin/python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return "Hello, World!"

if __name__ == '__main__':
    app.run(debug=True)
```

为了运行这个程序我们必须执行 app.py:

```
$ chmod a+x app.py
$ ./app.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

现在你可以启动你的网页浏览器，输入 http://localhost:5000 看看这个小应用程序的效果。

简单吧？现在我们将这个应用程序转换成我们的 RESTful service！

使用 Python 和 Flask 实现 RESTful services

使用Flask构建web services 是十分简单地，比我在 Mega-Tutorial 中构建的完整的服务端的应用程序要简单地多。

在 Flask 中有许多扩展来帮助我们构建 RESTful services，但是在我看来这个任务十分简单，没有必要使用 Flask 扩展。

我们 web service 的客户端需要添加、删除以及修改任务的服务，因此显然我们需要一种方式来存储任务。最直接的方式就是建立一个小型的数据库，但是数据库并不是本文的主体。学习在 Flask 中使用合适的数据库，我强烈建议阅读 Mega-Tutorial。

我们 web service 的客户端需要添加、删除以及修改任务的服务，因此显然我们需要一种方式来存储任务。最直接的方式就是建立一个小型的数据库，但是数据库并不是本文的主体。学习在 Flask 中使用合适的数据库，我强烈建议阅读 Mega-Tutorial。

我们现在来实现 web service 的第一个入口:

```
#!flask/bin/python
from flask import Flask, jsonify

app = Flask(__name__)

tasks = [
    {
        'id': 1,
        'title': u'Buy groceries',
        'description': u'Milk, Cheese, Pizza, Fruit, Tylenol',
        'done': False
    },
    {
        'id': 2,
        'title': u'Learn Python',
        'description': u'Need to find a good Python tutorial on the web',
        'done': False
    }
]

@app.route('/todo/api/v1.0/tasks', methods=['GET'])
def get_tasks():
    return jsonify({'tasks': tasks})
```

```
if __name__ == '__main__':
    app.run(debug=True)
```

正如你所见，没有多大的变化。我们创建一个任务的内存数据库，这里无非就是一个字典和数组。数组中的每一个元素都具有上述定义的任务的属性。

取代了首页，我们现在有一个 get_tasks 的函数，访问的URL为/todo/api/v1.0/tasks, 并且只允许GET的HTTP方法。

这个函数的响应不是文本，我们使用JSON数据格式来响应，Flask 的Jsonfy 函数从我们的数据结构中生成。

使用网页浏览器来测试我们的**web service** 不是一个最好的主义，因为浏览器不能轻易的模拟所有的HTTP请求方法，相反，我们会使用curl，如果你还没有安装curl清安装它。

通过执行app.py，启动web service。接着打开一个新的控制台窗口，运行以下命令：

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 294
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 04:53:53 GMT

{
  "tasks": [
    {
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "done": false,
      "id": 1,
      "title": "Buy groceries"
    },
    {
      "description": "Need to find a good Python tutorial on the web",
      "done": false,
      "id": 2,
      "title": "Learn Python"
    }
  ]
}
```

我们已经成功地调用我们的 RESTful service 的一个函数！

现在我们开始编写 GET 方法请求我们的任务资源的第二个版本。这是一个用来返回单独一个任务的函数:

```
from flask import abort

@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods=['GET'])
def get_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
        abort(404)
    return jsonify({'task': task[0]})
```

第二个函数有些意思。这里我们得到了 URL 中任务的 id，接着 Flask 把它转换成 函数中的 task_id 的参数。

我们用这个参数来搜索我们的任务数组。如果我们的数据库中不存在搜索的 id，我们将会返回一个类似 404 的错误，根据 HTTP 规范的意思是 “资源未找到”。

如果我们找到相应的任务，那么我们只需将它用 jsonify 打包成 JSON 格式并将其发送作为响应，就像我们以前那样处理整个任务集合。

调用 curl 请求的结果如下:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks/2
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 151
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:21:50 GMT

{
  "task": {
    "description": "Need to find a good Python tutorial on the web",
    "done": false,
    "id": 2,
    "title": "Learn Python"
  }
}
$ curl -i http://localhost:5000/todo/api/v1.0/tasks/3
HTTP/1.0 404 NOT FOUND
Content-Type: text/html
Content-Length: 238
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:21:52 GMT

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server.</p><p>If you     entered the URL manually please check your spelling and try
```

