1. **实验名称：**

使用Python和Flask设计RESTful API

1. **实验目的**
2. 使用 Python 和 Flask 实现 RESTful services
3. 改进Web Service接口
4. RESTful web service的安全认证
5. **实验环境**

Ubuntu 22.04.1

1. **实验内容及步骤**
2. 使用 Python 和 Flask 实现 RESTful services

这个web service提供增加，删除、修改任务清单，所以我们需要将任务清单存储起来。最简单的做法就是使用小型的数据库，但是数据库并不是本文涉及太多的。在这里例子我们将任务清单存储在内存中。

1. 实现 web service 的第一个入口代码如下

#!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)

我们将任务清单存储在list内（内存），list存放两个非常简单的数组字典。每个实体就是我们上面定义的字段。而 index 入口点有一个get\_tasks函数与/todo/api/v1.0/tasks URI关联，只接受http的GET方法。

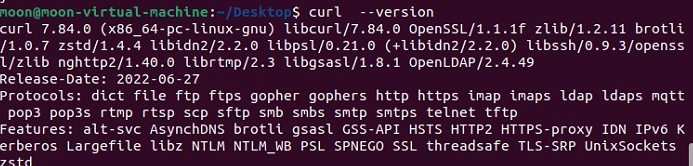
这个响应并非一般文本，是JSON格式的数据，是经过Flask框架的 jsonify模块格式化过的数据。

使用浏览器去测试web service并不是一个好的办法，因为要创建不同类弄的HTTP请求，事实上，我们将使用curl命令行。如果没有安装curl，快点去安装一个。

像刚才一样运行app.py。安装命令如下

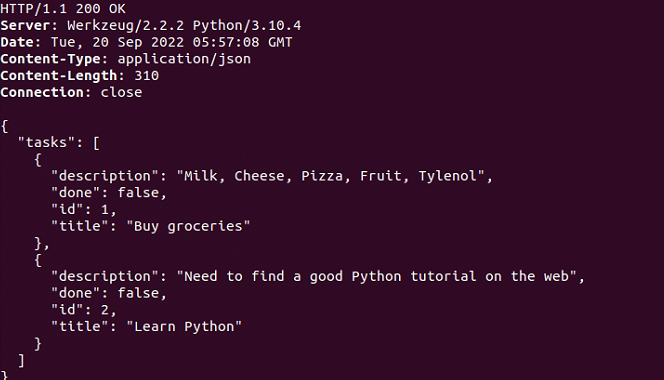
sudo apt install curl

输入 curl --version 确定是否安装成功



1. 打开一个终端运行以下命令调用 RESTful service方法

curl -i http://localhost:5000/todo/api/v1.0/tasks



这样就调用了一个RESTful service方法！

1. 写第二个版本的GET方法获取特定的任务。获取单个任务：

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)

task =list(task )

if len(task) == 0:

abort(404)

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

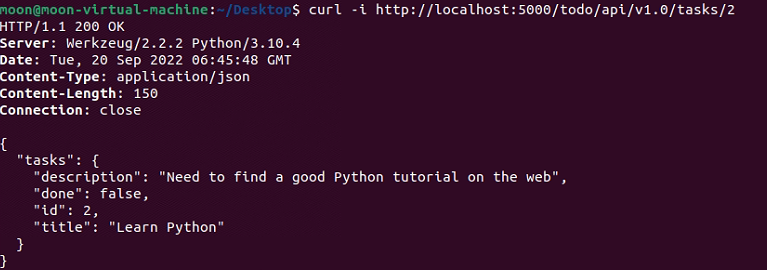
第二个函数将任务的id包含在URL内，Flask将task\_id参数传入了函数内。

通过参数，检索tasks数组。如果参数传过来的id不存在于数组内，我们需要返回错误代码404，按照HTTP的规定，404意味着是"Resource Not Found"，资源未找到。

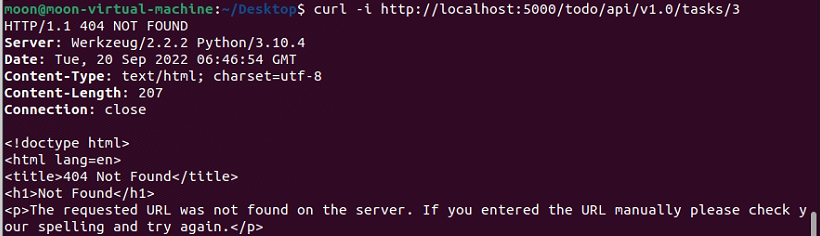
如果找到任务在内存数组内，我们通过jsonify模块将字典打包成JSON格式，并发送响应到客户端上。

调用 curl 请求的结果如下:

$ curl -i <http://localhost:5000/todo/api/v1.0/tasks/2>



$ curl -i http://localhost:5000/todo/api/v1.0/tasks/3



1. 改善我们的 404 错误处理程序:

当我们请求#2 id的资源时，可以获取，但是当我们请求#3的资源时返回了404错误。并且返回了一段奇怪的HTML错误，而不是我们期望的JSON，这是因为Flask产生了默认的404响应。客户端需要收到的都是JSON的响应，因此我们需要改进404错误处理：

from flask import make\_response

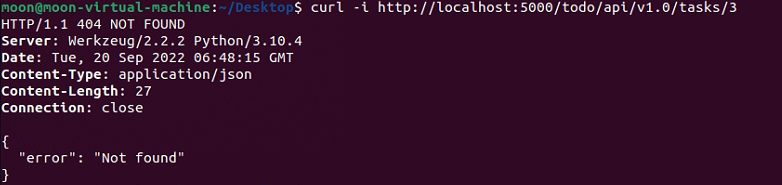
@app.errorhandler(404)

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



1. 使用 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

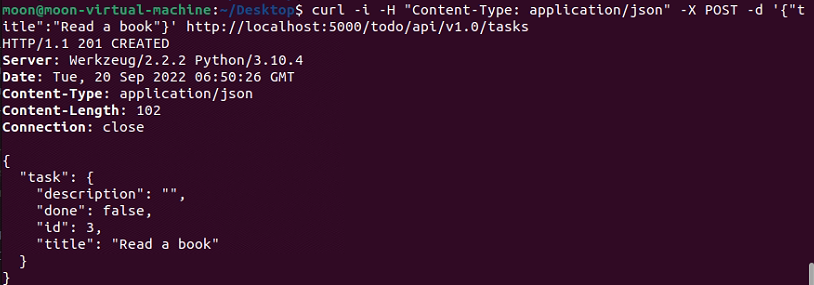
 request.json里面包含请求数据，如果不是JSON或者里面没有包括title字段，将会返回400的错误代码。

当创建一个新的任务字典，使用最后一个任务id数值加1作为新的任务id（最简单的方法产生一个唯一字段）。这里允许不带description字段，默认将done字段值为False。

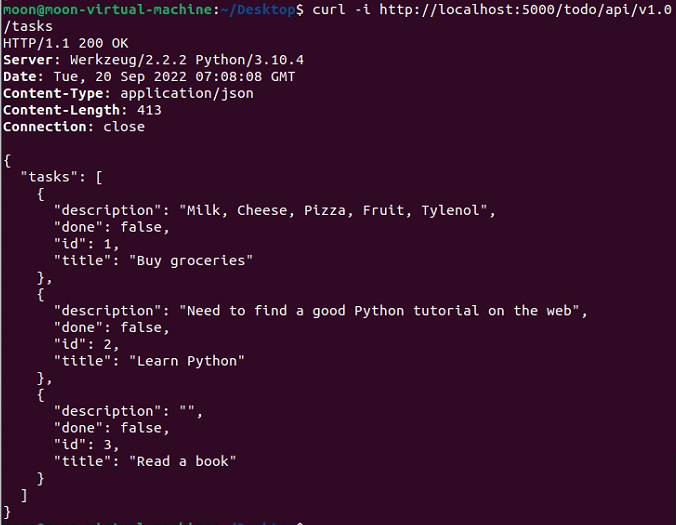
将新任务附加到tasks数组里面，并且返回客户端201状态码和刚刚添加的任务内容。HTTP定义了201状态码为“Created”。