当我们请求 id #2 的资源时候，我们获取到了，但是当我们请求 #3 的时候返回了 404 错误。有关错误奇怪的是返回的是 HTML 信息而不是 JSON，这是因为 Flask 按照默认方式生成 404 响应。由于这是一个 Web service 客户端希望我们总是以 JSON 格式回应，所以我们需要改善我们的 404 错误处理程序:

```
from flask import make_response

@app.errorhandler(404)
def not_found(error):
    return make_response(jsonify({'error': 'Not found'}), 404)
```

我们会得到一个友好的错误提示:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks/3
HTTP/1.0 404 NOT FOUND
Content-Type: application/json
Content-Length: 26
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:36:54 GMT

{
  "error": "Not found"
}
```

接下来就是 POST 方法，我们用来在我们的任务数据库中插入一个新的任务:

```
from flask import request

@app.route('/todo/api/v1.0/tasks', methods=['POST'])
def create_task():
    if not request.json or not 'title' in request.json:
        abort(400)
    task = {
        'id': tasks[-1]['id'] + 1,
        'title': request.json['title'],
        'description': request.json.get('description', ""),
        'done': False
    }
    tasks.append(task)
    return jsonify({'task': task}), 201
```

添加一个新的任务也是相当容易地。只有当请求以 JSON 格式形式，request.json 才会有请求的数据。如果没有数据，或者存在数据但是缺少 title 项，我们将会返回 400，这是表示请求无效。

接着我们会创建一个新的任务字典，使用最后一个任务的 id + 1 作为该任务的 id。我们允许 description 字段缺失，并且假设 done 字段设置成 False。

我们把新的任务添加到我们的任务数组中，并且把新添加的任务和状态 201 响应给客户端。

使用如下的 curl 命令来测试这个新的函数:

```
$ curl -i -H "Content-Type: application/json" -X POST -d '{"title":"Read a book"}' http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 201 Created
Content-Type: application/json
Content-Length: 104
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:56:21 GMT

{
  "task": {
    "description": "",
    "done": false,
    "id": 3,
    "title": "Read a book"
  }
}
```

注意：如果你在 Windows 上并且运行 Cygwin 版本的 curl，上面的命令不会有任何问题。然而，如果你使用原生的 curl，命令会有些不同:

```
curl -i -H "Content-Type: application/json" -X POST -d "{""title"":""Read a book""}" http://localhost:5000/todo/api/v1.0/tasks
```

当然在完成这个请求后，我们可以得到任务的更新列表:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 423
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:57:44 GMT

{
  "tasks": [
    {
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "done": false,
      "id": 1,
      "title": "Buy groceries"
    },
    {
      "description": "Need to find a good Python tutorial on the web",
      "done": false,
      "id": 2,
      "title": "Learn Python"
    },
    {
      "description": "",
      "done": false,
      "id": 3,
      "title": "Read a book"
    }
  ]
}
```

剩下的两个函数如下所示:

```
@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods=['PUT'])
def update_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
        abort(404)
    if not request.json:
        abort(400)
    if 'title' in request.json and type(request.json['title']) != unicode:
        abort(400)
    if 'description' in request.json and type(request.json['description']) is not unicode:
        abort(400)
    if 'done' in request.json and type(request.json['done']) is not bool:
        abort(400)
    task[0]['title'] = request.json.get('title', task[0]['title'])
    task[0]['description'] = request.json.get('description', task[0]['description'])
    task[0]['done'] = request.json.get('done', task[0]['done'])
```

```
    return jsonify({'task': task[0]})

@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods=['DELETE'])
def delete_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
        abort(404)
    tasks.remove(task[0])
    return jsonify({'result': True})
```

delete_task 函数没有什么特别的。对于 update_task 函数，我们需要严格地检查输入的参数以防止可能的问题。我们需要确保在我们把它更新到数据库之前，任何客户端提供我们的是预期的格式。