1. 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

基完成上面的事情，就可以看到更新之后的list数组内容：

$ curl -i http://localhost:5000/todo/api/v1.0/tasks



1. 确保是预期的JSON格式写入数据库里面

@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)

task =list(task )

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)

task =list(task )

if len(task) == 0:

abort(404)

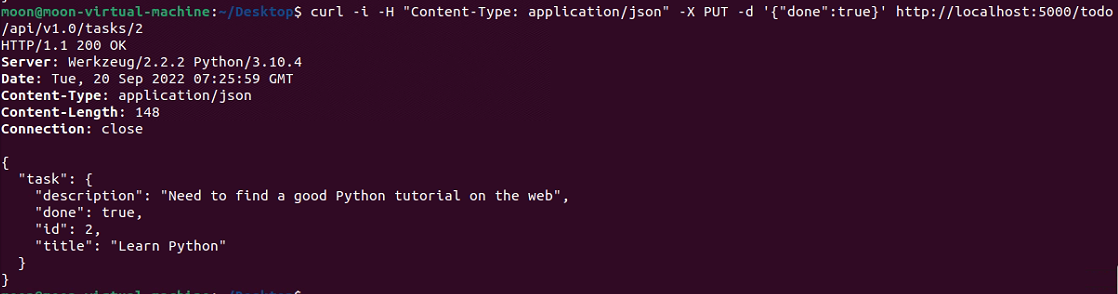
tasks.remove(task[0])

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。

1. 测试将任务#2的done字段变更为done状态：

$ curl -i -H "Content-Type: application/json" -X PUT -d '{"done":true}' http://localhost:5000/todo/api/v1.0/tasks/2



1. 改进Web Service接口

当前我们还有一个问题，客户端有可能需要从返回的JSON中重新构造URI，如果将来加入新的特性时，可能需要修改客户端。（例如新增版本。）

我们可以返回整个URI的路径给客户端，而不是任务的id。为了这个功能，创建一个小函数生成一个“public”版本的任务URI返回：

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

通过Flask的url\_for模块，获取任务时，将任务中的id字段替换成uri字段，并且把值改为uri值。

当我们返回包含任务的list时，通过这个函数处理后，返回完整的uri给客户端：

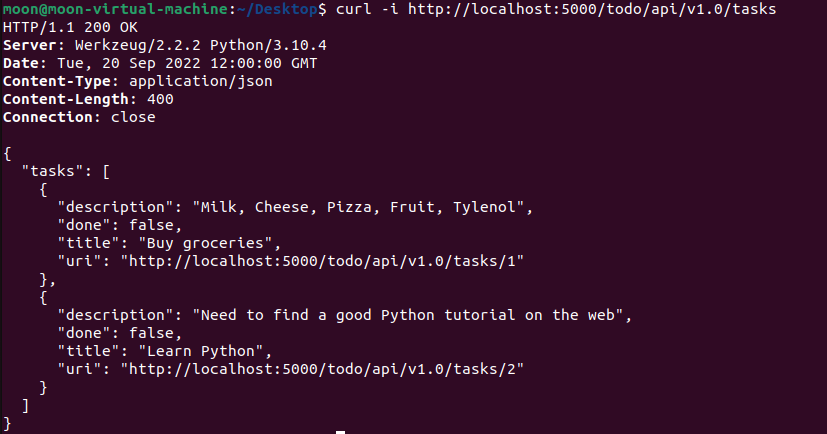
@app.route('/todo/api/v1.0/tasks', methods=['GET'])

def get\_tasks():

return jsonify({'tasks': list(map(make\_public\_task, tasks))})

现在看到的检索结果：

$ curl -i http://localhost:5000/todo/api/v1.0/tasks



这种办法避免了与其它功能的兼容，拿到的是完整uri而不是一个id。

1. RESTful web service的安全认证

我们已经完成了整个功能，但是我们还有一个问题。当前service是所有客户端都可以连接的，如果有别人知道了这个API就可以写个客户端随意修改数据了。

最简单的办法是在web service中，只允许用户名和密码验证通过的客户端连接。在一个常规的web应用中，应该有登录表单提交去认证，同时服务器会创建一个会话过程去进行通讯。这个会话过程id会被存储在客户端的cookie里面。不过这样就违返了我们REST中无状态的规则，因此，我们需求客户端每次都将他们的认证信息发送到服务器。

为此我们有两种方法表单认证方法去做，分别是 Basic 和 Digest。

这里有有个小Flask extension可以轻松做到。

（1）首先需要安装 [Flask-HTTPAuth](https://github.com/miguelgrinberg/flask-httpauth) ：

pip install flask-httpauth

（2）假设web service只有用户 ok 和密码为 python 的用户接入。

下面就设置了一个Basic HTTP认证：

from flask\_httpauth import HTTPBasicAuth

auth = HTTPBasicAuth()

@auth.get\_password

def get\_password(username):

if username == 'ok':

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，获取一个已知用户的密码。在复杂的系统中，函数是需要到数据库中检查的，但是这里只是一个小示例。

当发生认证错误之后，error\_handler回调函数会发送错误的代码给客户端。这里我们自定义一个错误代码401，返回JSON数据，而不是HTML。

（3）将@auth.login\_required装饰器添加到需要验证的函数上面：

@app.route('/todo/api/v1.0/tasks', methods=['GET'])

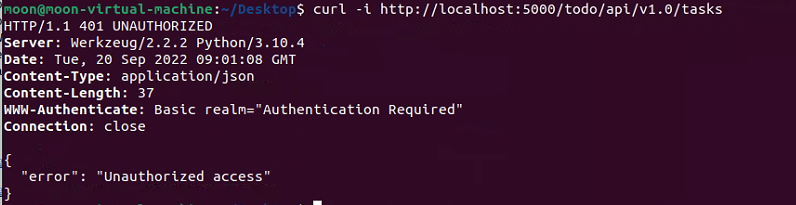
@auth.login\_required

def get\_tasks():

return jsonify({'tasks': tasks})

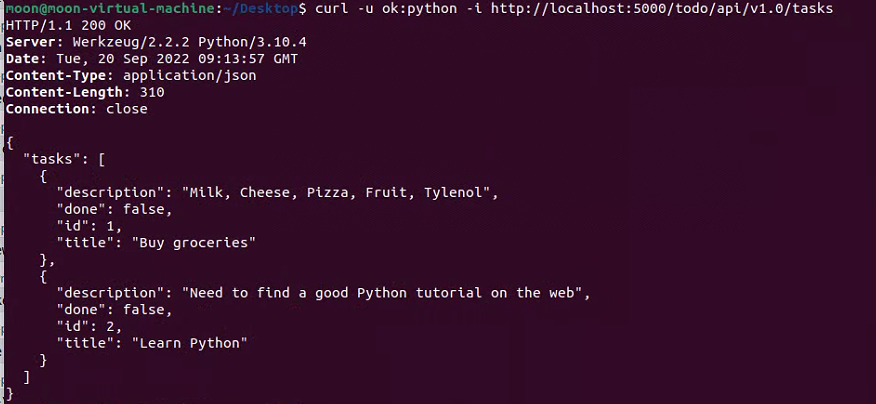
（4）试试使用curl调用这个函数：

$ curl -i http://localhost:5000/todo/api/v1.0/tasks



这里表示了没通过验证，下面是带用户名与密码的验证：

$ curl -u ok:python -i http://localhost:5000/todo/api/v1.0/tasks



这个认证extension十分灵活，可以随指定需要验证的APIs。

为了确保登录信息的安全，最好的办法还是使用https加密的通讯方式，客户端与服务器端传输认证信息都是加密过的，防止第三方的人去看到。

当使用浏览器去访问这个接口，会弹出一个登录对话框，如果密码错误就回返回401的错误代码。为了防止浏览器弹出验证对话框，客户端应该处理好这个登录请求。

有一个小技巧可以避免这个问题，就是修改返回的错误代码401。例如修改成403（”Forbidden“）就不会弹出验证对话框了。

@auth.error\_handler

def unauthorized():

return make\_response(jsonify({'error': 'Unauthorized access'}), 403)

当然，同时也需要客户端知道这个403错误的意义。