更新任务 #2 的函数调用如下所示:

```
$ curl -i -H "Content-Type: application/json" -X PUT -d '{"done":true}' http://localhost:5000/todo/api/v1.0/tasks/2
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 170
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 07:10:16 GMT

{
  "task": [
    {
      "description": "Need to find a good Python tutorial on the web",
      "done": true,
      "id": 2,
      "title": "Learn Python"
    }
  ]
}
```

优化 web service 接口

目前 API 的设计的问题就是迫使客户端在任务标识返回后去构造 URIs。这对于服务器是十分简单的，但是间接地迫使客户端知道这些 URIs 是如何构造的，这将会阻碍我们以后变更这些 URIs。

不直接返回任务的 ids，我们直接返回控制这些任务的完整的 URI，以便客户端可以随时使用这些 URIs。为此，我们可以写一个小的辅助函数生成一个 “公共” 版本任务发送到客户端:

```
from flask import url_for

def make_public_task(task):
    new_task = {}
    for field in task:
        if field == 'id':
            new_task['uri'] = url_for('get_task', task_id=task['id'], _external=True)
        else:
            new_task[field] = task[field]
    return new_task
```

这里所有做的事情就是从我们数据库中取出任务并且创建一个新的任务，这个任务的 id 字段被替换成通过 Flask 的 url_for 生成的 uri 字段。

当我们返回所有的任务列表的时候，在发送到客户端之前通过这个函数进行处理:

```
@app.route('/todo/api/v1.0/tasks', methods=['GET'])
def get_tasks():
    return jsonify({'tasks': map(make_public_task, tasks)})
```

这里就是客户端获取任务列表的时候得到的数据:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 406
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 18:16:28 GMT

{
  "tasks": [
    {
      "title": "Buy groceries",
      "done": false,
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "uri": "http://localhost:5000/todo/api/v1.0/tasks/1"
    },
    {
      "title": "Learn Python",
      "done": false,
      "description": "Need to find a good Python tutorial on the web",
      "uri": "http://localhost:5000/todo/api/v1.0/tasks/2"
    }
  ]
}
```

我们将会把上述的方式应用到其它所有的函数上以确保客户端一直看到 URIs 而不是 ids。

加强 RESTful web service 的安全性

我们已经完成了我们 web service 的大部分功能，但是仍然有一个问题。我们的 web service 对任何人都是公开的，这并不是一个好主意。

我们有一个可以管理我们的待办事项完整的 web service，但在当前状态下的 web service 是开放给所有的客户端。 如果一个陌生人弄清我们的 API 是如何工作的，他或她可以编写一个客户端访问我们的 web service 并且毁坏我们的数据。

大部分初级的教程会忽略这个问题并且到此为止。在我看来这是一个很严重的问题，我必须指出。

确保我们的 web service 安全服务的最简单的方法是要求客户端提供一个用户名和密码。在常规的 web 应用程序会提供一个登录的表单用来认证，并且服务器会创建一个会话为登录的用户以后的操作使用，会话的 id 以 cookie 形式存储在客户端浏览器中。然而 REST 的规则之一就是 “无状态”， 因此我们必须要求客户端在每一次请求中提供认证的信息。

我们一直试着尽可能地坚持 HTTP 标准协议。既然我们需要实现认证我们需要在 HTTP 上下文中去完成，HTTP 协议提供了两种认证机制: Basic 和 Digest。

有一个小的 Flask 扩展能够帮助我们，我们可以先安装 Flask-HTTPAuth:

```
$ flask/bin/pip install flask-httpauth
```

比方说，我们希望我们的 web service 只让访问用户名 miguel 和密码 python 的客户端访问。 我们可以设置一个基本的 HTTP 验证如下:

```
from flask.ext.httpauth import HTTPBasicAuth
auth = HTTPBasicAuth()

@auth.get_password
def get_password(username):
    if username == 'miguel':
        return 'python'
    return None

@auth.error_handler
def unauthorized():
    return make_response(jsonify({'error': 'Unauthorized access'}), 401)
```

get_password 函数是一个回调函数，Flask-HTTPAuth 使用它来获取给定用户的密码。在一个更复杂的系统中，这个函数是需要检查一个用户数据库，但是在我们的例子中只有单一的用户因此没有必要。

error_handler 回调函数是用于给客户端发送未授权错误代码。像我们处理其它的错误代码，这里我们定制一个包含 JSON 数据格式而不是 HTML 的响应。

随着认证系统的建立，所剩下的就是把需要认证的函数添加 @auth.login_required 装饰器。例如:

```
@app.route('/todo/api/v1.0/tasks', methods=['GET'])
@auth.login_required
def get_tasks():
    return jsonify({'tasks': tasks})
```

如果现在要尝试使用 curl 调用这个函数我们会得到:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 401 UNAUTHORIZED
Content-Type: application/json
Content-Length: 36
WWW-Authenticate: Basic realm="Authentication Required"
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 06:41:14 GMT

{
  "error": "Unauthorized access"
}
```

为了能够调用这个函数我们必须发送我们的认证凭据:

```
$ curl -u miguel:python -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 316
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 06:46:45 GMT

{
  "tasks": [
    {
      "title": "Buy groceries",
      "done": false,
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "uri": "http://localhost:5000/todo/api/v1.0/tasks/1"
    },
    {
      "title": "Learn Python",
      "done": false,
      "description": "Need to find a good Python tutorial on the web",
      "uri": "http://localhost:5000/todo/api/v1.0/tasks/2"
    }
  ]
}
```

认证扩展给予我们很大的自由选择哪些函数需要保护，哪些函数需要公开。

为了确保登录信息的安全应该使用 HTTP 安全服务器(例如: https://...)，这样客户端和服务器之间的通信都是加密的，以防止传输过程中第三方看到认证的凭据。

为了确保登录信息的安全应该使用 HTTP 安全服务器(例如: https://...)，这样客户端和服务器之间的通信都是加密的，以防止传输过程中第三方看到认证的凭据。

为了确保登录信息的安全应该使用 HTTP 安全服务器(例如: https://...)，这样客户端和服务器之间的通信都是加密的，以防止传输过程中第三方看到认证的凭据。

```
@auth.error_handler
def unauthorized():
    return make_response(jsonify({'error': 'Unauthorized access'}), 403)
```

可能的改进

我们编写的小型的 web service 还可以在不少的方面进行改进。

对于初学者来说，一个真正的 web service 需要一个真实的数据库进行支撑。我们现在使用的内存数据结构会有很多限制不应该被用于真正的应用。

另外一个可以提高的领域就是处理多用户。如果系统支持多用户的话，不同的客户端可以发送不同的认证凭证获取相应用户的任务列表。在这样一个系统中的话，我们需要第二个资源就是用户。在用户资源上的 POST 的请求代表注册换一个新用户。一个 GET 请求表示客户端获取一个用户的信息。一个 PUT 请求表示更新用户信息，比如可能是更新邮箱地址。一个 DELETE 请求表示删除用户账号。

GET 检索任务列表请求可以在几个方面进行扩展。首先可以携带一个可选的页的参数，以便客户端请求任务的一部分。另外，这种扩展更加有用：允许按照一定的标准筛选。比如，用户只想要看到完成的任务，或者只想看到任务的标题以 A 字母开头。所有的这些都可以作为 URL 的一个参数项。